

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Notes

FOR B.TECH III YEAR - II SEM(R17)

(2019-20)



MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade -
ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100,
Telangana State, INDIA.

Sno	Subject Code	Subject Name
1	R17A0522	DISTRIBUTED SYSTEMS
2	R17A0518	OBJECT ORIENTED ANALYSIS AND DESIGN
3	R17A0464	EMBEDDED SYSTEMS
4	R17A0520	SOFTWARE TESTING METHODOLOGIES
5	R17A0554	PYTHON PROGRAMMING
6	R17A0519	WEB TECHNOLOGIES

Distributed Systems

[R17A0522]

LECTURE NOTES

B.TECH III YEAR – II SEM (R17) (2019-20)



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12 (B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – ‘A’ Grade -
ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State,
India.

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

III Year B. Tech CSE -IISem

L T/P/D C

3 -/-/ 3

Core Elective 2 (R17A0522) Distributed systems

Objectives:

- To learn the principles, architectures, algorithms and programming models used in distributed systems.
- To examine state-of-the-art distributed systems, such as Google File System.
- To design and implement sample distributed systems.

UNIT I

Characterization of Distributed Systems: Introduction, Examples of Distributed systems, Resource Sharing and Web, Challenges.

System Models: Introduction, Architectural models, Fundamental models.

UNIT II

Time and Global States: Introduction, Clocks, Events and Process states, Synchronizing Physical clocks, Logical time and Logical clocks, Global states.

Coordination and Agreement: Introduction, Distributed mutual exclusion, Elections, Multicast Communication, Consensus and Related problems.

UNIT III

Interprocess Communication: Introduction, Characteristics of Interprocess communication, External Data Representation and Marshalling, Client-Server Communication, Group Communication, Case Study: IPC in UNIX.

Distributed Objects and Remote Invocation: Introduction, Communication between Distributed Objects, Remote Procedure Call, Events and Notifications, Case study: Java RMI.

UNIT IV

Distributed File Systems: Introduction, File service Architecture, Case Study: 1: Sun Network File System, Case Study 2: The Andrew File System.

Distributed Shared Memory: Introduction, Design and Implementation issues, Consistency Models.

UNIT V

Transactions and Concurrency Control: Introduction, Transactions, Nested Transactions, Locks, Optimistic concurrency control, Timestamp ordering, Comparison of methods for concurrency control.

Distributed Transactions: Introduction, Flat and Nested Distributed Transactions, Atomic commit protocols, Concurrency control in distributed transactions, Distributed deadlocks, Transaction recovery.

TEXT BOOKS:

1. Distributed Systems Concepts and Design, G Coulouris, J Dollimore and T Kindberg, Fourth Edition, Pearson Education. 2009.

REFERENCES:

1. Distributed Systems, Principles and paradigms, Andrew S.Tanenbaum, Maarten Van Steen, Second Edition,PHI.
2. Distributed Systems, An Algorithm Approach, Sikumar Ghosh, Chapman & Hall/CRC, Taylor & Fransis Group,2007.

Course Outcomes:

1. Able to compare different types of distributed systems and different models.
2. Able to analyze the algorithms of mutual exclusion, election & multicast communication.
3. Able to evaluate the different mechanisms for Interprocess communication and remote invocations.
4. Able to design and develop new distributed applications.
5. Able to apply transactions and concurrency control mechanisms in different distributed environments.

Index

Unit	Topic	Page
I	Characterization of Distributed Systems	1
	System Models	7
II	Time and Global States	18
	Coordination and Agreement	26
III	Inter Process Communication	38
	Distributed Objects and Remote Invocation	51
IV	Distributed File Systems	58
	Distributed Shared Memory	64
V	Transactions and Concurrency Control	68
	Distributed Transactions	74

UNIT I

Characterization of Distributed Systems: Introduction, Examples of Distributed systems, Resource Sharing and Web, Challenges.

System Models: Introduction, Architectural models, Fundamental models.

Characterization of Distributed Systems: Introduction

A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages.

A distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. This simple definition covers the entire range of systems in which networked computers can usefully be deployed.

Computers that are connected by a network may be spatially separated by any distance. They may be on separate continents, in the same building or in the same room. Our definition of distributed systems has the following significant consequences:

Concurrency: In a network of computers, concurrent program execution is the norm. I can do my work on my computer while you do your work on yours, sharing resources such as web pages or files when necessary. The capacity of the system to handle shared resources can be increased by adding more resources (for example. computers) to the network.

No global clock: When programs need to cooperate they coordinate their actions by exchanging messages. Close coordination often depends on a shared idea of the time at which the programs' actions occur. But it turns out that there are limits to the accuracy with which the computers in a network can synchronize their clocks – there is no single global notion of the correct time. This is a direct consequence of the fact that the *only* communication is by sending messages through a network.

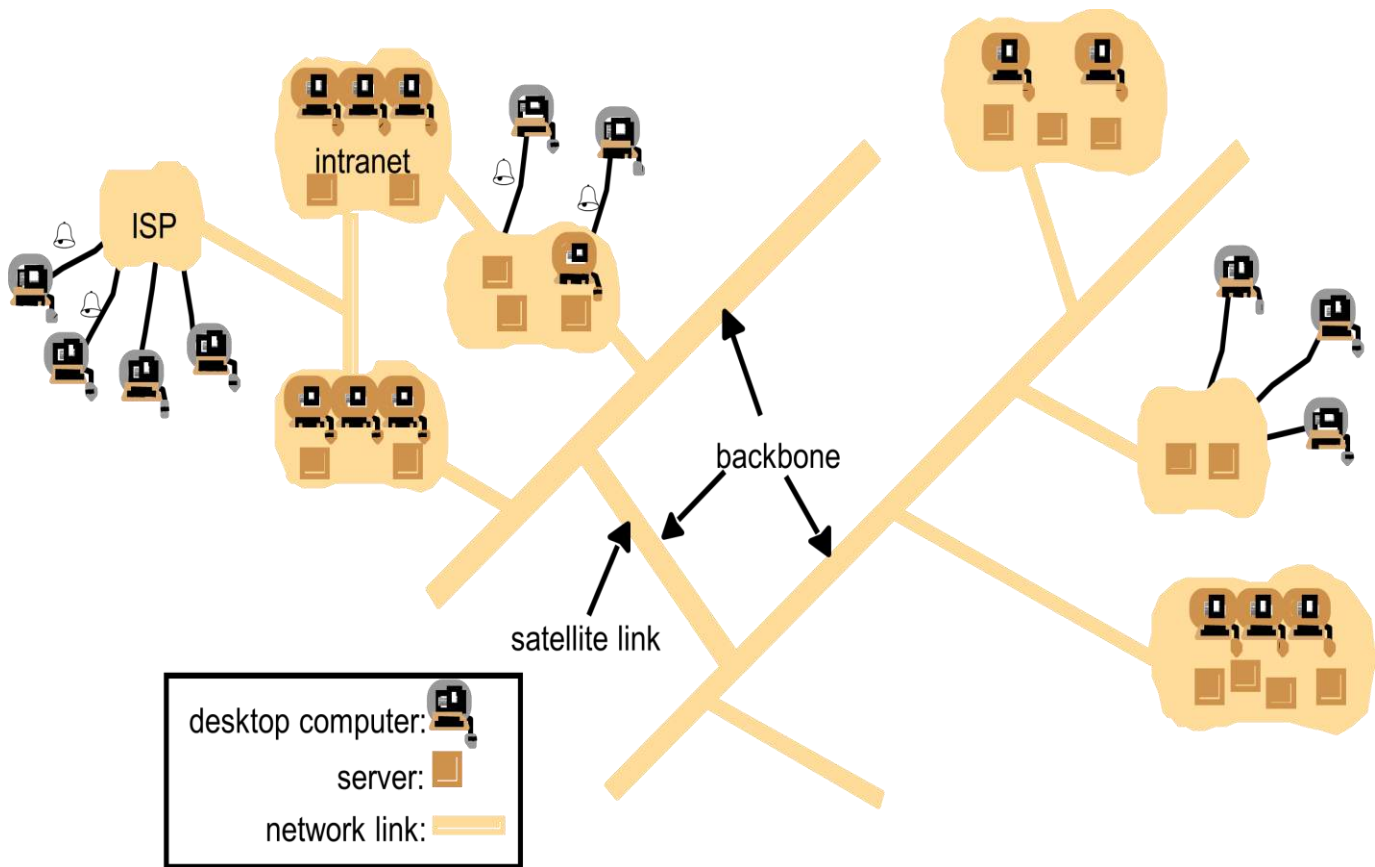
Independent failures: All computer systems can fail, and it is the responsibility of system designers to plan for the consequences of possible failures. Distributed systems can fail in new ways. Faults in the network result in the isolation of the computers that are connected to it, but that doesn't mean that they stop running. In fact, the programs on them may not be able to detect whether the network has failed or has become unusually slow. Similarly, the failure of a computer, or the unexpected termination of a program somewhere in the system (a *crash*), is not immediately made known to the other components with which it communicates. Each component of the system can fail independently, leaving the others still running.

Examples of Distributed systems

To place distributed systems in a realistic context through examples: the Internet, an intranet and mobile computing.

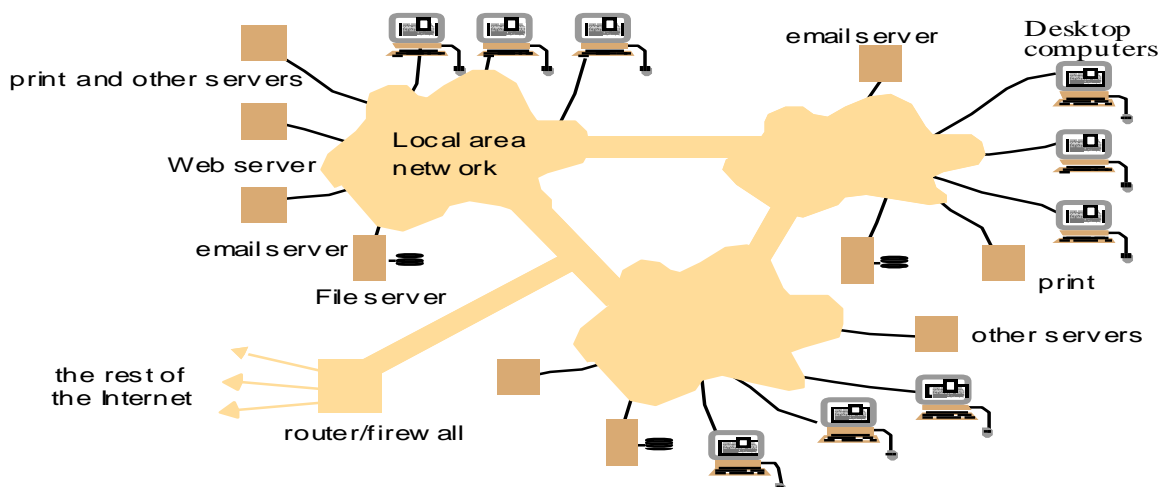
1. The Internet (Figure 1) :

- A vast interconnected collection of computer networks of many different types.
- Passing message by employing a common means of communication (Internet Protocol).
- The web is not equal to the Internet.



2. Intranets (Figure 2):

- An intranet is a private network that is contained within an enterprise.
- It may consist of many interlinked local area networks and also use leased lines in the Wide Area Network.
- It separately administrated and enforces local security policies.
- It is connected to the Internet via a router
- It uses firewall to protect an Intranet by preventing unauthorized messages leaving or entering
- Some are isolated from the Internet
- Users in an intranet share data by means of file services.

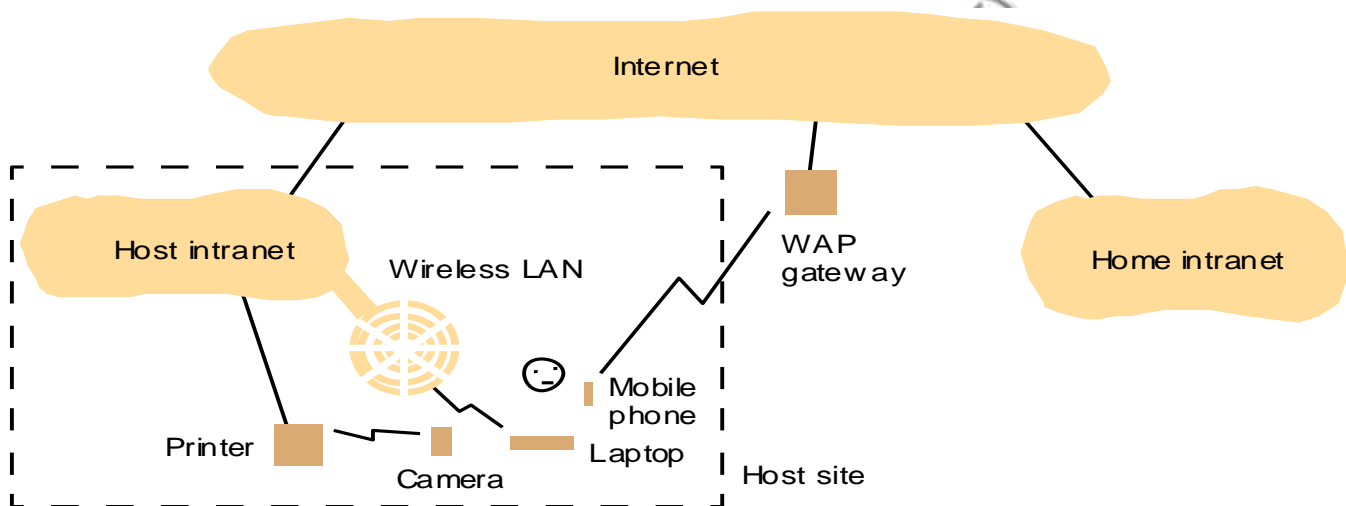


3. Mobile and Ubiquitous Computing (Figure 1.3)

- a. Distributed systems techniques are equally applicable to mobile computing involving laptops, PDAs and wearable computing devices.
- b. Mobile computing (nomadic computing) - perform of computing tasks while moving (nomadic computing)
- c. Ubiquitous computing - small computers embedded in appliances
 - i. harness of many small, cheap computation devices
 - ii. It benefits users while they remain in a single environment such as home.

Distributed In Figure 3 user has access to three forms of wireless connection:

- d. A laptop is connected to host's wireless LAN.
- e. A mobile (cellular) phone is connected to Internet using Wireless Application Protocol (WAP) via a gateway.
- f. A digital camera is connected to a printer over an infra-red link.



Resource Sharing and Web

- Equipments are shared to reduce cost. Data shared in database or web pages are high-level resources which are more significant to users without regard for the server or servers that provide these.
- Patterns of resource sharing vary widely in their scope and in how closely users work together:
 - Search Engine: Users need no contact between users
 - Computer Supported Cooperative Working (CSCW): Users cooperate directly share resources. Mechanisms to coordinate users' action are determined by the pattern of sharing and the geographic distribution.
- For effective sharing, each resource must be managed by a program that offers a communication interface enabling the resource to be accessed and updated reliably and consistently.
- Server is a running program (a process) on a networked computer that accepts requests from programs running on other computers to perform a service and responds appropriately .
- The requesting processes are referred to as a client.

- An executing web browser is a client. It communicates with a web server to request web pages from it.
- When a client invokes an operation upon the server, it is called the remote invocation.
- Resources may be encapsulated as objects and accessed by client objects. In this case a client object invokes a method upon a server object.

The World Wide Web (WWW)

- WWW is an evolving system for publishing and accessing resources and services across Internet. Web is an open system. Its operations are based on freely published communication standards and documents standards.
- Key feature: Web provides a hypertext structure among the documents that it stores. The documents contain links - references to other documents or resources. The structures of links can be arbitrarily complex and the set of resources that can be added is unlimited.
- Three main standard technological components:
 - HTML (Hypertext Markup Language) specify the contents and layout of web pages.
 - Contents: text, table, form, image, links, information for search engine, ...;
 - Layout: text format, background, frame, ...
 - URL (Uniform Resource Location): identify a resource to let browser find it.
 - scheme : scheme-specific-location
 - <http://web.cs.twsu.edu/> (HyperText Transfer Protocol)
 - URL (continued):
 - <ftp://ftp.twsu.edu/> (File Transfer Protocol)
 - <telnet://kirk.cs.twsu.edu> (log into a computer)
 - <mailto:chang@cs.twsu.edu> (identify a user's email address)
 - HTTP (HyperText Transfer Protocol) defines a standard rule by which browsers and any other types of client interact with web servers. Main features:
 - Request-reply interaction
 - Content types may or may not be handled by browser - using plug-in or external helper
 - One resource per request - Several requests can be made concurrently.
 - Simple access control
 - Services and dynamic pages
 - form - Common Gateway Interface program on server (Perl)
 - JavaScript (download from server and run on local computer)
 - Applet (download from server and run on local computer)

Challenges

As distributed systems are getting complex, developers face a number of challenges:

- Heterogeneity
- Openness
- Security
- Scalability
- Failure handling
- Concurrency
- Transparency
- Quality of service

Heterogeneity:

The Internet enables users to access services and run applications over a heterogeneous collection of computers and networks. Heterogeneity (that is, variety and difference) applies to all of the following:

- Hardware devices: computers, tablets, mobile phones, embedded devices, etc.
- Operating System: Ms Windows, Linux, Mac, Unix, etc.
- Network: Local network, the Internet, wireless network, satellite links, etc.
- Programming languages: Java, C/C++, Python, PHP, etc.
- Different roles of software developers, designers, system managers

Different programming languages use different representations for characters and data structures such as arrays and records. These differences must be addressed if programs written in different languages are to be able to communicate with one another. Programs written by different developers cannot communicate with one another unless they use common standards, for example, for network communication and the representation of primitive data items and data structures in messages. For this to happen, standards need to be agreed and adopted – as have the Internet protocols.

Middleware : The term middleware applies to a software layer that provides a programming abstraction as well as masking the heterogeneity of the underlying networks, hardware, operating systems and programming languages. Most middleware is implemented over the Internet protocols, which themselves mask the differences of the underlying networks, but all middleware deals with the difference in operating systems and hardware

Heterogeneity and mobile code : The term mobile code is used to refer to program code that can be transferred from one computer to another and run at the destination – Java applets are an example. Code suitable for running on one computer is not necessarily suitable for running on another because executable programs are normally specific both to the instruction set and to the host operating system.

Transparency:

Transparency is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components. In other words, distributed systems designers must hide the complexity of the systems as much as they can.

- 8 forms of transparency:
 - Access transparency – access to local and remote resources using identical operations
 - Location transparency – access to resources without knowing the physical location of the machine
 - Concurrency transparency – several processes operate concurrently without interfering each other
 - Replication transparency – replication of resources in multiple servers. Users are not aware of the replication
 - Failure transparency – concealment of faults, allows users to complete their tasks without knowing of the failures

- Mobility transparency – movement of resources and clients within a system without affecting users operations
- Performance transparency – systems can be reconfigured to improve performance by considering their loads
- Scaling transparency – systems and applications can be expanded without changing the structure or the application algorithms

Openness

The openness of a computer system is the characteristic that determines whether the system can be extended and re-implemented in various ways. The openness of distributed systems is determined primarily by the degree to which new resource-sharing services can be added and be made available for use by a variety of client programs. If the well-defined interfaces for a system are published, it is easier for developers to add new features or replace sub-systems in the future. Example: Twitter and Facebook have API that allows developers to develop their own software interactively.

Concurrency

Both services and applications provide resources that can be shared by clients in a distributed system. There is therefore a possibility that several clients will attempt to access a shared resource at the same time. For example, a data structure that records bids for an auction may be accessed very frequently when it gets close to the deadline time. For an object to be safe in a concurrent environment, its operations must be synchronized in such a way that its data remains consistent. This can be achieved by standard techniques such as semaphores, which are used in most operating systems.

Security

Many of the information resources that are made available and maintained in distributed systems have a high intrinsic value to their users. Their security is therefore of considerable importance. Security for information resources has three components:

confidentiality (protection against disclosure to unauthorized individuals)

integrity (protection against alteration or corruption),

availability for the authorized (protection against interference with the means to access the resources).

Scalability

Distributed systems must be scalable as the number of user increases. The scalability is defined by B. Clifford Neuman as

A system is said to be scalable if it can handle the addition of users and resources without suffering a noticeable loss of performance or increase in administrative complexity

Scalability has 3 dimensions:

- Size
 - Number of users and resources to be processed. Problem associated is overloading
- Geography
 - Distance between users and resources. Problem associated is communication reliability
- Administration
 - As the size of distributed systems increases, many of the system needs to be controlled. Problem associated is administrative mess

Failure Handling

Computer systems sometimes fail. When faults occur in hardware or software, programs may produce incorrect results or may stop before they have completed the intended computation. The handling of failures is particularly difficult.

- Dealing with failures in distributed systems:

- Detecting failures – known/unknown failures
- Masking failures – hide the failure from become severe. E.g. retransmit messages, backup of file data
- Tolerating failures – clients can be designed to tolerate failures – e.g. inform users of failure and ask them to try later
- Recovery from failures - recover and rollback data after a server has crashed
- Redundancy- the way to tolerate failures – replication of services and data in multiple servers

Quality of service

- The main nonfunctional properties of distributed systems that affect the quality of service experienced by users or clients are: reliability, security, performance, adaptability.
- Reliability
- Security
- Performance
- Adaptability

System Models: Introduction

- Architectural Models
 - Client-Server Model
 - Peer-Peer Model
- Fundamental Models
 - Interaction Model
 - Failure Model
 - Security Model

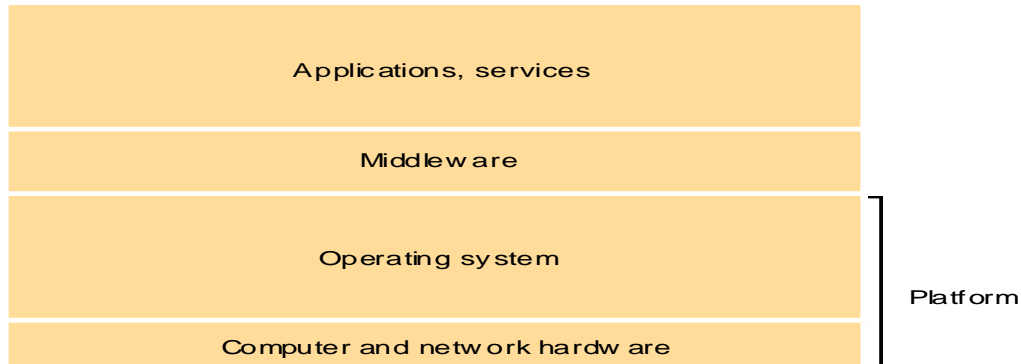
Architectural Models:

- An architectural model of a distributed system is concerned with the placement of its parts and the relationships between them.
- The architecture of a system is its structure in terms of separately specified components.
- The overall goal is to ensure that the structure will meet present and likely future demands on it.
- Major concerns are to make the system:
 - Reliable
 - Manageable
 - Adaptable
 - Cost-effective
- An architectural Model of a distributed system first simplifies and abstracts the functions of the individual components of a distributed system.
- An initial simplification is achieved by classifying processes as:
 - Server processes
 - Client processes
 - Peer processes
 - Cooperate and communicate in a symmetric manner to perform a task.

Software Layers

- Software architecture referred to:
 - The structure of software as layers or modules in a single computer.
 - The services offered and requested between processes located in the same or different computers.

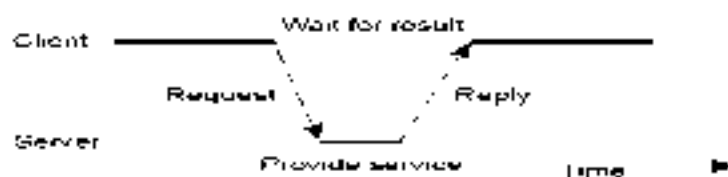
- Software architecture is breaking up the complexity of systems by designing them through layers and services.
 - Layer: a group of related functional components.
 - Service: functionality provided to the next layer.



- Platform
 - The lowest-level hardware and software layers are often referred to as a platform for distributed systems and applications.
 - ❖ These low-level layers provide services to the layers above them, which are implemented independently in each computer.
 - ❖ These low-level layers bring the system's programming interface up to a level that facilitates communication and coordination between processes.
- Middleware
 - A layer of software whose purpose is
 - ❖ to mask heterogeneity presented in distributed systems.
 - ❖ To provide a convenient programming model to application developers.
 - Major Examples of middleware are:
 - ❖ Sun RPC (Remote Procedure Calls)
 - ❖ OMG CORBA (Common Request Broker Architecture)
 - ❖ Microsoft D-COM (Distributed Component Object Model)
 - ❖ Sun Java RMI

Client-Server model

- Most often architecture for distributed systems.
- Client process interact with individual server processes in a separate host computers in order to access the shared resources
- Servers may in turn be clients of other servers.
 - ❖ E.g. a web server is often a client of a local file server that manages the files in which the web pages are stored.
 - ❖ E.g. a search engine can be both a server and a client: it responds to queries from browser clients and it runs web crawlers that act as clients of other web servers.



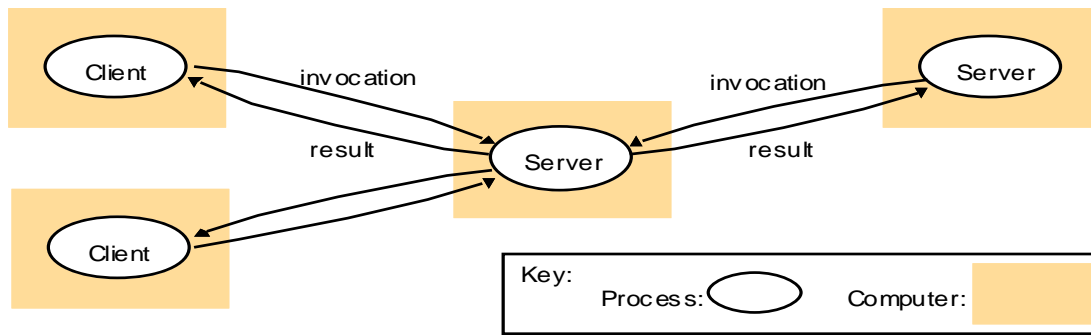
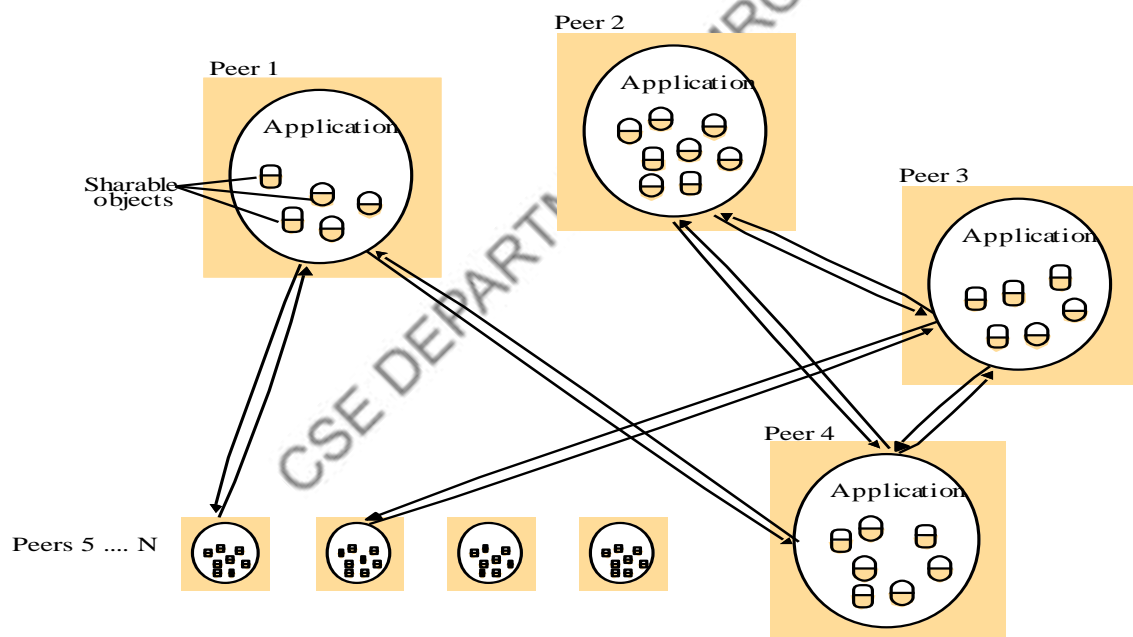


Figure 4. Clients invoke individual servers

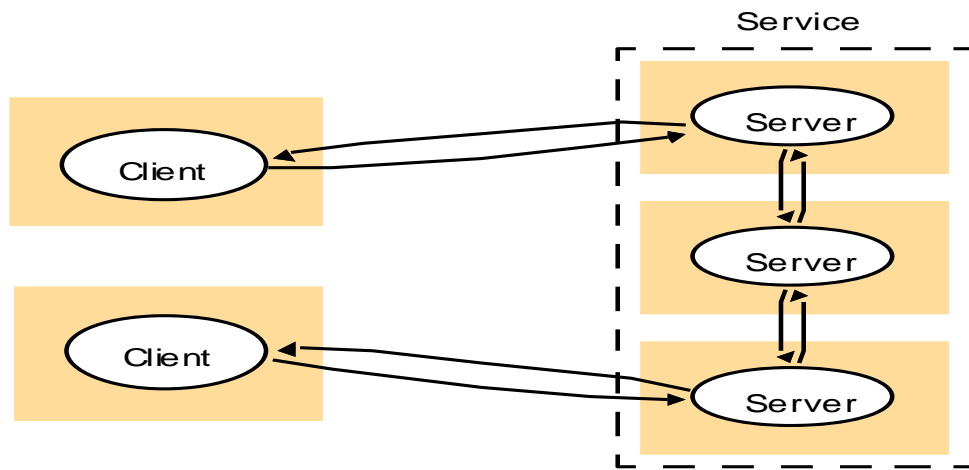
Peer-to-Peer model

- All of the processes play similar roles, interacting cooperatively as peers to perform a distributed activities or computations without any distinction between clients and servers or the computers that they run on.
- E.g., music sharing systems Napster



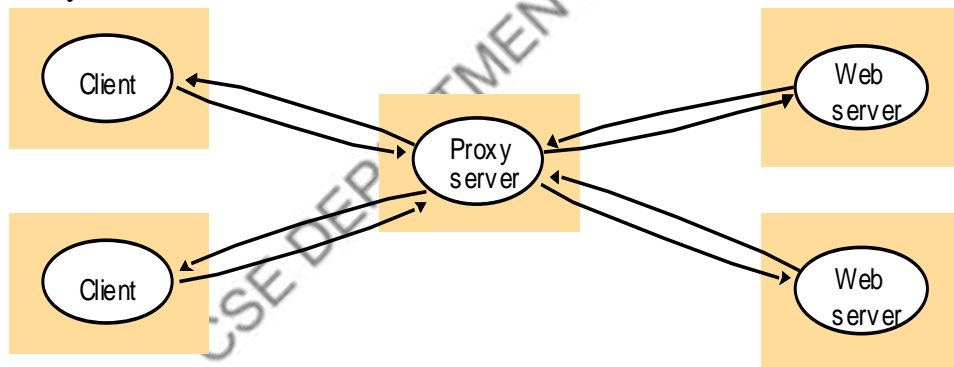
Variants of Client Sever Model

- The problem of client-server model is placing a service in a server at a single address that does not scale well beyond the capacity of computer host and bandwidth of network connections.
- To address this problem, several variations of client-server model have been proposed.
- Some of these variations are discussed in the next slide.
- **Services provided by multiple servers**
 - Services may be implemented as several server processes in separate host computers interacting as necessary to provide a service to client processes.
 - E.g. cluster that can be used for search engines.



▪ Proxy servers and caches

- A cache is a store of recently used data objects.
- When a new object is received at a computer it is added to the cache store, replacing some existing objects if necessary.
- When an object is needed by a client process the caching service first checks the cache and supplies the object from there if an up-to-date copy is available.
- If not, an up-to-date copy is fetched.
- Caches may be collected with each client or they may be located in a proxy server that can be shared by several clients.

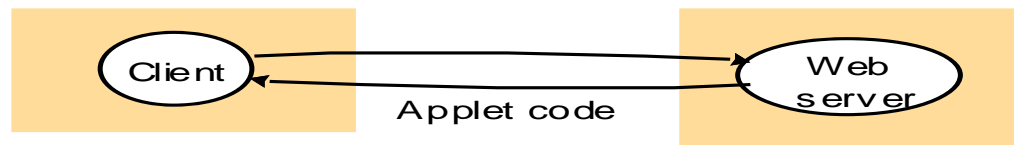


▪ Mobile code

- Applets are a well-known and widely used example of mobile code.
- Applets downloaded to clients give good interactive response
- Mobile codes such as Applets are a potential security threat to the local resources in the destination computer.
- Browsers give applets limited access to local resources. For example, by providing no access to local user file system.

E.g. a stockbroker might provide a customized service to notify customers of changes in the prices of shares; to use the service, each customer would have to download a special applet that receives updates from the broker's server, display them to the user and perhaps performs automatic to buy and sell operations triggered by conditions set up by the customer and stored locally in the customer's computer

a) client request results in the downloading of applet code



b) client interacts with the applet



▪ Mobile agents

- A running program (code and data) that travels from one computer to another in a network carrying out of a task, usually on behalf of some other process.
- Examples of the tasks that can be done by mobile agents are:
 - ❖ To collecting information.
 - ❖ To install and maintain software maintain on the Computers within an organization.
 - ❖ To compare the prices of products from a number of vendors.
 - ❖ Mobile agents are a potential security threat to the resources in computers that they visit.
 - ❖ The environment receiving a mobile agent should decide on which of the local resources to be allowed to use.
 - ❖ Mobile agents themselves can be vulnerable
 - ❖ They may not be able to complete their task if they are refused access to the information they need.

➔ Network computers

- ❖ It downloads its operating system and any application software needed by the user from a remote file server.
- ❖ Applications are run locally but the file are managed by a remote file server.
- ❖ Network applications such as a Web browser can also be run.

➔ Thin clients

- ❖ It is a software layer that supports a window-based user interface on a computer that is local to the user while executing application programs on a remote computer.
- ❖ This architecture has the same low management and hardware costs as the network computer scheme.
- ❖ Instead of downloading the code of applications into the user's computer, it runs them on a compute server.
- ❖ Compute server is a powerful computer that has the capacity to run large numbers of application simultaneously.
- ❖ The compute server will be a multiprocessor or cluster computer running a multiprocessor version of an operation system such as UNIX or Windows.

➔ Performance Issues

- ❖ Performance issues arising from the limited processing and communication capacities of computers and networks are considered under the following subheading:
 - ❖ Responsiveness

- ❖ E.g. a web browser can access the cached pages faster than the non-cached pages.
- ❖ Throughput
- ❖ Load balancing
- ❖ E.g. using applets on clients, remove the load on the server.

→ Quality of service

- The ability of systems to meet deadlines.
- It depends on availability of the necessary Computing and network resources at the appropriate time.
- This implies a requirement for the system to provide guaranteed computing and communication resources that are sufficient to enable applications to complete each task on time.
 - ❖ E.g. the task of displaying a frame of video

Fundamental Models:

- Fundamental Models deal with a more formal description of the properties that are common in all of the architectural models.
- Fundamental Models are concerned with a more formal description of the properties that are common in all of the architectural models.
- All architectural models are composed of processes that communicate with each other by sending messages over a computer networks.
- Aspects of distributed systems that are discussed in fundamental models are:

Interaction model:

- Computation occurs within processes.
- The processes interact by passing messages, resulting in:
 - Communication (information flow)
 - Coordination (synchronization and ordering of activities) between processes
 - Interaction model reflects the facts that communication takes place with delays.
- Distributed systems are composed of many processes, interacting in the following ways:
 - Multiple server processes may cooperate with one another to provide a service
 - E.g. Domain Name Service
 - A set of peer processes may cooperate with one another to achieve a common goal
 - E.g. voice conferencing
- Two significant factors affecting interacting processes in a distributed system are:
 - Communication performance is often a limiting characteristic.
 - It is impossible to maintain a single global notion of time.
- Performance of communication channels
 - The communication channels in our model are realized in a variety of ways in distributed systems, for example
 - By an implementation of streams
 - By simple message passing over a computer network
 - Communication over a computer network has the performance characteristics such as:
 - Latency
 - The delay between the start of a message's transmission from one process to the beginning of its receipt by another.

- **Bandwidth**
 - The total amount of information that can be transmitted over a computer network in a given time.
 - Communication channels using the same network, have to share the available bandwidth.
- **Jitter**
 - The variation in the time taken to deliver a series of messages.
 - It is relevant to multimedia data.
 - For example, if consecutive samples of audio data are played with differing time intervals then the sound will be badly distorted.
- **Two variants of the interaction model**
 - In a distributed system it is hard to set time limits on the time taken for process execution, message delivery or clock drift.
 - Two models of time assumption in distributed systems are:
 - ❖ **Synchronous distributed systems**
 - It has a strong assumption of time
 - The time to execute each step of a process has known lower and upper bounds.
 - Each message transmitted over a channel is received within a known bounded time.
 - Each process has a local clock whose drift rate from real time has a known bound.
 - ❖ **Asynchronous distributed system**
 - It has no assumption about time.
 - There is no bound on process execution speeds.
 - ❑ Each step may take an arbitrary long time.
 - There is no bound on message transmission delays.
 - ❑ A message may be received after an arbitrary long time.
 - There is no bound on clock drift rates.
 - ❑ The drift rate of a clock is arbitrary.
- **Event ordering**
 - In many cases, we are interested in knowing whether an event (sending or receiving a message) at one process occurred before, after, or concurrently with another event at another process.
 - The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks.
 - ❖ For example, consider a mailing list with users X, Y, Z, and A.
 - ❖ User X sends a message with the subject Meeting.
 - 1. Users Y and Z reply by sending a message with the subject RE: Meeting.
 - In real time, X's message was sent first, Y reads it and replies; Z reads both X's message and Y's reply and then sends another reply, which references both X's and Y's messages.
 - But due to the independent delays in message delivery, the messages may be delivered in the order is shown in figure 10.
 - It shows user A might see the two messages in the wrong order.

- ❖ E.g. message contents may be corrupted or non-existent messages may be delivered or real messages may be delivered more than once.
- ❖ The omission failures are classified together with arbitrary failures shown in

<i>Class of failure</i>	<i>Affects</i>	<i>Description</i>
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a <i>send</i> , but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

▪ **Timing failure**

- Timing failures are applicable in synchronized distributed systems where time limits are set on process execution time, message delivery time and clock drift rate.

<i>Class of Failure</i>	<i>Affects</i>	<i>Description</i>
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

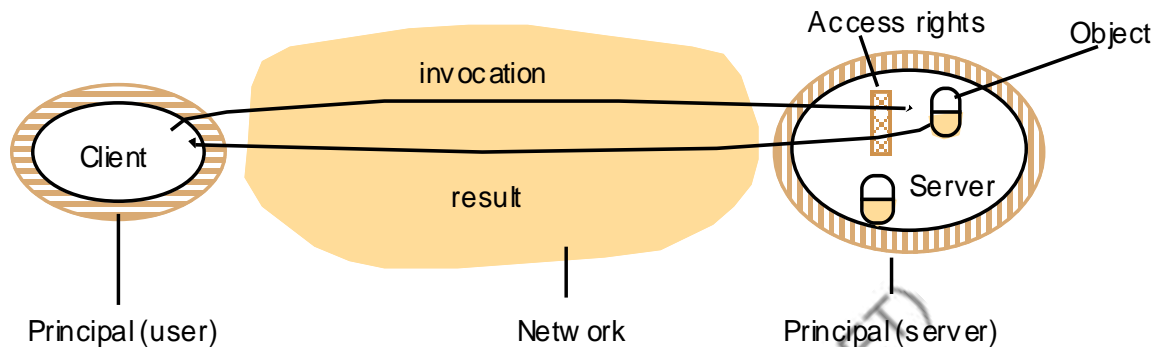
▪ **Masking failure**

- It is possible to construct reliable services from components that exhibit failure.
 - ❖ E.g. multiple servers that hold replicas of data can continue to provide a service when one of them crashes.
- A service masks a failure, either by hiding it altogether or by converting it into a more acceptable type of failure.
 - ❖ E.g. checksums are used to mask corrupted messages- effectively converting an arbitrary failure into an omission failure.

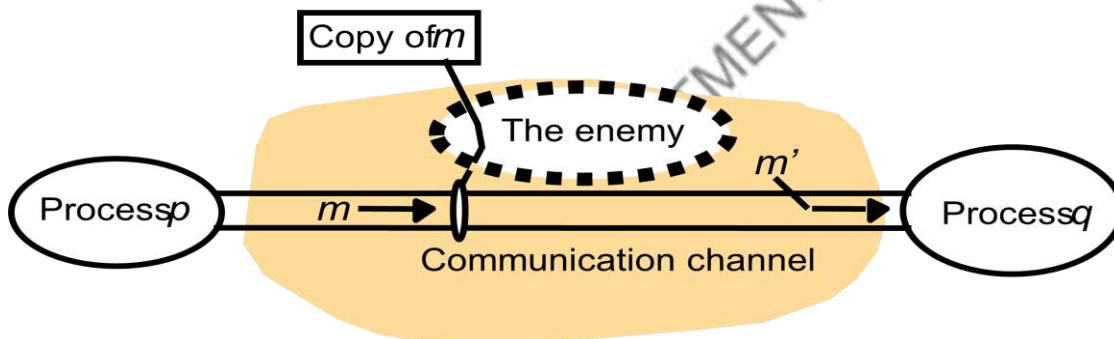
Security model

- Security model defines and classifies the forms of attacks.
- It provides a basis for analysis of threats to a system
- It is used to design of systems that are able to resist threats.
- The security of a distributed system can be achieved by securing the processes and the channels used in their interactions.
- Also, by protecting the objects that they encapsulate against unauthorized access.
- Protecting Objects
 - Access rights
 - Access rights specify who is allowed to perform the operations on a object.
 - Who is allowed to read or write its state.

- Principal
 - Principal is the authority associated with each invocation and each result.
 - A principal may be a user or a process.
 - The invocation comes from a user and the result from a server.
- The sever is responsible for
 - Verifying the identity of the principal (user) behind each invocation.
 - Checking that they have sufficient access rights to perform the requested operation on the particular object invoked.
 - Rejecting those that do not.



- The enemy
 - To model security threats, we assume an enemy that is capable of sending any message to any process and reading or copying any message between a pair of processes.



- Threats from a potential enemy are classified as:
 - ❖ Threats to processes
 - ❖ Threats to communication channels
 - ❖ Denial of service
- Defeating security threats
- Secure systems are based on the following main techniques:
 - ❖ Cryptography and shared secrets
 - Cryptography is the science of keeping message secure.
 - Encryption is the process of scrambling a message in such a way as to hide its contents.
 - ❖ Authentication
 - The use of shared secrets and encryption provides the basis for the authentication of messages.
 - ❖ Secure channels
 - Encryption and authentication are use to build secure channels as a service layer on top of the existing communication services.

- A secure channel is a communication channel connecting a pair of processes, each of which acts on behalf of a principal.
- VPN (Virtual Private Network) and secure socket layer (SSL) protocols are instances of secure channel.
- A secure channel has the following properties:
 - » Each of the processes knows the identity of the principal on whose behalf the other process is executing.
 - » In a secure channel, the server knows the identity of the principal behind the invocations and can check their access rights before performing an operation.
 - » A secure channel ensures the privacy and integrity of the data transmitted across it.
 - » Each message includes a physical or logical time stamp to prevent messages from being replayed or reordered.
- Other possible threats from an enemy
 - Denial of service
 - ❖ This is a form of attack in which the enemy interferes with the activities of authorized users by making excessive and pointless invocations on services of message transmissions in a network.
 - ❖ It results in overloading of physical resources (network bandwidth, server processing capacity).
 - Mobile code
 - ❖ Mobile code is security problem for any process that receives and executes program code from elsewhere, such as the email attachment.
 - ❖ Such attachment may include a code that accesses or modifies resources that are available to the host process but not to the originator of the code

UNIT II

Time and Global States: Introduction, Clocks, Events and Process states, Synchronizing Physical clocks, Logical time and Logical clocks, Global states.

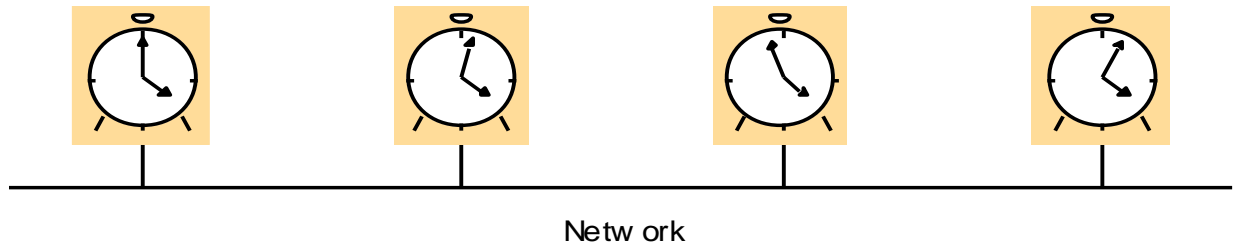
Coordination and Agreement: Introduction, Distributed mutual exclusion, Elections, Multicast Communication, Consensus and Related problems.

Time and Global States: Introduction

- Time is an Important and interesting issue in distributed systems.
- One we can measure accurately.
- Can use as a metric.
- Example: e-commerce transaction timestamp for auditing and accountability purposes.
- Need time for maintaining consistency of databases.
- According to Einstein's theory of relativity a stationary observer on earth and an observer moving away from earth, if they observed an event at the same time, the event may "happen" at different times for them
- Relative order of two events can be reversed unless one caused the other.
- Thus the notion of physical time is problematic in distributed systems.
- We will examine synchronization of clocks using message passing;
- We will study logical clocks: vector clocks.
- We will also look at algorithms to capture global states of distributed systems as they execute.

Clocks, event and process states

- History (pi) = hi = $\langle e_i^0, e_i^1, e_i^2 \dots \rangle$
- \rightarrow_i represents relation
- A processor clock is derived from a hardware clock:
- $C_i(t) = \alpha H_i(t) + \beta$
- May result in clock skew and drift
- Coordinated universal time: most accurate clock use atomic oscillators with very low drift rates of 1 in 10^{13}
- In use since 1967: standard second is defined as 9,192,631,770 periods of transitions between the two hyperfine levels of the ground state of Caesium-133 (Cs^{133})
- Days, hours, years are rooted in astronomical time.
- Japanese earthquake should have caused Earth to rotate a bit faster, shortening the length of the day by about 1.8 microseconds (a microsecond is one millionth of a second).
- Currents within earth's core also affect decreases and increases.
- Abbreviated UTC (is equivalent to Greenwich Mean Time (GMT)).
- This is based on atomic time (leap second is inserted, rarely leap second is deleted to keep up astronomical time).
- UTC signal are regularly synchronized and broadcast.
- In USA the radio station WWV broadcasts time signals on several shortwave frequencies.
- Satellite sources for time include Global Positioning Systems (GPS).
- NIST is another source.



- ➔ Each node maintain a physical clock. However, they tend to drift even after an accurate initial setting.
 - z **Skew:** the difference between the readings of any two clocks.
 - z **Clock drift:** the crystal-based clock count time at different rates. Oscillator has different frequency. Drift rate is usually used to measure the change in the offset per unit of time. Ordinary quartz crystal clock, 1second per 11.6 days.

Synchronizing Physical clocks

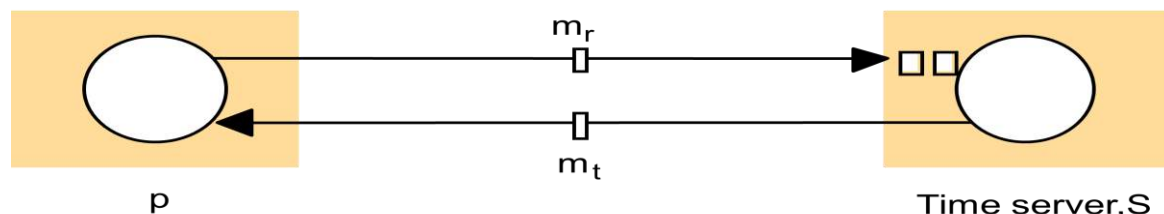
- ➔ External synchronization: C_i is synchronized to a common standard.
 - z $|S(t) - C_i(t)| < D$, for $i = 1, 2, \dots, N$ and for all real time t , namely clock C_i are accurate to within the bound D . S is standard time.
- ➔ Internal synchronization: C_i is synchronized with one another to a known degree of accuracy.
 - z $|C_i(t) - C_j(t)| < D$ for $i, j = 1, 2, \dots, N$, and for all real time t , namely, clocks C_i agree with each other within the bound D .

Synchronization in a synchronous system

- ➔ In a synchronous system, bounds exist for clock drift rate, transmission delay and time for computing of each step.
- ➔ One process sends the time t on its local clock to the other in a message m . The receiver should set its clock to $t + T_{\text{trans}}$. It doesn't matter whether t is accurate or not
 - o Synchronous system: T_{trans} could range from min to max. The uncertainty is $u = (\text{max} - \text{min})$. If receiver set clock to be $t + \text{min}$ or $t + \text{max}$, the skew is as much as u . If receiver set the clock to be $t + (\text{min} + \text{max})/2$, the skew is at most $u/2$.
 - o Asynchronous system: no upper bound max. only lower bound.

Cristian's method

- o Cristian's method: Time server, connected to a device receiving signals from UTC. Upon request, the server S supplies the time t according to its clock.
- o The algorithm is probabilistic and can achieve synchronization only if the observed round trip time are short compared with required accuracy.
- o From p 's point of view, the earliest time S could place the time in m_t was min after p dispatch m_r . The latest was min before m_t arrived at p .



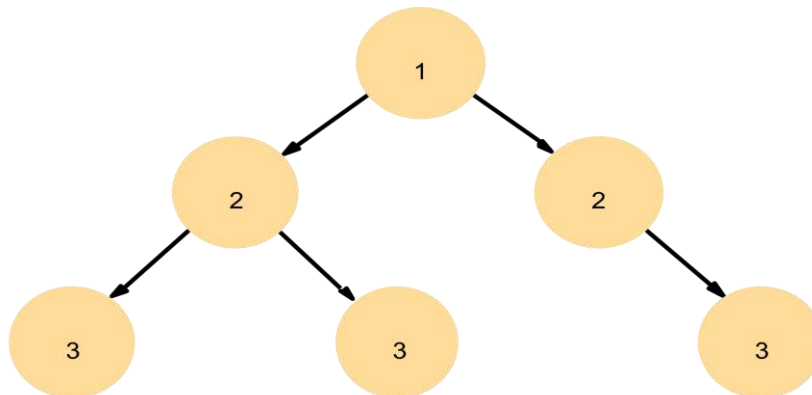
- The time of S by the time p receives the message mt is in the range of $[t + \min, t + T_{round} - \min]$.
- P can measure the roundtrip time then p should set its time as $(t + T_{round}/2)$ as a good estimation.
- The width of this range is $(T_{round} - 2\min)$. So the accuracy is $\pm(T_{round}/2 - \min)$
- Suffers from the problem associated with single server that single time server may fail.
- Cristian suggested to use a group of synchronized time servers. A client multicast is request to all servers and use only the first reply.
- A faulty time server that replies with spurious time values or an imposter time server with incorrect times.

Berkeley Algorithm

- Internal synchronization when developed for collections of computers running Berkeley UNIX.
- A coordinator is chosen to act as the master. It periodically polls the other computers whose clocks are to be synchronized, called slave. The slaves send back their clock values to it. The master estimate their local clock times by observing the round-trip time similar to Cristian's method. It averages the values obtained including its own.
- Instead of sending the updated current time back to other computers, which further introduce uncertainty of message transmission, the master sends the amount by which each individual slave's clock should adjust.
- The master takes a fault-tolerant average, namely a subset of clocks is chosen that do not differ from one another by more than a specified bound.
- The algorithm eliminates readings from faulty clocks. Such clocks could have a adverse effect if an ordinary average was taken.

The Network Time Protocol

- Cristian's method and Berkeley algorithm are primarily for Intranets. The Network Time Protocol(NTP) defines a time service to distribute time information over the Internet.
 - Clients across the Internet to be synchronized accurately to UTC. Statistical techniques
 - Reliable service that can survive lengthy losses of connectivity. Redundant servers and redundant paths between servers.
 - Clients resynchronized sufficiently frequently to offset the rates of drift.
 - Protection against interference with time services. Authentication technique from claimed trusted sources.



Note: Arrows denote synchronization control, numbers denote strata.

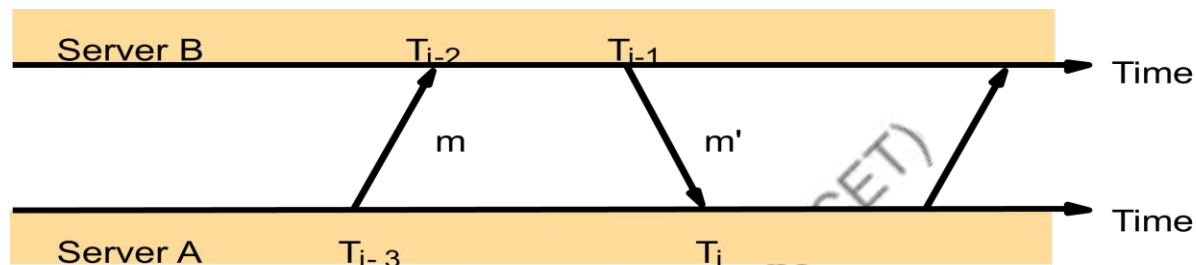
Hierarchical structure called synchronization subnet

- Primary server: connected directly to a time source.
- Secondary servers are synchronized with primary server.
- Third servers are synchronized with secondary servers.

Such subnet can reconfigure as servers become unreachable or failures occur.

NTP servers synchronize in one of three modes:

1. Multicast mode: for high-speed LAN. One or more servers periodically multicasts the time to servers connected by LAN, which set their times assuming small delay. Achieve low accuracy.
2. Procedure call: similar to Cristian's algorithm. One server receives request, replying with its timestamp. Higher accuracy than multicast or multicast is not supported.
3. Symmetric mode: used by servers that supply time in LAN and by higher level of synchronization subnet. Highest accuracy. A pair of servers operating in symmetric mode exchange messages bearing timing information.



Each message bears timestamps of recent message events: the local times when the previous NTP message between the pair was sent and received, and the local time when the current message was transmitted. The recipient of the NTP message notes the local time when it receives the message.

Logical time and Logical clocks

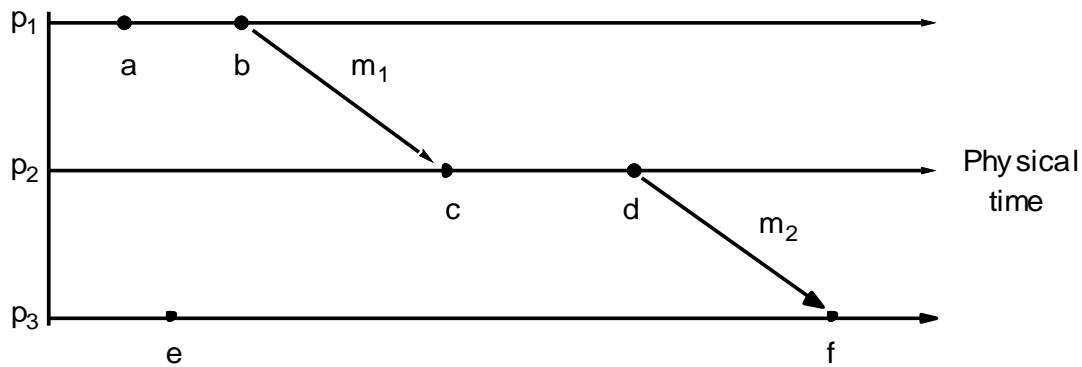
- In single process, events are ordered by local physical time. Since we cannot synchronize physical clocks perfectly across a distributed system, we cannot use physical time to find out the order of any arbitrary pair of events.
- We will use logical time to order events happened at different nodes. Two simple points:
 - If two events occurred at the same process, then they occurred in the order in which p_i observes them
 - Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message.
- Lamport (1978) called the partial ordering by generalizing these two relationships the happened-before relation.

HB1: If \exists process $p_i : e \rightarrow_i e'$, then $e \rightarrow e'$

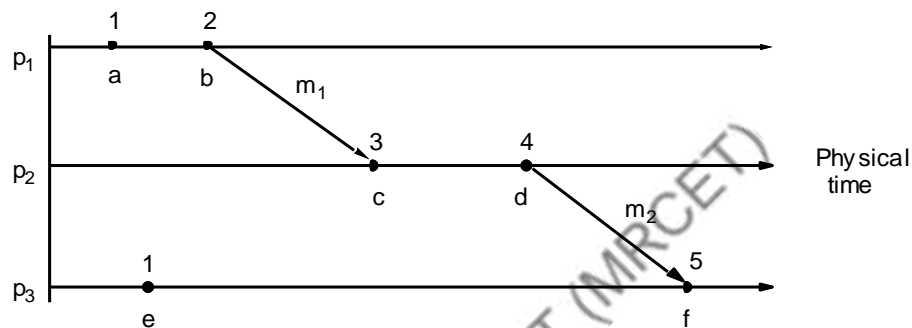
HB2: For any message m , $send(m) \rightarrow receive(m)$

HB3: If e, e' , and e'' are events, such that $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$

Events occurred at three processes



Lamport timestamps for the events



$a \rightarrow_1 b$ and $c \rightarrow_2 d$

$b \rightarrow c$ and $d \rightarrow f$

combining them, $a \rightarrow f$

$a \not\rightarrow e$ and $e \not\rightarrow a$
concurrent $a \parallel e$

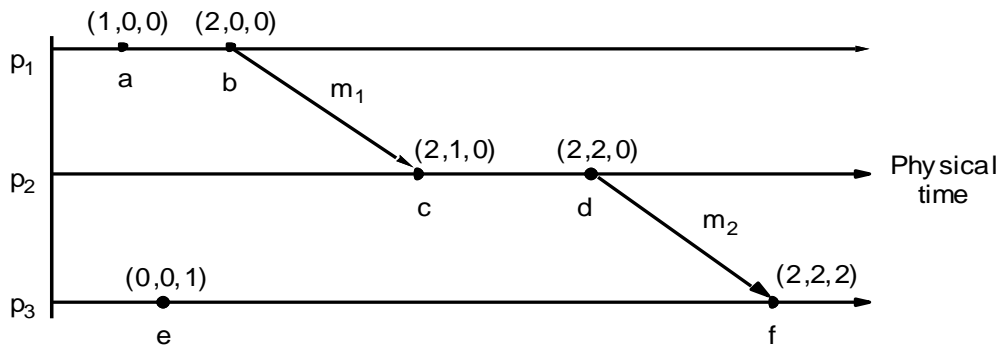
Logical Clocks

- Lamport invented a logical clock L_i , which is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock. Each process p_i keeps its own logical clock.
- LC1: L_i is incremented before each event is issued at process p_i : $L_i = L_i + 1$
- LC2: a) P_i sends a message m , it piggybacks on m the value $t = L_i$
b) On receiving (m, t) , a process p_j computes $L_j = \max(L_j, t)$ and then applies LC1 before timestamping the event $\text{receive}(m)$.
- It can be easily shown that:
- If $e \rightarrow e'$ then $L(e) < L(e')$.
- However, the converse is not true. If $L(e) < L(e')$, then we cannot infer that $e \rightarrow e'$. E.g b and e
- $L(b) > L(e)$ but $b \parallel e$
- How to solve this problem?

Vector Clock

- Lamport's clock: $L(e) < L(e')$ we cannot conclude that $e \rightarrow e'$.
- Vector clock to overcome the above problem.
- N processes is an array of N integers. Each process keeps its own vector clock V_i , which it uses to timestamp local events.

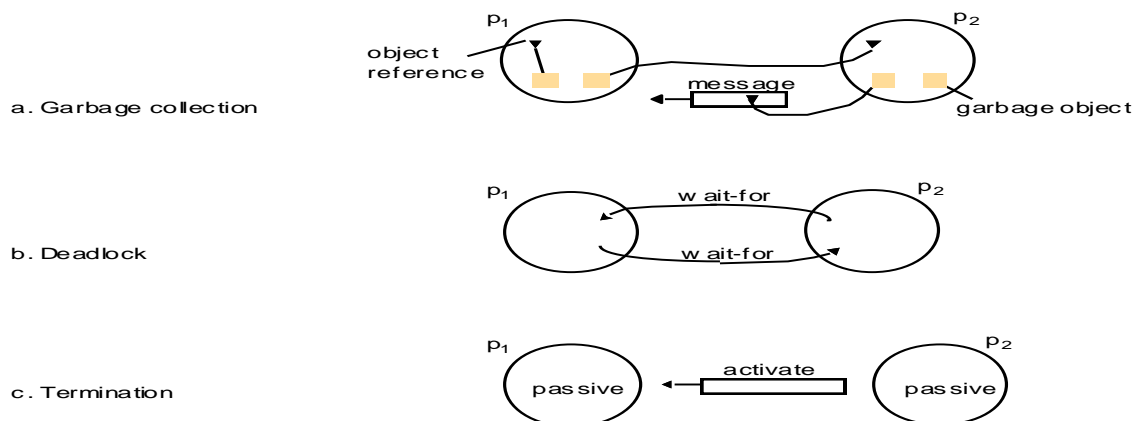
- VC1: initially, $V_i[j] = 0$, for $i, j = 1, 2, \dots, N$
- VC2: just before p_i timestamps an event, it sets $V_i[i] = v_i[i] + 1$
- VC3: p_i includes the value $t = V_i$ in every message it sends
- VC4: when p_i receives a timestamp t in a message, it sets $V_i[j] = \max(V_i[j], t[j])$ for $j = 1, 2, \dots, N$. Merge operation.



- To compare vector timestamps, we need to compare each bit. Concurrent events cannot find a relationship.
- Drawback compared with Lamport time, taking up an amount of storage and message payload proportional to N .

Global states

- We want to find out whether a particular property is true of a distributed system as it executes.
- We will see three examples:
 - Distributed garbage collection: if there are no longer any reference to objects anywhere in the distributed system, the memory taken up by the objects should be reclaimed.
 - Distributed deadlock detection: when each of a collection of processes waits for another process to send it a message, and where there is a cycle in the graph of this “wait-for” relationship.
 - Distributed termination detection: detect if a distributed algorithm has terminated. It seems that we only need to test whether each process has halted. However, it is not true. E.g. two processes and each of which may request values from the other. It can be either in passive or active state. Passive means it is not engaged in any activity but is prepared to respond. Two processes may both in passive states. At the same time, there is a message in on the way from P_2 to P_1 , after P_1 receives it, it will become active again. So the algorithm has not terminated.



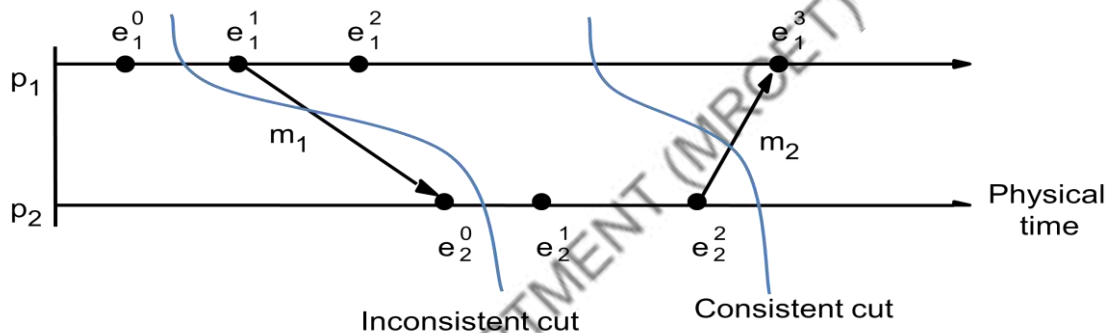
Global States and consistent cuts

- It is possible to observe the succession of states of an individual process, but the question of how to ascertain a global state of the system – the state of the collection of processes is much harder.
- The essential problem is the absence of global time. If we had perfectly synchronized clocks at which processes would record its state, we can assemble the global state of the system from local states of all processes at the same time.
- The question is: can we assemble the global state of the system from local states recorded at different real times?
- The answer is “YES”.

$$history(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \dots \rangle$$

$$finite\ prefix\ of\ history : h_i^k = \langle e_i^0, e_i^1, e_i^2, \dots, e_i^k \rangle$$

- A series of events occurs at each process. Each event is either an internal action of the process (variables updates) or it is the sending or receipt of a message over the channel.
- S_i^k is the state of process P_i before k th event occurs, so S_i^0 is the initial state of P_i .
- Thus the global state corresponds to initial prefixes of the individual process histories.



- A cut of the system's execution is a subset of its global history that is a union of prefixes of process histories $C = h_1^{c_1} \cup h_2^{c_2} \cup \dots \cup h_N^{c_N}$
- The state of each process is in the state after the last event occurs in its own cut. The set of last events from all processes are called frontier of the cut.
- **Inconsistent cut**: since P_2 contains receiving of m_1 , but at P_1 it does not include sending of that message. This cut shows the an effect without a cause. We will never reach a global state that corresponds to process state at the frontier by actual execution under this cut.
- **Consistent cut**: it includes both the sending and receipt of m_1 . It includes the sending but not the receipt of m_2 . It is still consistent with actual execution.
- A cut C is **consistent** if, for each event it contains, it also contains all the events that happened-before that event.

$$for\ all\ events\ e \in C, f \rightarrow e \Rightarrow f \in C$$

- A consistent global state is one that corresponds to a consistent cut.
- A run is a total ordering of all the events in a global history that is consistent with each local history's ordering.
- A linearization or consistent run is an ordering of the events in a global history that is consistent with this happened-before relation.

Global state predicate

- Global state predicate is a function that maps from the set of global states of processes in the system to true or false.
- Stable characteristics associated with object being garbage, deadlocked or terminated: once the system enters a state in which the predicate is True. It remains True in all future states reachable from that state.
- Safety (evaluates to deadlocked false for all states reachable from S_0)
- Liveness (evaluate to reaching termination true for some of the states reachable from S_0)

Chandy and Lamport's 'snapshot' algorithm

- Chandy and Lamport(1985) describe a "snapshot" algorithm for determining global states of distributed system.
- Record a set of process and channel states for a set of processes P_i such that even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent.
- The algorithm records state locally at processes without giving a method for gathering the global state.
- Assumptions:
 1. Neither channels nor processes fail; communication is reliable so that every message sent is eventually received intact, exactly once;
 2. Channels are unidirectional either incoming or outgoing and provide FIFO order message delivery;
 3. The graph of processes and channels is strongly connected (there is a path between any two processes).
 4. Any process may initiate a global snapshot at any time.
 5. The processes may continue their normal execution and send and receive normal messages while the snapshot takes place.
- Each process records its own state and also for each incoming channel a set of messages sent to it.
- Allow us to record process states at different times but to account for the differential between process states in terms of message transmitted but not yet received.
- If process p_i has sent a message m to process p_j , but p_j has not received it, then we account for m as belong to the state of the channel between them.
- Use of special marker message. It has a dual role, as a prompt for the receiver to save its own state if it has not done so; and as a means of determining which messages to include in the channel state.
- *****

On p_i 's receipt of a *marker* message over channel c :

if (p_i has not yet recorded its state) it

records its process state now;

records the state of c as the empty set;

turns on recording of messages arriving over other incoming channels;

else

p_i records the state of c as the set of messages it has received over c since it saved its state.

end if

Marker sending rule for process p_i

After p_i has recorded its state, for each outgoing channel c :

p_i sends one marker message over c

(before it sends any other message over c).

Coordination and Agreement: Introduction

- A set of processes coordinate their actions. How to agree on one or more values
 - Avoid fixed master-slave relationship to avoid single points of failure for fixed master.
- Distributed mutual exclusion for resource sharing
- A collection of process **share resources**, mutual exclusion is needed to prevent interference and ensure consistency. (critical section)
- No shared variables or facilities are provided by single local kernel to solve it. Require a solution that is based solely on message passing.
- Important factor to consider while designing algorithm is the failure

Distributed mutual exclusion

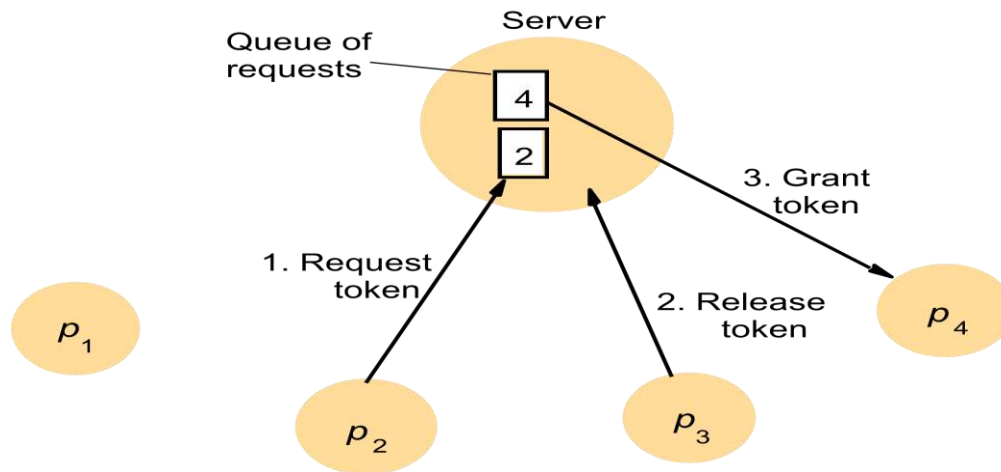
- Application level protocol for executing a critical section
 - enter() // enter critical section-block if necessary
 - resourceAccess() //access shared resources
 - exit() //leave critical section-other processes may enter
- Essential requirements:
 - ME1: (safety) at most one process may execute in the critical section
 - ME2: (liveness) Request to enter and exit the critical section eventually succeed.
 - ME3(ordering) One request to enter the CS happened-before another, then entry to the CS is granted in that order.
- ME2 implies freedom from both deadlock and starvation. Starvation involves fairness condition. The order in which processes enter the critical section. It is not possible to use the request time to order them due to lack of global clock. So usually, we use happen-before ordering to order message requests.

Performance Evaluation

- Bandwidth consumption, which is proportional to the number of messages sent in each entry and exit operations.
- The client delay incurred by a process at each entry and exit operation.
- throughput of the system. Rate at which the collection of processes as a whole can access the critical section. Measure the effect using the synchronization delay between one process exiting the critical section and the next process entering it; the shorter the delay is, the greater the throughput is.

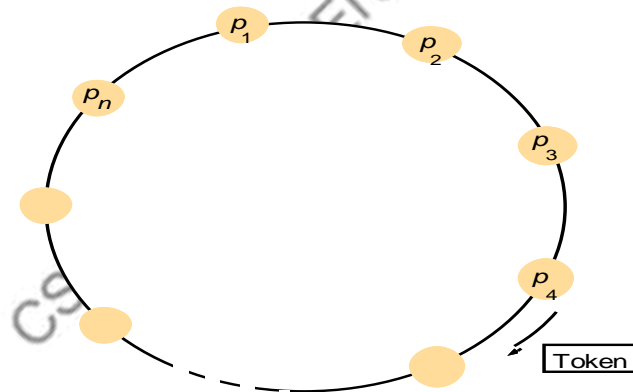
Central Server Algorithm

- The simplest way to grant permission to enter the critical section is to employ a server.
- A process sends a request message to server and awaits a reply from it.
- If a reply constitutes a token signifying the permission to enter the critical section.
- If no other process has the token at the time of the request, then the server replied immediately with the token.
- If token is currently held by other processes, the server does not reply but queues the request.
- Client on exiting the critical section, a message is sent to server, giving it back the token.
 - ME1: safety
 - ME2: liveness Are satisfied but not
 - ME3: ordering



Ring-based Algorithm

- Simplest way to arrange mutual exclusion between N processes without requiring an additional process is arrange them in a logical ring.
- Each process p_i has a communication channel to the next process in the ring, $p_{(i+1)/\text{mod } N}$.
- The unique token is in the form of a message passed from process to process in a single direction clockwise.
- If a process does not require to enter the CS when it receives the token, then it immediately forwards the token to its neighbor.
- A process requires the token waits until it receives it, but retains it.
- To exit the critical section, the process sends the token on to its neighbor.



ME1: safety ME2: liveness Are satisfied but not ME3: ordering

Using Multicast and logical clocks

- Mutual exclusion between N peer processes based upon multicast.
- Processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have replied to this message.
- The condition under which a process replies to a request are designed to ensure ME1 ME2 and ME3 are met.
- Each process p_i keeps a Lamport clock. Message requesting entry are of the form $\langle T, p_i \rangle$.
- Each process records its state of either RELEASE, WANTED or HELD in a variable state.
 - If a process requests entry and all other processes is RELEASED, then all processes reply immediately.
 - If some process is in state HELD, then that process will not reply until it is finished.
 - If some process is in state WANTED and has a smaller timestamp than the incoming request, it will queue the request until it is finished.

- If two or more processes request entry at the same time, then whichever bears the lowest timestamp will be the first to collect N-1 replies.

Ricart and Agrawala's algorithm

On initialization

state := RELEASED;

To enter the section

state := WANTED;

Multicast request to all processes;

T := request's timestamp;

Wait until (number of replies received = (N - 1));

state := HELD;

request processing deferred

On receipt of a request $\langle T_i, p_i \rangle$ at p_j ($i \neq j$)

if (state = HELD or (state = WANTED and $(T, p_j) < (T_i, p_i)$))

then

queue request from p_i without replying;

else

reply immediately to p_i ;

end if

To exit the critical section

state := RELEASED;

reply to any queued requests;

Maekawa's voting algorithm

- It is not necessary for all of its peers to grant access. Only need to obtain permission to enter from subsets of their peers, as long as the subsets used by any two processes overlap.
- Think of processes as voting for one another to enter the CS. A candidate process must collect sufficient votes to enter.
- Processes in the intersection of two sets of voters ensure the safety property ME1 by casting their votes for only one candidate
- A voting set V_i associated with each process p_i .
- there is at least one common member of any two voting sets, the size of all voting set are the same size to be fair.
- The optimal solution to minimizes K is $K \sim \sqrt{N}$ and $M=K$.

$$V_i \subseteq \{p_1, p_2, \dots, p_N\}$$

such that for all $i, j = 1, 2, \dots, N$:

$$p_i \in V_i$$

$$V_i \cap V_j \neq \emptyset$$

$$|V_i| = K$$

Each process is contained in M of the voting set V_i

Maekawa's algorithm

```
On initialization
state := RELEASED;
voted := FALSE;
For  $p_i$  to enter the critical section
state := WANTED;
Multicast request to all processes in  $V_i$ ;
Wait until (number of replies received =  $K$ );
state := HELD;
On receipt of a request from  $p_j$  at  $p_i$ 
if (state = HELD or voted = TRUE)
then
    queue request from  $p_j$  without replying;
else
    send reply to  $p_j$ ;
    voted := TRUE;
end if

For  $p_i$  to exit the critical section
state := RELEASED;
Multicast release to all processes in  $V_i$ ;
On receipt of a release from  $p_j$  at  $p_i$ 
if (queue of requests is non-empty)
then
    remove head of queue – from  $p_h$ ;
    say,
    send reply to  $p_h$ ;
    voted := TRUE;
else
    voted := FALSE;
end if
```

Elections

✂ Algorithm to choose a unique process to play a particular role is called an election algorithm. E.g. central server for mutual exclusion, one process will be elected as the server. Everybody must agree. If the server wishes to retire, then another election is required to choose a replacement.

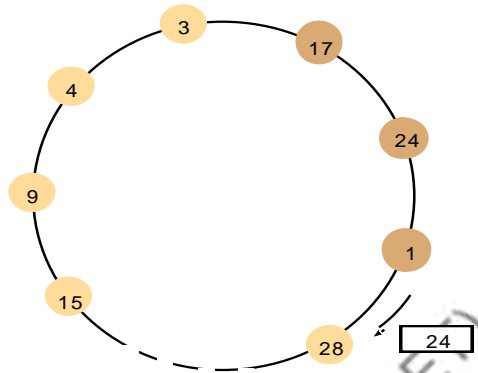
✂ Requirements:

- ☐ E1(safety): a participant p_i has $electd_i = \perp$ or $electd_i = P$
Where P is chosen as the non-crashed process at the end of run with the largest identifier. (concurrent elections possible.)
- ☐ E2(liveness): All processes P_i participate in election process and eventually set $electd_i \neq \perp$ or crash

A ring based election algorithm

- All processes arranged in a logical ring.
- Each process has a communication channel to the next process.
- All messages are sent clockwise around the ring.
- Assume that no failures occur, and system is asynchronous.
- Goal is to elect a single process coordinator which has the largest identifier.
 - **Initially**, every process is marked as non-participant. Any process can begin an election.
 - The **starting** process marks itself as participant and place its identifier in a message to its neighbour.
 - A process receives a message and **compare** it with its own. If the arrived identifier is **larger**, it passes on the message.
 - If arrived identifier is **smaller** and receiver is not a participant, substitute its own identifier in the message and forward it. It does not forward the message if it is already a participant.

- On forwarding of any case, the process marks itself as a participant.
- If the received identifier is that of the receiver itself, then this process' s identifier must be the greatest, and it becomes the **coordinator**.
- The coordinator marks itself as non-participant set elected_i and sends an **elected** message to its neighbour enclosing its ID.
- When a process receives elected message, marks itself as a non-participant, sets its variable elected_i and forwards the message.



- E1 is met. All identifiers are compared, since a process must receive its own ID back before sending an elected message.
- E2 is also met due to the guaranteed traversals of the ring.
- Tolerate no failure makes ring algorithm of limited practical use.
- If only a single process starts an election, the worst-performance case is then the anti-clockwise neighbour has the highest identifier. A total of N-1 messages is used to reach this neighbour. Then further N messages are required to announce its election. The elected message is sent N times. Making 3N-1 messages in all.
- Turnaround time is also 3N-1 sequential message transmission time

The bully algorithm

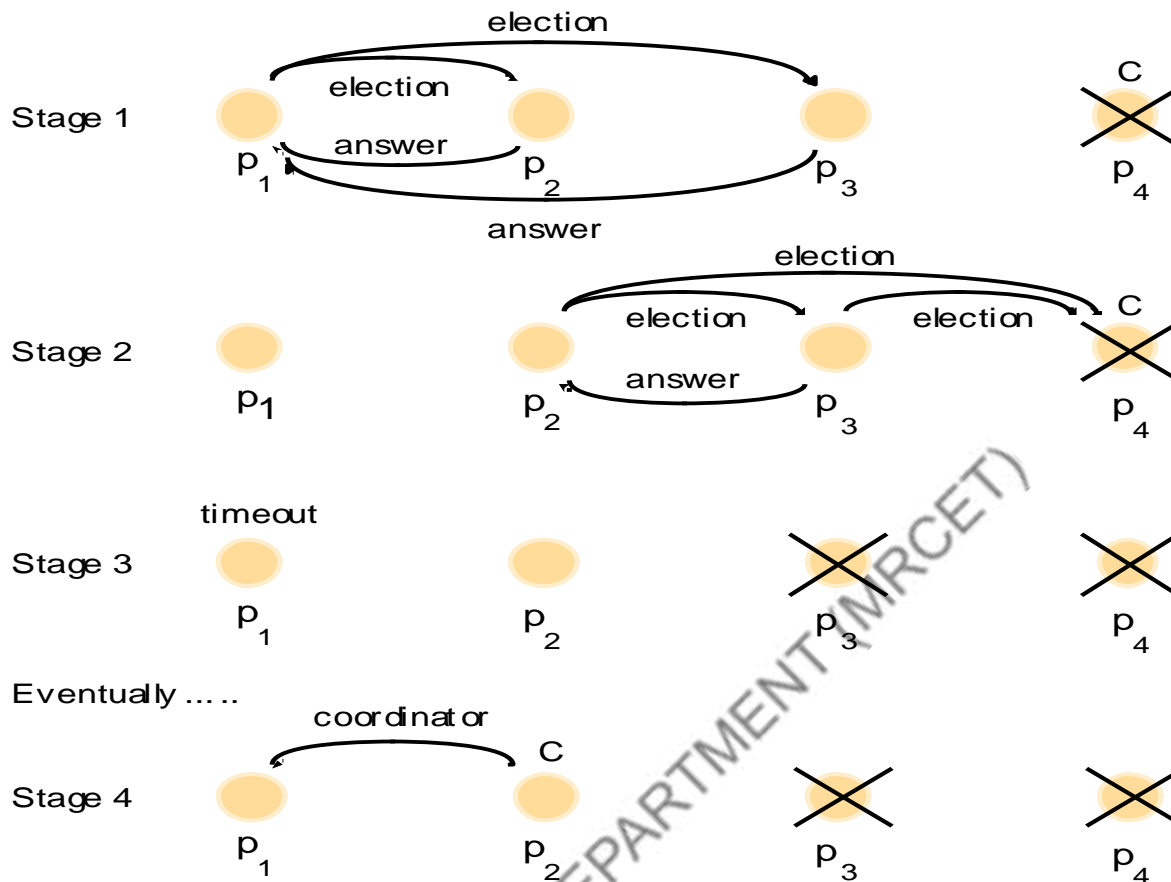
- Allows process to crash during an election, although it assumes the message delivery between processes is reliable.
- Assume system is synchronous to use timeouts to detect a process failure.
- Assume each process knows which processes have higher identifiers and that it can communicate with all such processes.
- Three types of messages:
 - Election is sent to announce an election message. A process begins an election when it notices, through timeouts, that the coordinator has failed. $T = 2T_{trans} + T_{process}$ From the time of sending
 - Answer is sent in response to an election message.
 - Coordinator is sent to announce the identity of the elected process.

1. The process begins a election by sending an election message to these processes that have a higher ID and awaits an answer in response. If none arrives within time T, the process considers itself the coordinator and sends coordinator message to all processes with lower identifiers. Otherwise, it waits a further time T' for coordinator message to arrive. If none, begins another election.
2. If a process receives a coordinator message, it sets its variable elected_i to be the coordinator ID.

3. If a process receives an election message, it send back an answer message and begins another election unless it has begun one already.

E1 may be broken if timeout is not accurate or replacement. (suppose P3 crashes and replaced by another process. P2 set P3 as coordinator and P1 set P2 as coordinator)

E2 is clearly met by the assumption of reliable transmission.

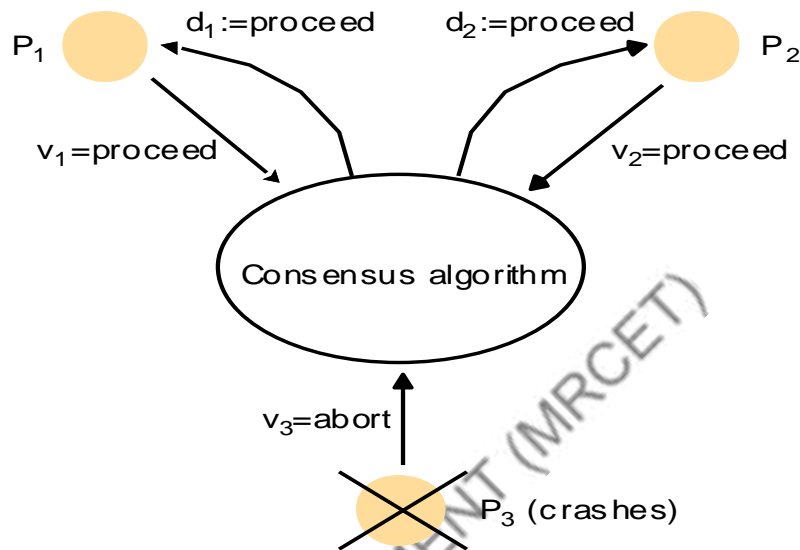


- Best case the process with the second highest ID notices the coordinator's failure. Then it can immediately elect itself and send $N-2$ coordinator messages.
- The bully algorithm requires $O(N^2)$ messages in the worst case - that is, when the process with the least ID first detects the coordinator's failure. For then $N-1$ processes altogether begin election, each sending messages to processes with higher ID.

Consensus and Related Problems (agreement)

- The problem is for processes to agree on a value after one or more of the processes has proposed what that value should be. (e.g. all controlling computers should agree upon whether let the spaceship proceed or abort after one computer proposes an action.)
- Assumptions: Communication is reliable but the processes may fail (arbitrary process failure as well as crash). Also specify that up to some number f of the N processes are faulty.
- Every process p_i begins in the undecided state and propose a single value v_i , drawn from a set D . The processes communicate with one another, exchanging values. Each process then sets the value of a decision variable d_i . In doing so it enters the decided state, in which it may no longer change d_i .
- Requirements:
 - Termination: Eventually each correct process sets its decision variable.

- Agreement: The decision value of all correct processes is the same: if p_i and p_j are correct and have entered the decided state, then $d_i = d_j$
- Integrity: if the correct processes all proposed the same value, then any correct process in the decided state has chosen that value. This condition can be loosened. For example, not necessarily all of them, may be some of them.
- It will be straightforward to solve this problem if no process can fail by using multicast and majority vote.
- Termination guaranteed by reliability of multicast, agreement and integrity guaranteed by the majority definition and each process has the same set of proposed value to evaluate. .



Byzantine general problem (proposed in 1982)

- Three or more generals are to agree to attack or to retreat. One, the commander, issues the order. The others, lieutenants to the commander, are to decide to attack or retreat.
- But one or more of the general may be treacherous-that is, faulty. If the commander is treacherous, he proposes attacking to one general and retreating to another. If a lieutenant is treacherous, he tells one of his peers that the commander told him to attack and another that they are to retreat.
- A. Byzantine general problem is different from consensus in that a distinguished process supplies a value that the others are to agree upon, instead of each of them proposing a value.
- Requirements:
 - Termination: eventually each correct process sets its decision variable.
 - Agreement: the decision value of all correct processes is the same.
 - Integrity: If the commander is correct, then all correct processes decide on the value that the commander proposed.
- If the commander is correct, the integrity implies agreement; but the commander need not be correct.
- B. Interactive consistency problem : Another variant of consensus, in which every process proposes a single value. Goal of this algorithm is for the correct processes to agree on a decision vector of values, one for each process.
- Requirements: Termination: eventually each correct process sets its decision variable. Agreement: the decision vector of all correct processes is the same. Integrity: If p_i is correct, then all correct processes decide on v_i as the i th component of the vector.

Consensus in a synchronous system

- Basic multicast protocol assuming up to f of the N processes exhibit crash failures.
- Each correct process collects proposed values from the other processes. This algorithm proceeds in $f+1$ rounds, in each of which the correct processes Basic-multicast the values between themselves. At most f processes may crash, by assumption. At worst, all f crashes during the round, but the algorithm guarantees that at the end of the rounds all the correct processes that have survived have the same final set of values are in a position to agree.

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization

$Values_i^1 := \{v_i\}; Values_i^0 = \{\};$

In round r ($1 \leq r \leq f + 1$)

$B\text{-multicast}(g, Values_i^r - Values_i^{r-1}); //$ Send only values that have not been sent

$Values_i^{r+1} := Values_i^r;$

while (in round r)

{

On B-deliver(V_j) from some p_j

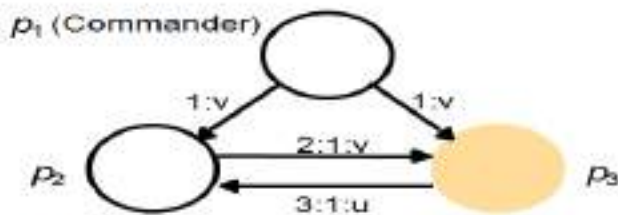
$Values_i^{r+1} := Values_i^{r+1} \cup V_j;$

}

After $(f + 1)$ rounds

Assign $d_i = \text{minimum}(Values_i^{f+1});$

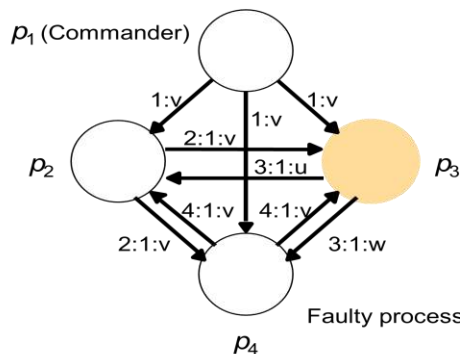
Three byzantine generals



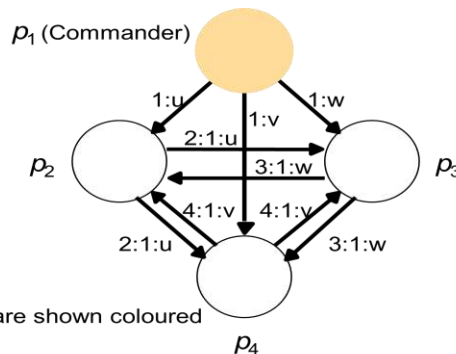
Faulty processes are shown coloured

3:1:u: first number indicates source, the second number indicates Who says. From P_3 , P_1 says u.

If solution exists, P_2 bound to decide on v when commander is correct. If no solution can distinguish between correct and faulty commander, p_2 must also choose the value sent by commander. By Symmetry, P_3 should also choose commander, p_2 does the same thing. But it contradicts with agreement. No solution is $N \leq 3f$. All because that a correct general can not tell which process is faulty. Digital signature can solve this problem.



Faulty processes are shown coloured



Multicast Communication

- Group (multicast) communication: for each of a group of processes to receive copies of the messages sent to the group, often with delivery guarantees
 - The set of messages that every process of the group should receive
 - On the delivery ordering across the group members
- Challenges
 - Efficiency concerns include minimizing overhead activities and increasing throughput and bandwidth utilization
 - Delivery guarantees ensure that operations are completed
- Types of group
 - Static or dynamic: whether joining or leaving is considered
 - Closed or open
 - A group is said to be closed if only members of the group can multicast to it. A process in a closed group sends to itself any messages to the group
 - A group is open if processes outside the group can send to it
- Simple basic multicasting (**B-multicast**) is sending a message to every process that is a member of a defined group
 - $B\text{-multicast}(g, m)$ for each process $p \in \text{group } g$, $\text{send}(p, \text{message } m)$
 - On $\text{receive}(m)$ at p : $B\text{-deliver}(m)$ at p
- **Reliable multicasting (R-multicast)** requires these properties
 - Integrity: a correct process sends a message to only a member of the group and does it only once
 - Validity: if a correct process sends a message, it will eventually be delivered.
 - Agreement: if a message is delivered to a correct process, all other correct processes in the group will deliver it

On initialization

$Received := \{\};$

For process p to R-multicast message m to group g

$B\text{-multicast}(g, m); \quad // p \in g \text{ is included as a destination}$

On $B\text{-deliver}(m)$ at process q with $g = \text{group}(m)$

if $(m \notin Received)$

then

$Received := Received \cup \{m\};$

if $(q \neq p)$ then $B\text{-multicast}(g, m);$ end if

$R\text{-deliver } m;$

end if

- Implementing reliable R-multicast over B-multicast
 - When a message is delivered, the receiving process multicasts it
 - Duplicate messages are identified (possible by a sequence number) and not delivered

Types of message ordering

Three types of message ordering

- **FIFO (First-in, first-out) ordering**: if a correct process delivers a message before another, every correct process will deliver the first message before the other
- **Casual ordering**: any correct process that delivers the second message will deliver the previous message first
- **Total ordering**: if a correct process delivers a message before another, any other correct process that delivers the second message will deliver the first message first

•Note that

- FIFO ordering and casual ordering are only partial orders
- Not all messages are sent by the same sending process
- Some multicasts are concurrent, not able to be ordered by happened- before
- Total order demands consistency, but not a particular order
-

•Totally ordered messages

T_1 and T_2 .

•FIFO-related messages F_1 and F_2 .

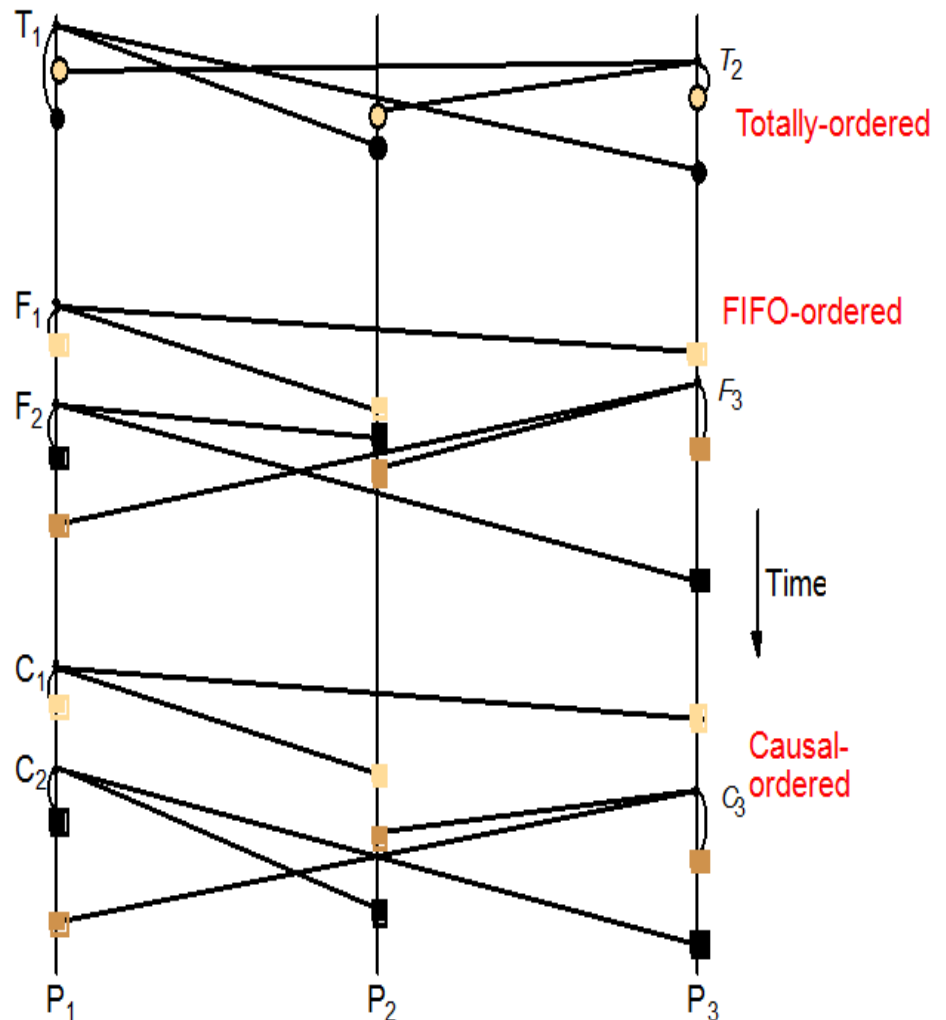
•Causally-related messages C_1 and C_3

• Causal ordering implies FIFO ordering

• Total ordering does not imply causal ordering.

• Causal ordering does not imply total ordering.

• Hybrid mode: causal-total ordering, FIFO-total ordering.



Multicast Communication

- Group (multicast) communication: for each of a group of processes to receive copies of the messages sent to the group, often with delivery guarantees
 - The set of messages that every process of the group should receive
 - On the delivery ordering across the group members
- Challenges
 - Efficiency concerns include minimizing overhead activities and increasing throughput and bandwidth utilization
 - Delivery guarantees ensure that operations are completed
- Types of group
 - Static or dynamic: whether joining or leaving is considered
 - Closed or open
 - A group is said to be closed if only members of the group can multicast to it. A process in a closed group sends to itself any messages to the group
 - A group is open if processes outside the group can send to it
- Simple basic multicasting (**B-multicast**) is sending a message to every process that is a member of a defined group
 - $B\text{-multicast}(g, m)$ for each process $p \in \text{group } g$, $\text{send}(p, \text{message } m)$
 - On $\text{receive}(m)$ at p : $B\text{-deliver}(m)$ at p
- **Reliable multicasting (R-multicast)** requires these properties
 - Integrity: a correct process sends a message to only a member of the group and does it only once
 - Validity: if a correct process sends a message, it will eventually be delivered.
 - Agreement: if a message is delivered to a correct process, all other correct processes in the group will deliver it

On initialization

$Received := \{\};$

For process p to R-multicast message m to group g

$B\text{-multicast}(g, m); \quad // p \in g \text{ is included as a destination}$

On $B\text{-deliver}(m)$ at process q with $g = \text{group}(m)$

if $(m \notin Received)$

then

$Received := Received \cup \{m\};$

if $(q \neq p)$ then $B\text{-multicast}(g, m);$ end if

$R\text{-deliver } m;$

end if

- Implementing reliable R-multicast over B-multicast
 - When a message is delivered, the receiving process multicasts it
 - Duplicate messages are identified (possible by a sequence number) and not delivered

Types of message ordering

Three types of message ordering

- **FIFO (First-in, first-out) ordering:** if a correct process delivers a message before another, every correct process will deliver the first message before the other
- **Casual ordering:** any correct process that delivers the second message will deliver the previous message first
- **Total ordering:** if a correct process delivers a message before another, any other correct process that delivers the second message will deliver the first message first

•Note that

- FIFO ordering and casual ordering are only partial orders
- Not all messages are sent by the same sending process
- Some multicasts are concurrent, not able to be ordered by happened- before
- Total order demands consistency, but not a particular order
-

•Totally ordered messages

T_1 and T_2 .

•FIFO-related messages F_1 and F_2 .

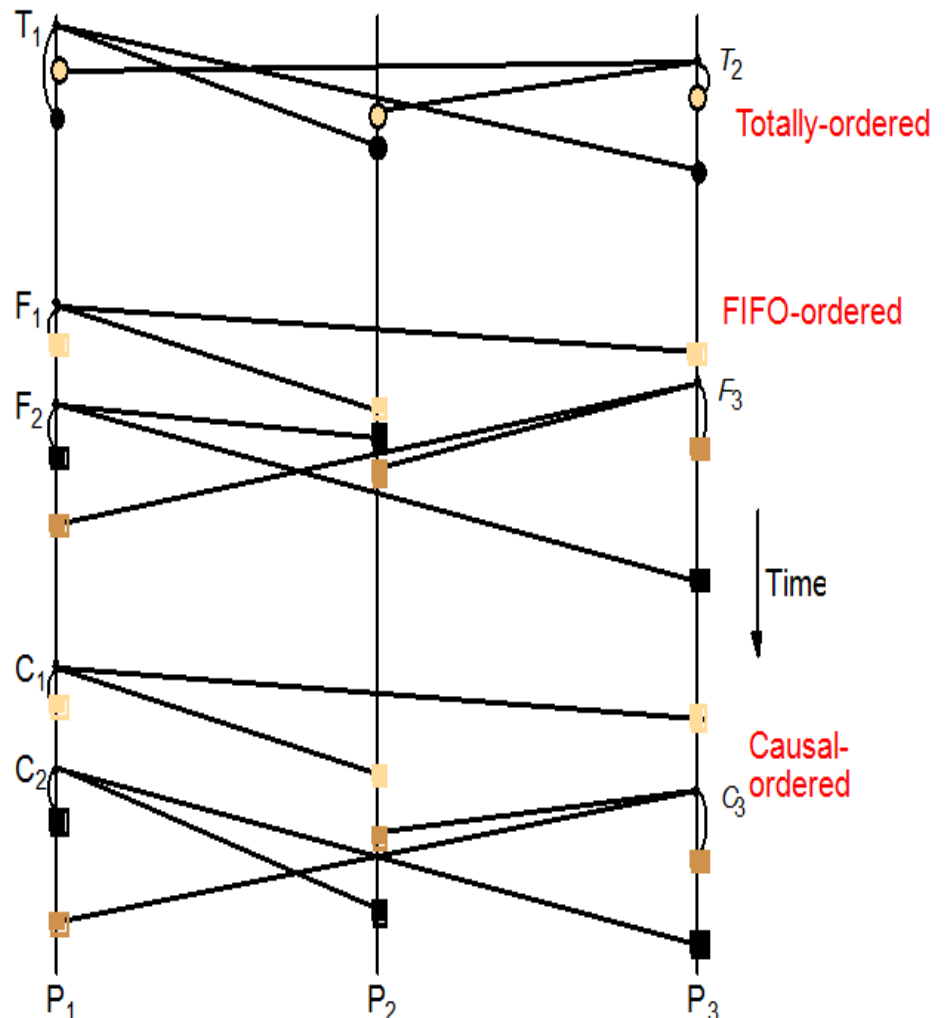
•Causally-related messages C_1 and C_3

• Causal ordering implies FIFO ordering

• Total ordering does not imply causal ordering.

• Causal ordering does not imply total ordering.

• Hybrid mode: causal-total ordering, FIFO-total ordering.



UNIT III

Interprocess Communication: Introduction, Characteristics of Interprocess communication, External Data Representation and Marshalling, Client-Server Communication, Group Communication, Case Study: IPC in UNIX.

Distributed Objects and Remote Invocation: Introduction, Communication between Distributed Objects, Remote Procedure Call, Events and Notifications, Case study: Java RMI.

INTRODUCTION:

The java API for inter process communication in the internet provides both datagram and stream communication.

The two communication patterns that are most commonly used in distributed programs.

Client-Server communication

The request and reply messages provide the basis for remote method invocation (RMI) or remote procedure call (RPC).

Group communication

The same message is sent to several processes.

This chapter is concerned with middleware.

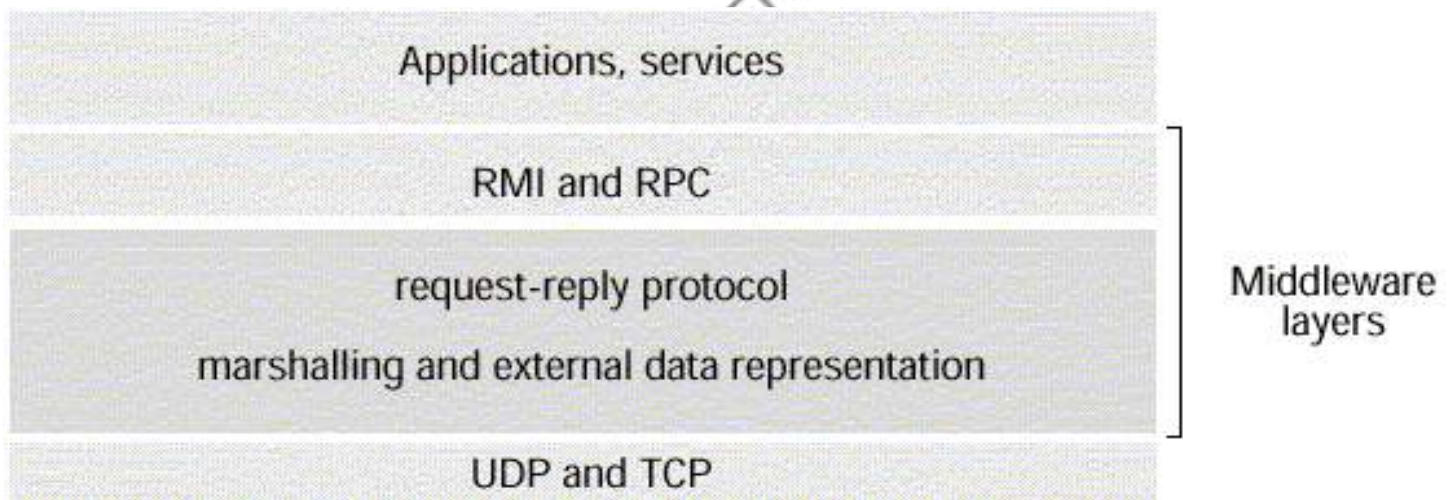


Figure 1. Middleware layers

Remote Method Invocation (RMI)

It allows an object to invoke a method in an object in a remote process.

E.g. CORBA and Java RMI

Remote Procedure Call (RPC)

It allows a client to call a procedure in a remote server.

The CHARACTERISTICS of INTERPROCESS COMMUNICATION

Synchronous and asynchronous communication

- In the synchronous form, both send and receive are blocking operations.
- In the asynchronous form, the use of the send operation is non-blocking and the receive operation can have blocking and non-blocking variants.

Message destinations

- A local port is a message destination within a computer, specified as an integer.
- A port has an exactly one receiver but can have many senders.

Reliability

- A reliable communication is defined in terms of validity and integrity.
- A point-to-point message service is described as reliable if messages are guaranteed to be delivered despite a reasonable number of packets being dropped or lost.
- For integrity, messages must arrive uncorrupted and without duplication.

Ordering

- Some applications require that messages be delivered in sender order.

External Data Representation

- The information stored in running programs is represented as data structures, whereas the information in messages consists of sequences of bytes.
- Irrespective of the form of communication used, the data structure must be converted to a sequence of bytes before transmission and rebuilt on arrival.
- External Data Representation is an agreed standard for the representation of data structures and primitive values.

Data representation problems are:

- Using agreed external representation, two conversions necessary
- Using sender's or receiver's format and convert at the other end .

Marshalling

- Marshalling is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.

Unmarshalling

- Unmarshalling is the process of disassembling a collection of data on arrival to produce an equivalent collection of data items at the destination.
-

Three approaches to external data representation and marshalling are:

- CORBA
- Java's object serialization
- XML

1. CORBA Common Data Representation (CDR)

CORBA CDR is the external data representation defined with CORBA 2.0.

It consists 15 primitive types:

- Short (16 bit)
- Long (32 bit)
- Unsigned short
- Unsigned long
- Float(32 bit)
- Double(64 bit)
- Char
- Boolean(TRUE,FALSE)
- Octet(8 bit)
- Any(can represent any basic or constructed type)
- Composite type are shown in Figure 8.

Type	Representation
<i>sequence</i>	length (unsigned long) followed by elements in order
<i>string</i>	length (unsigned long) followed by characters in order (can also can have wide characters)
<i>array</i>	array elements in order (no length specified because it is fixed)
<i>struct</i>	in the order of declaration of the components
<i>enumerated</i>	unsigned long (the values are specified by the order declared)
<i>union</i>	type tag followed by the selected member

Figure 8. CORBA CDR for constructed types

Constructed types: The primitive values that comprise each constructed type are added to a sequence of bytes in a particular order, as shown in Figure 8.

Figure 9 shows a message in CORBA CDR that contains the three fields of a struct whose respective types are string, string, and unsigned long.

example: struct with value {'Smith', 'London', 1934}

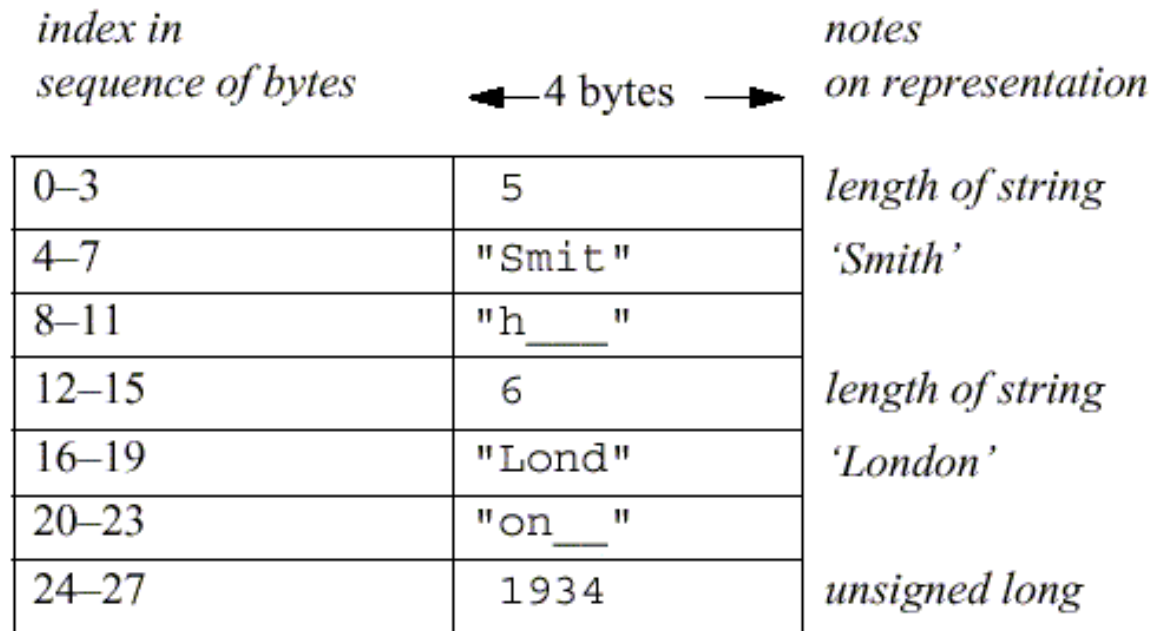


Figure 9. CORBA CDR message

2. Java object serialization

In Java RMI, both object and primitive data values may be passed as arguments and results of method invocation.

An object is an instance of a Java class.

Example the Java class equivalent to the Person struct

```
Public class Person implements Serializable {
```

```
    Private String name;
```

```
    Private String place;
```

```
    Private int year;
```

```
    Public Person(String aName ,String aPlace, int aYear) {
```

```
        name = aName;
```

```
        place = aPlace;
```

```
        year = aYear;
```

```
    }
```

```
//followed by methods for accessing the instance variables
```

```
}
```

The serialized form is illustrated in Figure 10.

<i>Serialized values</i>				<i>Explanation</i>
Person	8-byte version number	h0		class name, version number
3	int year	java.lang.String name	java.lang.String place	number, type and name of instance variables
1934	5 Smith	6 London	h1	values of instance variables

Figure 10. Indication of Java serialization form

3. Remote Object References

- Remote object references are needed when a client invokes an object that is located on a remote server.
- A remote object reference is passed in the invocation message to specify which object is to be invoked.
- Remote object references must be unique over space and time.
- In general, may be many processes hosting remote objects, so remote object referencing must be unique among all of the processes in the various computers in a distributed system.

generic format for remote object references is shown in Figure 11.

<i>32 bits</i>	<i>32 bits</i>	<i>32 bits</i>	<i>32 bits</i>	
Internet address	port number	time	object number	interface of remote object

Figure 11. Representation of a remote object references

- internet address/port number: process which created object
- time: creation time
- object number: local counter, incremented each time an object is created in the creating process
- interface: how to access the remote object (if object reference is passed from one client to another).

Client-Server Communication

- The client-server communication is designed to support the roles and message exchanges in typical client-server interactions.
- In the normal case, request-reply communication is synchronous because the client process blocks until the reply arrives from the server.
- Asynchronous request-reply communication is an alternative that is useful where clients can afford to retrieve replies later.
- Often built over UDP datagrams
- Client-server protocol consists of request/response pairs, hence no acknowledgements at transport layer are necessary
- Avoidance of connection establishment overhead
- No need for flow control due to small amounts of data are transferred
- The request-reply protocol was based on a trio of communication primitives: `doOperation`, `getRequest`, and `sendReply` shown in Figure 12.

The request-reply protocol is shown in Figure 12.

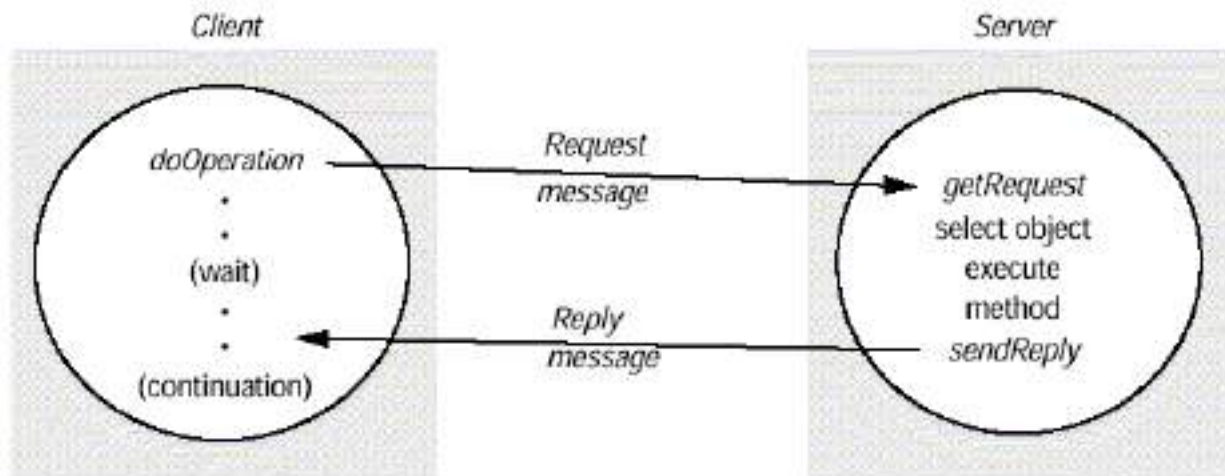


Figure 12. Request-reply communication

- The designed request-reply protocol matches requests to replies.
- If UDP datagrams are used, the delivery guarantees must be provided by the request-reply protocol, which may use the server reply message as an acknowledgement of the client request message.
- Figure 13 outlines the three communication primitives.

```
public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)
```

sends a request message to the remote object and returns the reply.

The arguments specify the remote object, the method to be invoked and the arguments of that method.

```
public byte[] getRequest ();
```

acquires a client request via the server port.

```
public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
```

sends the reply message *reply* to the client at its Internet address and port.

The

information to be transmitted in a request message or a reply message is shown in Figure 14.

<i>messageType</i>	<i>int (0=Request, 1= Reply)</i>
<i>requestId</i>	<i>int</i>
<i>objectReference</i>	<i>RemoteObjectRef</i>
<i>methodId</i>	<i>int or Method</i>
<i>arguments</i>	<i>// array of bytes</i>

Figure

14. Request-reply message structure

In a protocol message

- The first field indicates whether the message is a request or a reply message.
- The second field request id contains a message identifier.
- The third field is a remote object reference .
- The forth field is an identifier for the method to be invoked.

Message identifier

- A message identifier consists of two parts:
 - A requestId, which is taken from an increasing sequence of integers by the sending process
 - An identifier for the sender process, for example its port and Internet address.

Failure model of the request-reply protocol

- If the three primitive *doOperation*, *getRequest*, and *sendReply* are implemented over UDP datagram, they have the same communication failures.
 - Omission failure

- Messages are not guaranteed to be delivered in sender order.

RPC exchange protocols

- Three protocols are used for implementing various types of RPC.
 - The request (R) protocol.
 - The request-reply (RR) protocol.

The request-reply-acknowledge (RRA) protocol.

Name	Messages sent by		
	Client	Server	Client
R	Request		
RR	Request	Reply	
RRA	Request	Reply	Acknowledge reply

Figure15.RPC exchange protocols

- In the R protocol, a single request message is sent by the client to the server.
- The R protocol may be used when there is no value to be returned from the remote method.
- The RR protocol is useful for most client-server exchanges because it is based on request-reply protocol.
- RRA protocol is based on the exchange of three messages: request-reply-acknowledge reply.
- HTTP: an example of a request-reply protocol
 - HTTP is a request-reply protocol for the exchange of network resources between web clients and web servers.
 - HTTP protocol steps are:
 - Connection establishment between client and server at the default server port or at a port specified in the URL
 - client sends a request
 - server sends a reply
 - connection closure
- HTTP methods
 - GET

- Requests the resource, identified by URL as argument.
- If the URL refers to data, then the web server replies by returning the data
- If the URL refers to a program, then the web server runs the program and returns the output to the client.

<i>method</i>	<i>URL</i>	<i>HTTP version</i>	<i>headers</i>	<i>message body</i>
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

➤ HEAD

- ❖ This method is similar to GET, but only meta data on resource is returned (like date of last modification, type, and size)

➤ POST

- ❖ Specifies the URL of a resource (for instance, a server program) that can deal with the data supplied with the request.
- ❖ This method is designed to deal with:
 - Providing a block of data to a data-handling process
 - Posting a message to a bulletin board, mailing list or news group.
 - Extending a dataset with an append operation

➤ PUT

- ❖ Supplied data to be stored in the given URL as its identifier.

➤ DELETE

- ❖ The server deletes an identified resource by the given URL on the server.

➤ OPTIONS

- ❖ A server supplies the client with a list of methods.
- ❖ It allows to be applied to the given URL

➤ TRACE

- ❖ The server sends back the request message

➤ A reply message specifies

- ❖ The protocol version
- ❖ A status code
- ❖ Reason
- ❖ Some headers

- ❖ An optional message body

<i>HTTP version</i>	<i>status code</i>	<i>reason</i>	<i>headers</i>	<i>message body</i>
HTTP/1.1	200	OK		resource data

Figure 17. HTTP reply message

Group Communication

- The pairwise exchange of messages is not the best model for communication from one process to a group of other processes.
- A multicast operation is more appropriate.
- Multicast operation is an operation that sends a single message from one process to each of the members of a group of processes.
- The simplest way of multicasting, provides no guarantees about message delivery or ordering.
- Multicasting has the following characteristics:
 - Fault tolerance based on replicated services
 - A replicated service consists of a group of servers.
 - Client requests are multicast to all the members of the group, each of which performs an identical operation.
 - Finding the discovery servers in spontaneous networking
- Multicast messages can be used by servers and clients to locate available discovery services in order to register their interfaces or to look up the interfaces of other services in the distributed system.
 - Better performance through replicated data
 - Data are replicated to increase the performance of a service.
 - Propagation of event notifications
 - Multicast to a group may be used to notify processes when something happens.

IP multicast

- IP multicast is built on top of the Internet protocol, IP.
- IP multicast allows the sender to transmit a single IP packet to a multicast group.
- A multicast group is specified by class D IP address for which first 4 bits are 1110 in IPv4.
- The membership of a multicast group is dynamic.
- A computer belongs to a multicast group if one or more processes have sockets that belong to the multicast group.

The following details are specific to IPv4:

Multicast IP routers

IP packets can be multicast both on local network and on the wider Internet.

Local multicast uses local network such as Ethernet.

To limit the distance of propagation of a multicast datagram, the sender can specify the number of routers it is allowed to pass- called the time to live, or TTL for short.

Multicast address allocation

Multicast addressing may be permanent or temporary.

Permanent groups exist even when there are no members.

Multicast addressing by temporary groups must be created before use and cease to exist when all members have left.

The session directory (sd) program can be used to start or join a multicast session.

session directory provides a tool with an interactive interface that allows users to browse advertised multicast sessions and to advertise their own session, specifying the time and duration.

Java API to IP multicast

The Java API provides a datagram interface to IP multicast through the class MulticastSocket, which is a subset of DatagramSocket with the additional capability of being able to join multicast groups.

The class MulticastSocket provides two alternative constructors, allowing socket to be created to use either a specified local port, or any free local port.

A process can join a multicast group with a given multicast address by invoking the joinGroup method of its multicast socket.

Case Study: IPC in UNIX.

A process can be of two type:

Independent process.

Co-operating process.

An independent process is not affected by the execution of other processes while a co-operating process can be affected by other executing processes. Though one can think that those processes, which are running independently, will execute very efficiently but in practical, there are many situations when co-operative nature can be utilised for increasing computational speed, convenience and modularity. Inter process communication (IPC) is a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them. Processes can communicate with each other using these two ways:

Shared Memory

The Figure 1 below shows a basic structure of communication between processes via shared memory method and via message passing.

An operating system can implement both method of communication. First, we will discuss the shared memory method of communication and then message passing. Communication between processes using shared memory requires processes to share some variable and it completely depends on how programmer will implement it. One way of communication using shared memory can be imagined like this: Suppose process1 and process2 are executing simultaneously and they share some resources or use some information from other process, process1 generate information about certain computations or resources being used and keeps it as a record in shared memory. When process2 need to use the shared information, it will check in the record stored in shared memory and take note of the information generated by process1 and act accordingly. Processes can use shared memory for extracting information as a record from other process as well as for delivering any specific information to other process.

Let's discuss an example of communication between processes using shared memory method.

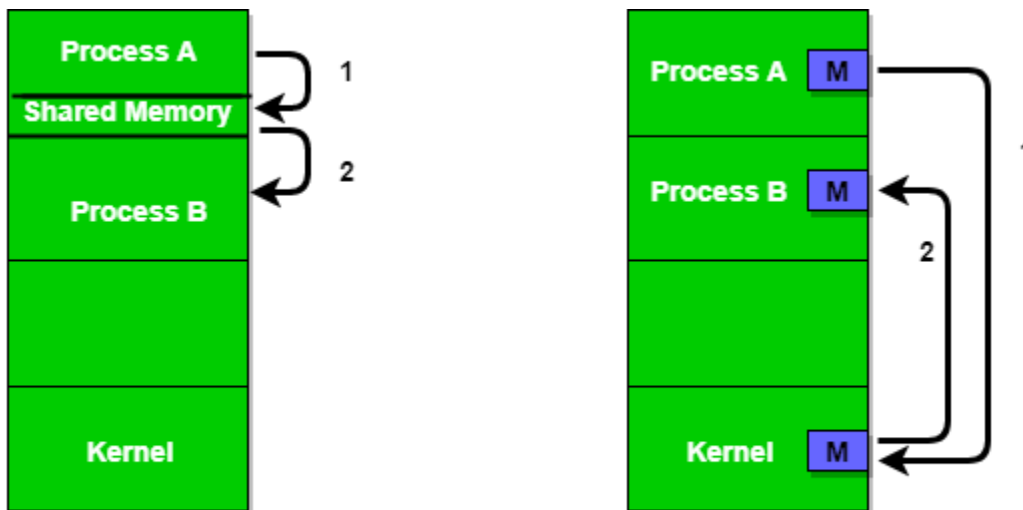


Figure 1 - Shared Memory and Message Passing

i) Shared Memory Method

Ex: Producer-Consumer problem

There are two processes: Producer and Consumer. Producer produces some item and Consumer consumes that item. The two processes share a common space or memory location known as buffer where the item produced by Producer is stored and from where the Consumer consumes the item if needed. There are two versions of this problem: first one is known as unbounded buffer problem in which Producer can keep on producing items and there is no limit on size of buffer, the second one is known as bounded buffer problem in which producer can produce up to a certain amount of item and after that it starts waiting for consumer to consume it. We will discuss the bounded buffer problem. First, the Producer and the Consumer will share some common memory, then producer will start producing items. If the total produced item is equal to the size of buffer, producer will wait to get it consumed by the Consumer. Similarly, the consumer first check for the availability of the item and if no item is available, Consumer will wait for producer to produce it. If there are items available, consumer will consume it.

ii) Messaging Passing Method

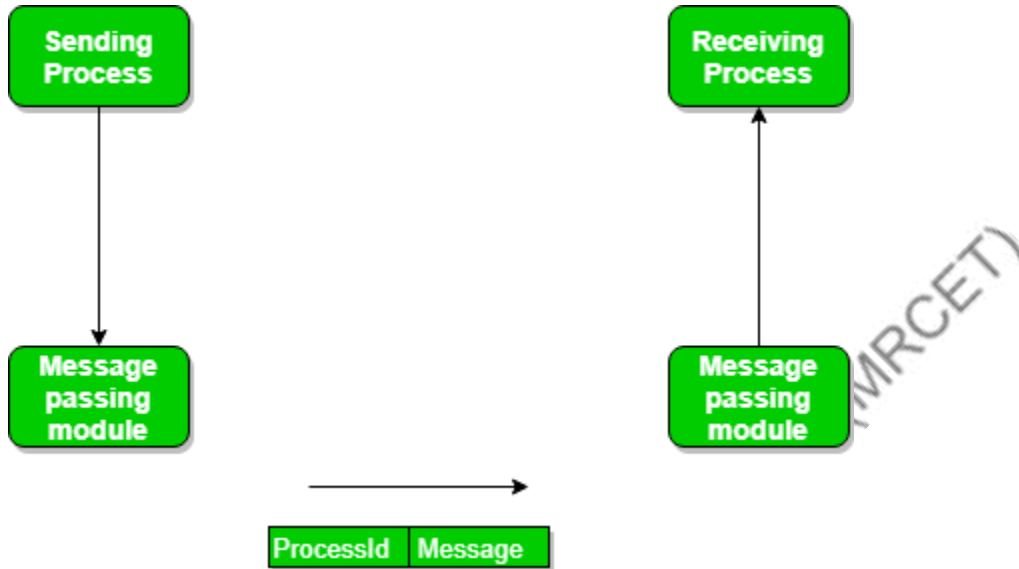
Now, We will start our discussion for the communication between processes via message passing. In this method, processes communicate with each other without using any kind of shared memory. If two processes p1 and p2 want to communicate with each other, they proceed as follow:

Establish a communication link (if a link already exists, no need to establish it again.)

Start exchanging messages using basic primitives.

We need at least two primitives:

- send(message, destination) or send(message)
- receive(message, host) or receive(message)



The message size can be of fixed size or of variable size. If it is of fixed size, it is easy for OS designer but complicated for programmer and if it is of variable size then it is easy for programmer but complicated for the OS designer. A standard message can have two parts: header and body.

The header part is used for storing Message type, destination id, source id, message length and control information. The control information contains information like what to do if runs out of buffer space, sequence number, priority. Generally, message is sent using FIFO style.

Distributed Objects and Remote Invocation

Topics covered in this chapter:

- Communication between distributed objects
- Remote procedure call
- Events and notification
- Java RMI

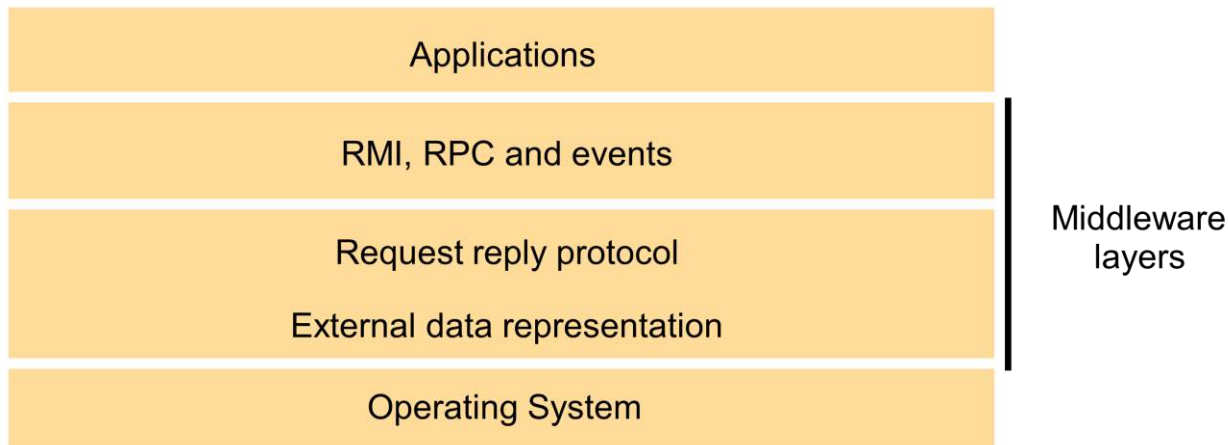
What are issues in distributing objects?

- How can we identify objects?

- What is involved in invoking a method implemented by the class?
 - o What methods are available?
 - o How can we pass parameters and get results?
- Can we track events in a distributed system?

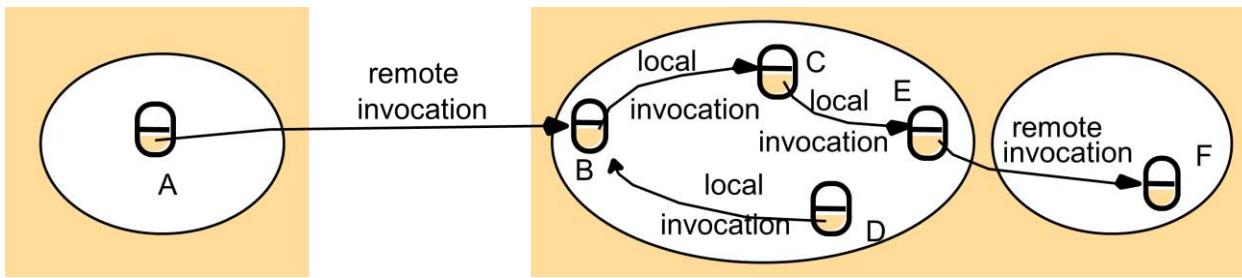
Distributed Objects

- Remote procedure call – client calls the procedures in a server program that is running in a different process
- Remote method invocation (RMI) – an object in one process can invoke methods of objects in another process
- Event notification – objects receive notification of events at other objects for which they have registered
- This mechanism must be location-transparent.
- Middleware Roles
 - provide high-level abstractions such as RMI
 - enable location transparency
 - free from specifics of
 - communication protocols
 - operating systems and communication hardware
 - interoperability



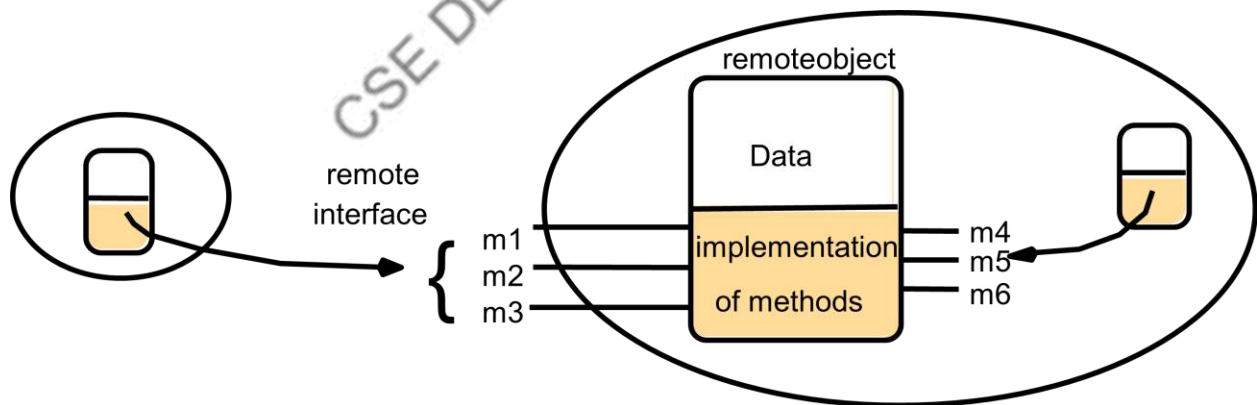
The Distributed Objects Model

- Remote method invocation – Method invocations between objects in different processes, whether in the same computer or not.
- Local method invocation – Method invocations between objects in the same process.
- Remote object – Objects that can receive remote invocations.
- Remote and local method invocations are shown in Figure 5.3.



- each process contains objects, some of which can receive remote invocations, others only local invocations
- those that can receive remote invocations are called *remote objects*
- objects need to know the *remote object reference* of an object in another process in order to invoke its methods. How do they get it?
- the *remote interface* specifies which methods can be invoked remotely
- Remote object reference
 - An object must have the remote object reference of an object in order to do remote invocation of an object
 - Remote object references may be passed as input arguments or returned as output arguments
- Remote interface
 - Objects in other processes can invoke only the methods that belong to its remote interface (Figure 5.4).
 - CORBA – uses IDL to specify remote interface

JAVA – extends interface by the **Remote** keyword.

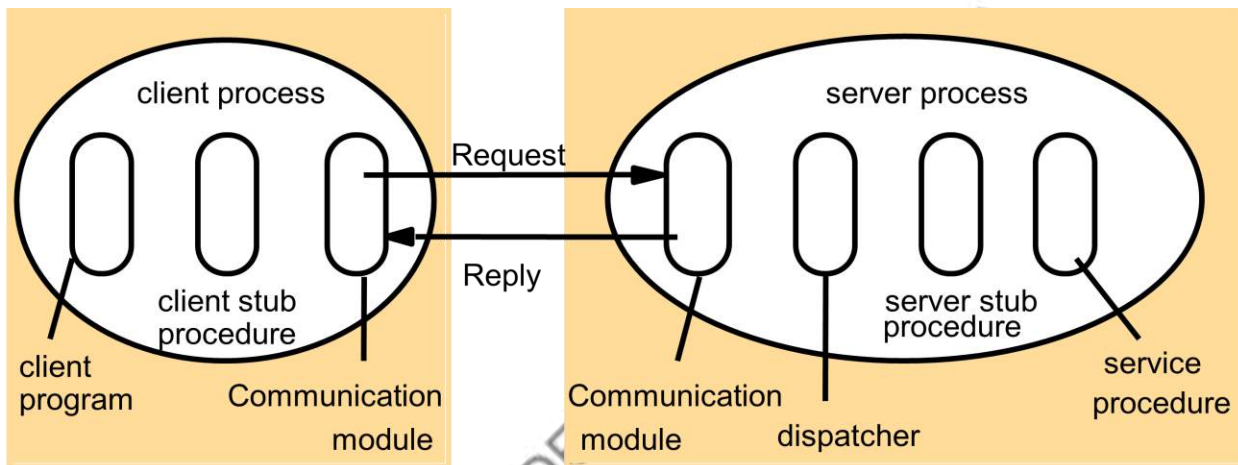


two important issues in making RMI natural extension of local method: (These problems won't occur in the local invocation.)

- **Number of times of invocations** are invoked in response to a single remote invocation
- **Level of location transparency**
- **Exactly once invocation semantics** - Every method is executed exactly once. (Ideal situation)

Remote Procedure Call Basics

- Problems with sockets
 - The read/write (input/output) mechanism is used in socket programming.
 - Socket programming is different from procedure calls which we usually use.
 - To make distributed computing transparent from locations, input/output is not the best way.
- A procedure call is a standard abstraction in local computation.
- Procedure calls are extended to distributed computation in Remote Procedure Call (RPC) as shown in Figure 5.7.
 - A caller invokes execution of procedure in the called via the local stub procedure.
 - The implicit network programming hides all network I/O code from the programmer.
 - Objectives are simplicity and ease of use.

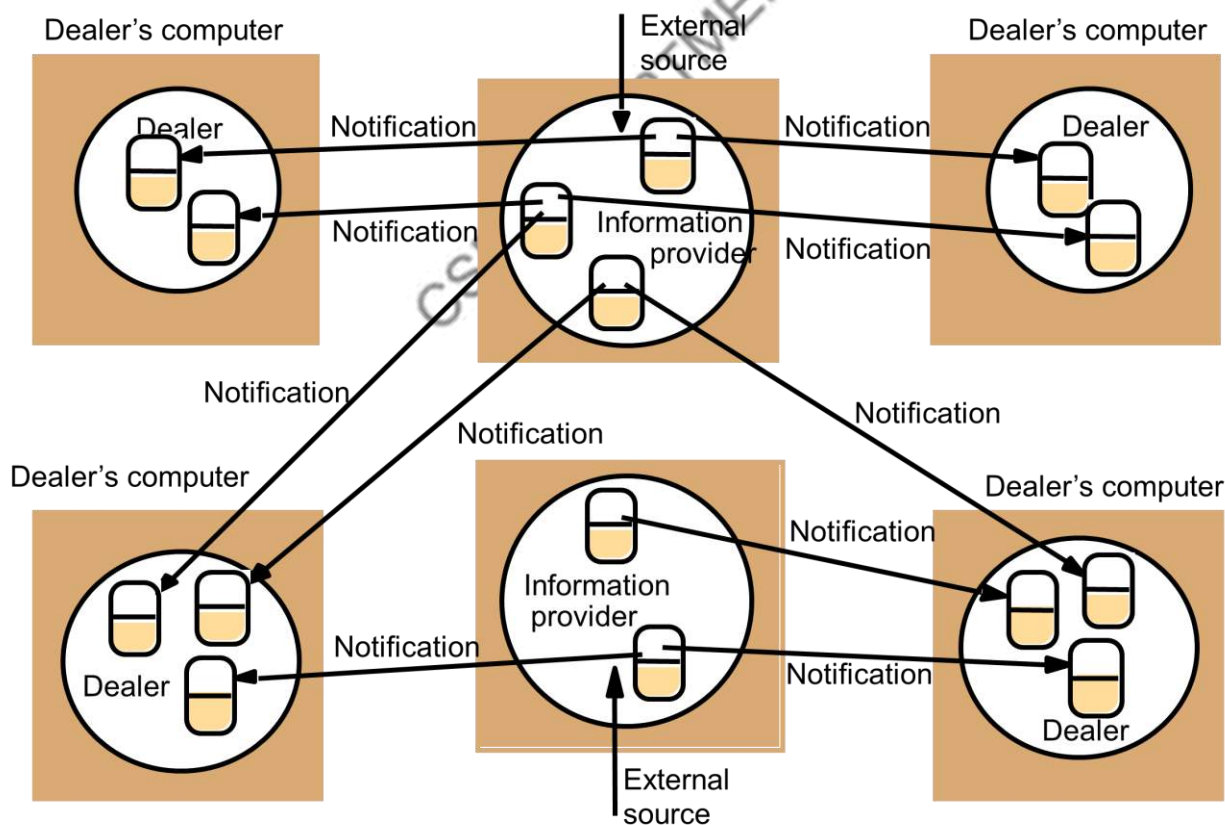


- The concept is to provide a transparent mechanism that enables the user to utilize remote services through standard procedure calls.
- Client sends request, then blocks until a remote server sends a response (reply).
- **Advantages:** user may be unaware of remote implementation (handled in a stub in library); uses standard mechanism.
- **Disadvantages:** prone to failure of components and network; different address spaces; separate process lifetimes.
- Differences with respect to message passing:
 - Message passing systems are peer-to-peer while RPC is more master/slave.
 - In message passing the calling process creates the message while in RPC the system create the message.
- Semantics of RPC:
 - Caller blocks.
 - Caller may send arguments to remote procedure.

- Callee may return results.
- Caller and callee access different address spaces.

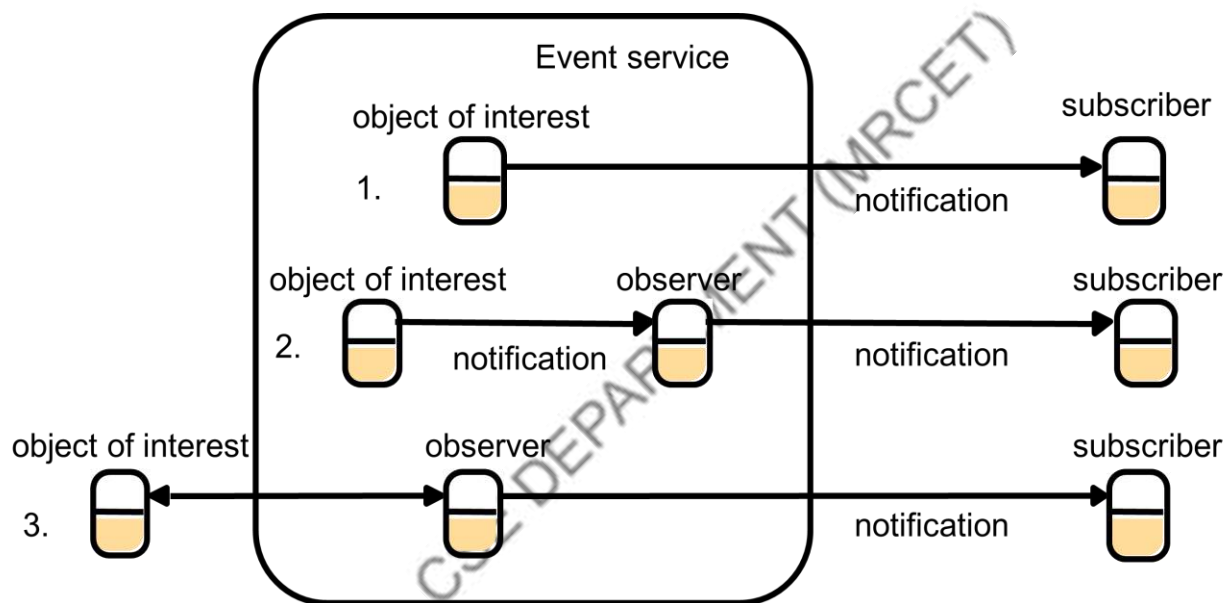
Events and Notifications

- The idea behind the use of events is that one object can react to a change occurring in another object.
- The actions done by the user are seen as events that cause state changes in objects.
- The objects are notified whenever the state changes.
- Local event model can be extended to distributed event-based systems by using the publish-subscribe paradigm.
- In **publish-subscribe** paradigm
 - An object that has event publishes.
 - Those that have interest subscribe.
- Objects that represent events are called **notifications**.
- Distributed event-based systems have two main characteristics:
 - **Heterogeneous** – Event-based systems can be used to connect heterogeneous components in the Internet.
 - **Asynchronous** – Notification are sent asynchronously by event-generating objects to those subscribers.



- The architecture of distributed event notification specifies the roles of participants as in Fig. 5.10:
 - It is designed in a way that publishers work independently from subscribers.

- Event service maintains a database of published events and of subscribers' interests.
- The **roles of the participants** are:
 - **Object of Interest** – This is an object experiences changes of state, as a result of its operations being invoked.
- The **roles of the participants** are (continued):
 - **Event** – An event occurs at an object of interest as the result of the completion of a method invocation.
 - **Notification** – A notification is an object that contains information about an event.
 - **Subscriber** – A subscriber is an object that has subscribed to some type of events in another object.
 - **Observer objects** – The main purpose of an observer is to separate an object of interest from its subscribers.
 - **Publisher** – This is an object that declares that it will generate notifications of particular types of event.



- A variety of **delivery semantics** can be employed:
 - **IP multicast protocol** – information delivery on the latest state of a player in an Internet game
 - **Reliable multicast protocol** – information provider / dealer
 - **Totally ordered multicast** - Computer Supported Cooperative Working (CSCW) environment
 - **Real-time** – nuclear power station / hospital patient monitor
- **Roles for observers** – the task of processing notifications can be divided among observers:
 - **Forwarding** – Observers simply forward notifications to subscribers.
 - **Filtering of notifications** – Observers address notifications to those subscribers who find these notifications are useful.
 - **Patterns of events** – Subscribers can specify patterns of events of interest.

- **Notification mailboxes** – A subscriber can set up a notification mailbox which receives the notification on behalf of the subscriber.

Java RMI

- **Start the server** in one window or in the background with the security policy

java -Djava.security.policy=policy HelloServer

or without the security policy

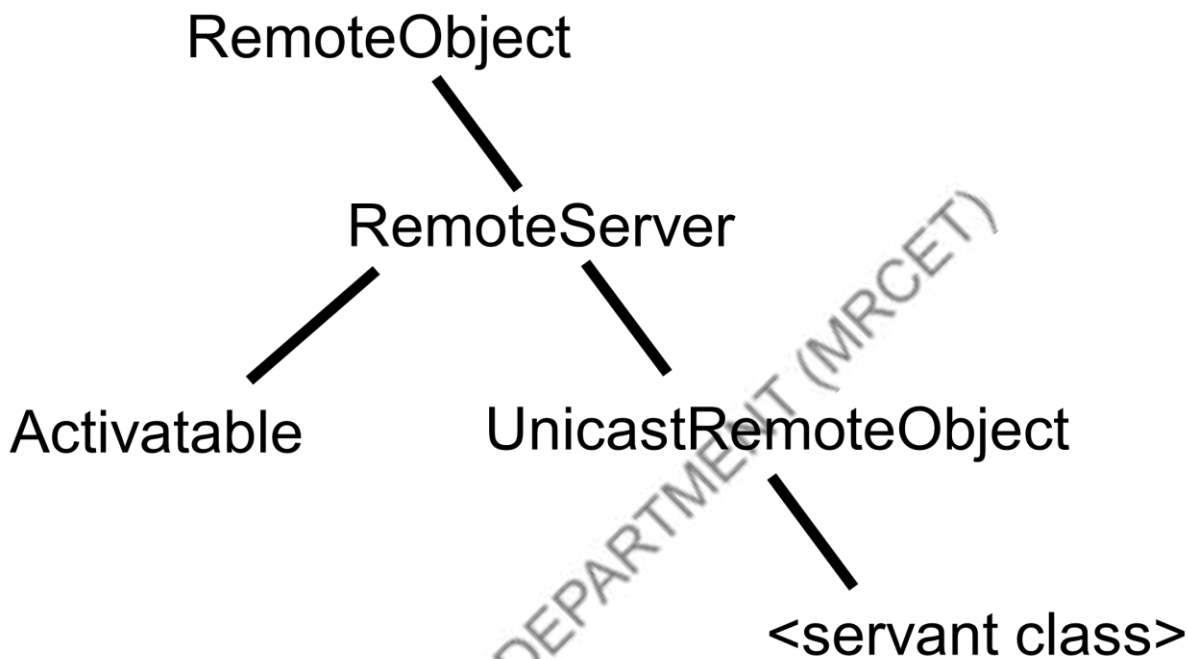
java HelloServer

- **Run the client** in another window

java HelloClient testing

- An object must have the remote object reference of other object in order to do remote invocation of that object.
- Parameter and result passing
 - Remote object references may be passed as input arguments or returned as output arguments.
 - Parameters of a method in Java are input parameters.
 - Returned result of a method in Java is the single output parameter.
 - Objects are serialized to be passed as parameters.
 - When a remote object reference is returned, it can be used to invoke remote methods.
 - Local serializable objects are copied by value.
- Downloading of classes
 - Java is designed to allow classes to be downloaded from one virtual machine to another.
 - If the recipient of a remote object reference does not possess the proxy class, its code is downloaded automatically.
- RMIregistry
 - The RMIregistry is designed to allow is the binder for Java RMI.
 - It maintains a table mapping textual, URL-style names to references to remote objects.
- Server Program
 - The server consists of a main method and a servant class to implement each of its remote interface.
 - The main method of a server needs to create a security manager to enable Java security to apply the protection for an RMI server.
- Client Program
 - Any client program needs to get started by using a binder to look up a remote reference.

- A client can set a security manager and then looks up a remote object reference.
- **Callback** refers to server's action in notifying the client.
- **Callback Facility** - Instead of client polling the server, the server calls a method in the client when it is updated.
- Details
 - Client creates a remote object that implements an interface for the server to call.
 - The server provides an operation for clients to **register** their callbacks.
 - When an event occurs, the server calls the interested clients.



RMI Summary

- Each object has a (global) remote object reference and a remote interface that specifies which of its operations can be invoked remotely.
- Local method invocations provide exactly-once semantics; the best RMI can guarantee is at-most-once.
- Middleware components (proxies, skeletons and dispatchers) hide details of marshalling, message passing and object location from programmers.

UNIT IV

Distributed File Systems: Introduction, File service Architecture, Case Study: 1: Sun Network File System , Case Study 2: The Andrew File System.

Distributed Shared Memory: Introduction, Design and Implementation issues, Consistency Models.

Distributed File Systems: Introduction

- File system were originally developed for centralized computer systems and desktop computers.
- File system was as an operating system facility providing a convenient programming interface to disk storage.
- Distributed file systems support the sharing of information in the form of files and hardware resources.
- With the advent of distributed object systems (CORBA, Java) and the web, the picture has become more complex.
- Figure 1 provides an overview of types of storage system.

	Sharing	Persis- tence	Distributed cache/replicas	Consistency maintenance	Example
Main memory	×	×	×	1	RAM
File system	×	✓	×	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Web server
Distributed shared memory	✓	×	✓	✓	Ivy (Ch. 18)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	✓	OceanStore(Ch. 10)

Figure 1. Storage systems and their properties

Types of consistency between copies:

1 - strict one-copy consistency

✓ - approximate consistency

×

- Figure 2 shows a typical layered module structure for the implementation of a non-distributed file system in a conventional operating system.

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

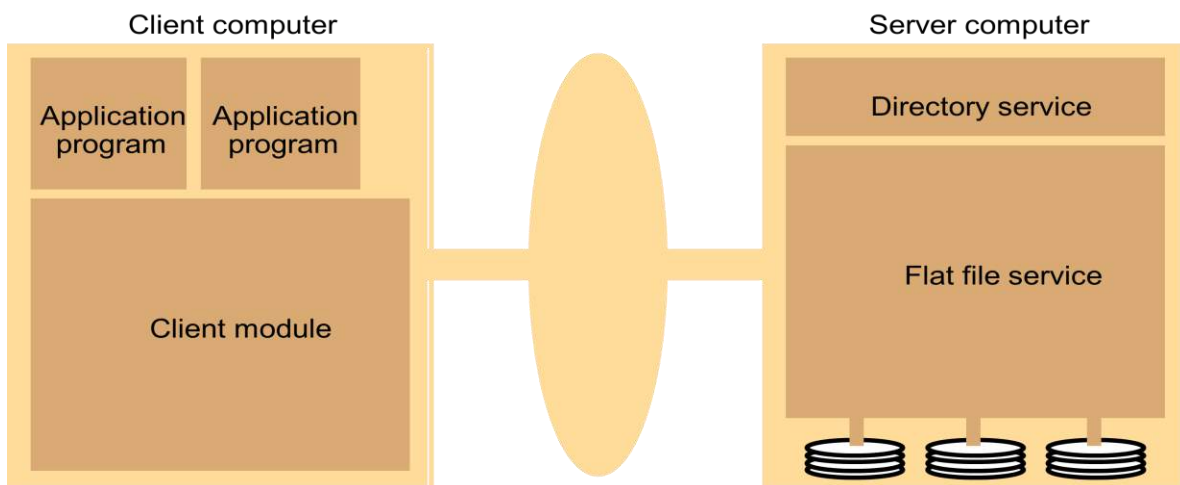
- File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files.
- Files contain both data and attributes.
- A typical attribute record structure is illustrated in Figure 3.

File length
Creation timestamp
Read timestamp
Write timestamp
Attribute timestamp
Reference count
Owner
File type
Access control list

- Distributed File system requirements
 - Related requirements in distributed file systems are:
 - ❖ Transparency
 - ❖ Concurrency
 - ❖ Replication
 - ❖ Heterogeneity
 - ❖ Fault tolerance
 - ❖ Consistency
 - ❖ Security
 - ❖ Efficiency

File Service Architecture

- An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components:
 - A flat file service
 - A directory service
 - A client module.
- The relevant modules and their relationship is shown in Figure



- The Client module implements exported interfaces by flat file and directory services on server side.
- Responsibilities of various modules can be defined as follows:
 - Flat file service:
 - ❖ Concerned with the implementation of operations on the contents of file. Unique File Identifiers (UFIDs) are used to refer to files in all requests for flat file service operations. UFIDs are long sequences of bits chosen so that each file has a unique among all of the files in a distributed system.
 - Directory service:
 - ❖ Provides mapping between text names for the files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to directory service. Directory service supports functions needed generate directories, to add new files to directories.
 - Client module:
 - ❖ It runs on each computer and provides integrated service (flat file and directory) as a single API to application programs. For example, in UNIX hosts, a client module emulates the full set of Unix file operations.
 - ❖ It holds information about the network locations of flat-file and directory server processes; and achieve better performance through implementation of a cache of recently used file blocks at the client.
 - Access control
 - ❖ In distributed implementations, access rights checks have to be performed at the server because the server RPC interface is an otherwise unprotected point of access to files.
 - Directory service interface
 - ❖ Figure contains a definition of the RPC interface to a directory service.

<i>Lookup(Dir, Name) -> FileId</i>	Locates the text name in the directory and
-throws NotFound	returns the relevant UFID. If <i>Name</i> is not in the directory, throws an exception.
<i>AddName(Dir, Name, File)</i>	If <i>Name</i> is not in the directory, adds(<i>Name,File</i>)
-throws NameDuplicate	to the directory and updates the file's attribute record.
	If <i>Name</i> is already in the directory: throws an exception.
<i>UnName(Dir, Name)</i>	If <i>Name</i> is in the directory, the entry containing <i>Name</i>
	is removed from the directory.
	If <i>Name</i> is not in the directory: throws an exception.
<i>GetNames(Dir, Pattern) -> NameSeq</i>	Returns all the text names in the directory that match
	the regular expression <i>Pattern</i> .

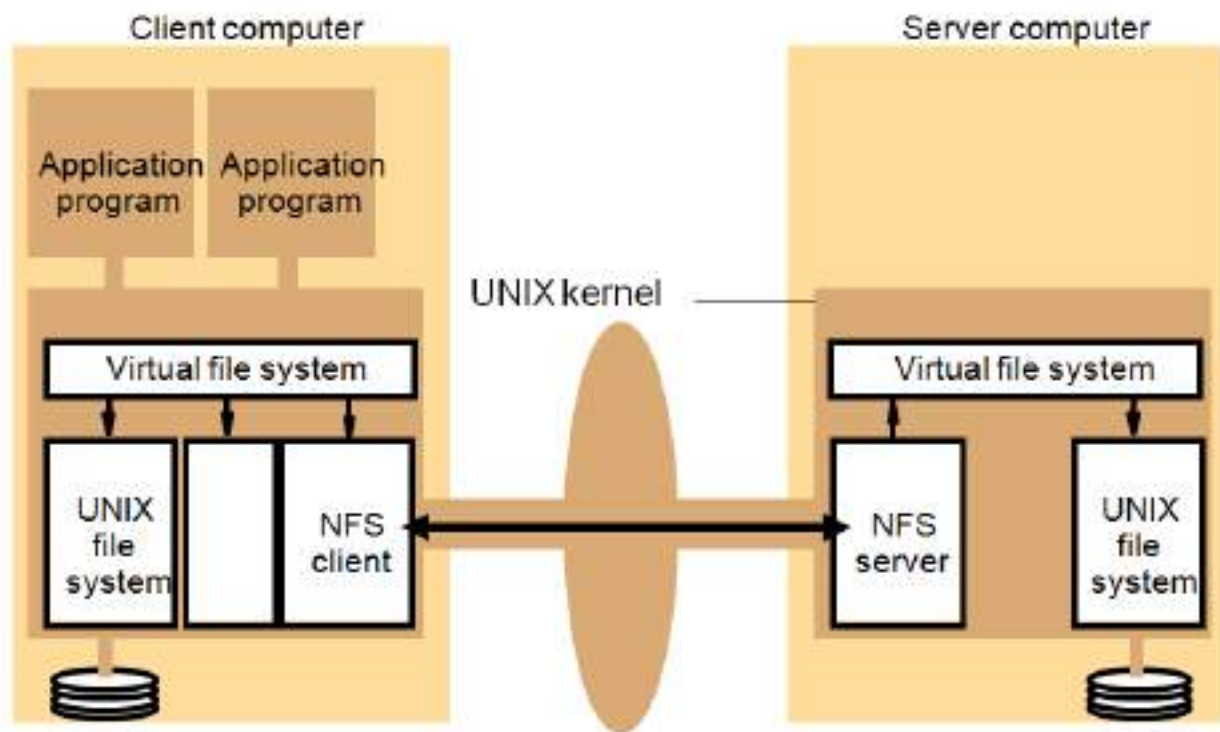
- Hierarchic file system
 - ❖ A hierarchic file system such as the one that UNIX provides consists of a number of directories arranged in a tree structure.
- File Group
 - ❖ A file group is a collection of files that can be located on any server or moved between servers while maintaining the same names.
 - A similar construct is used in a UNIX file system.
 - It helps with distributing the load of file serving between several servers.
 - File groups have identifiers which are unique throughout the system (and hence for an open system, they must be globally unique).

Case Study: 1: Sun Network File System

NFS (Network File System)

- Developed by Sun Microsystems (in 1985)
- Most popular, open, and widely used.
- NFS protocol standardized through IETF (RFC 1813)

NFS architecture



fh = file handle:

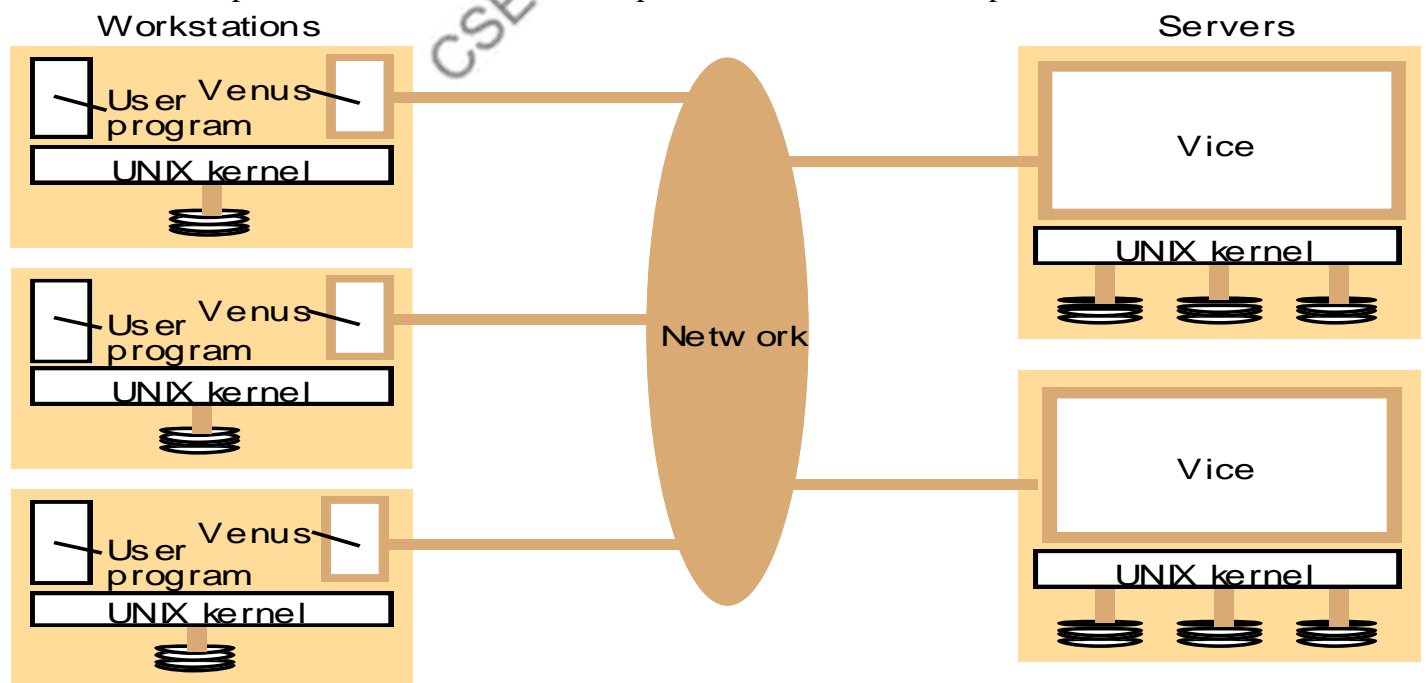
Filesystem identifier	i-node number	i-node generation
-----------------------	---------------	-------------------

A simplified representation of the RPC interface provided by NFS version 3 servers is shown in Figure

- *read(fh, offset, count) -> attr, data*
- *write(fh, offset, count, data) -> attr*
- *create(dirfh, name, attr) -> newfh, attr*
- *remove(dirfh, name) status*
- *getattr(fh) -> attr*
- *setattr(fh, attr) -> attr*
- *lookup(dirfh, name) -> fh, attr*
- *rename(dirfh, name, todirfh, toname)*
- *link(newdirfh, newname, dirfh, name)*
- *readdir(dirfh, cookie, count) -> entries*
- *symlink(newdirfh, newname, string) -> status*
- *readlink(fh) -> string*
- *mkdir(dirfh, name, attr) -> newfh, attr*
- *rmdir(dirfh, name) -> status*
- *statfs(fh) -> fsstats*

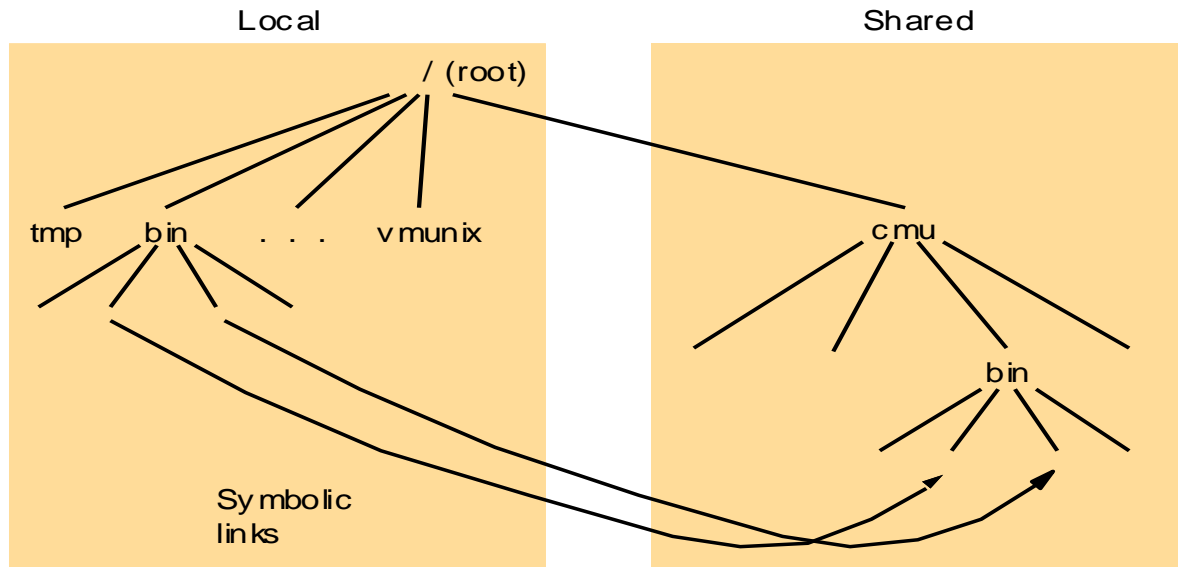
case study 2: AFS (Andrew File System)

- Developed by Carnegie Mellon University as part of Andrew distributed computing environments (in 1986)
- A research project to create campus wide file system.
- Public domain implementation is available on Linux (LinuxAFS)
- It was adopted as a basis for the DCE/DFS file system in the Open Software Foundation (OSF, www.opengroup.org) DEC (Distributed Computing Environment)
- Like NFS, AFS provides transparent access to remote shared files for UNIX programs running on workstations.
- AFS is implemented as two software components that exist at UNIX processes called Vice and Venus.

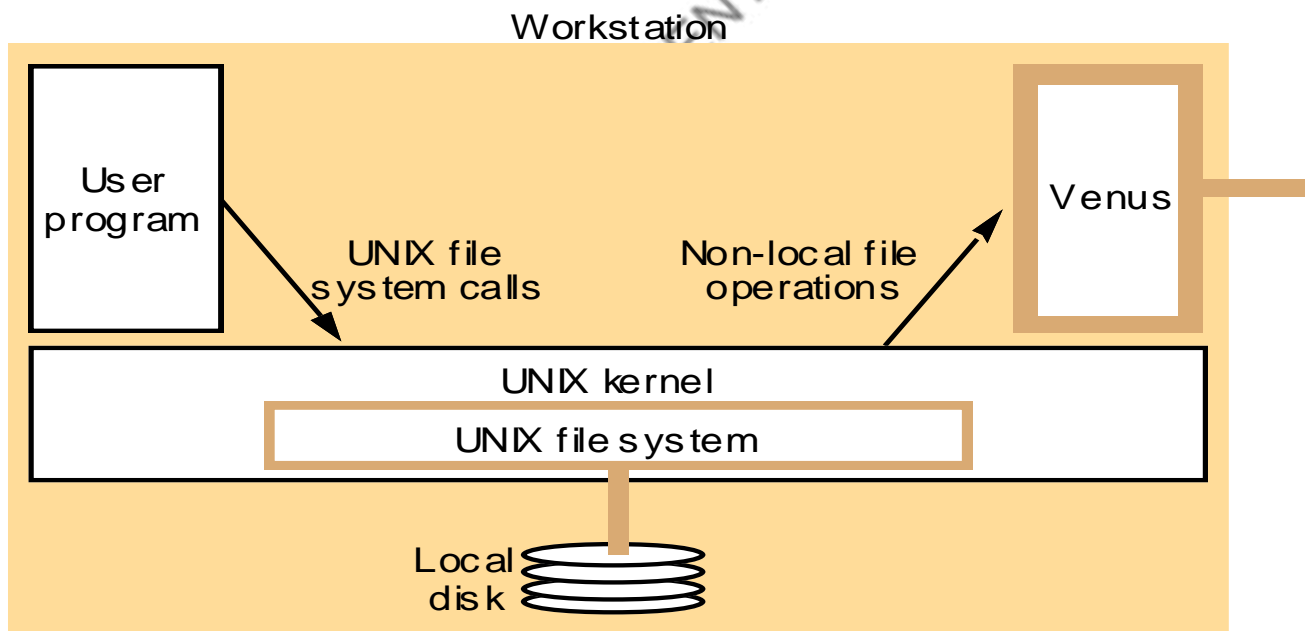


- The files available to user processes running on workstations are either local or shared.

- Local files are handled as normal UNIX files.
- They are stored on the workstation's disk and are available only to local user processes.
- Shared files are stored on servers, and copies of them are cached on the local disks of workstations.
- The name space seen by user processes is illustrated in Figure 12.



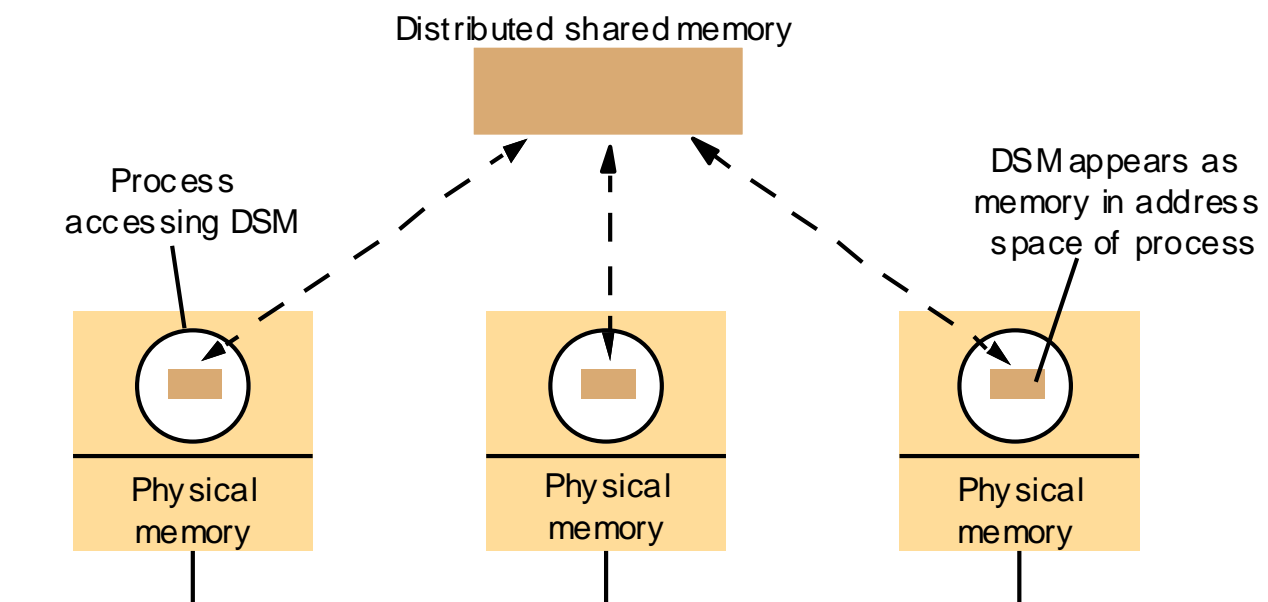
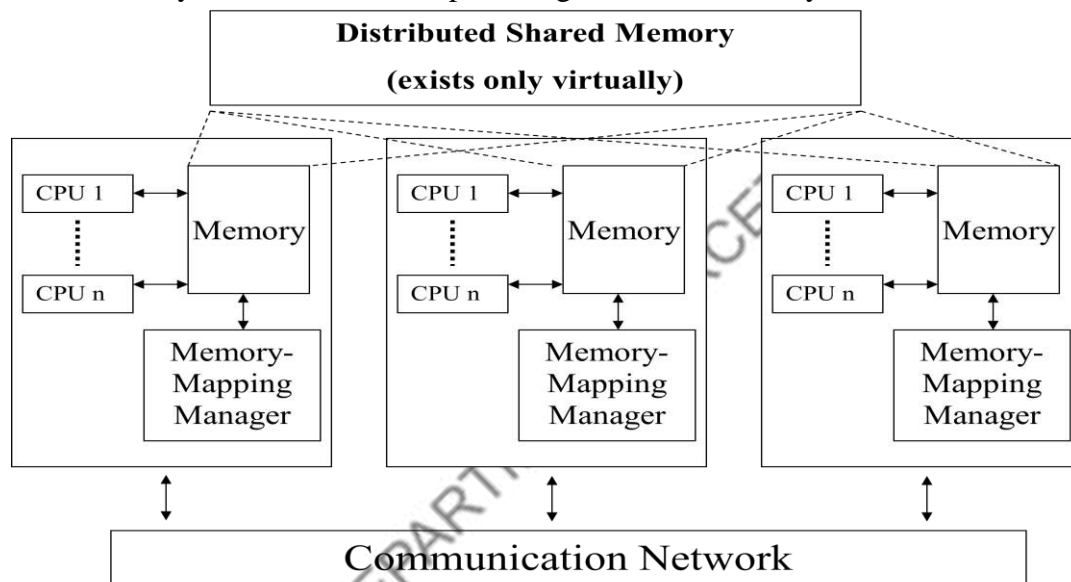
- The UNIX kernel in each workstation and server is a modified version of BSD UNIX.
- The modifications are designed to intercept open, close and some other file system calls when they refer to files in the shared name space and pass them to the Venus process in the client computer.



System call interception in AFS

Distributed Shared Memory: Introduction

- Distributed Shared Memory (DSM) allows programs running on separate computers to share data without the programmer having to deal with sending messages
- Instead underlying technology will send the messages to keep the DSM consistent (or relatively consistent) between computers
- DSM allows programs that used to operate on the same computer to be easily adapted to operate on separate computers
- Programs access what appears to them to be normal memory
- Hence, programs that use DSM are usually shorter and easier to understand than programs that use message passing
- However, DSM is not suitable for all situations. Client-server systems are generally less suited for DSM, but a server may be used to assist in providing DSM functionality for data shared between clients



DSM vs Message passing

DSM	Message passing
Variables are shared directly	Variables have to be marshalled yourself
Processes can cause error to one another by altering data	Processes are protected from one another by having private address spaces
Processes may execute with non-overlapping lifetimes	Processes must execute at the same time
Invisibility of communication's cost	Cost of communication is obvious

DSM implementations

- Hardware: Mainly used by shared-memory multiprocessors. The hardware resolves LOAD and STORE commands by communicating with remote memory as well as local memory
- Paged virtual memory: Pages of virtual memory get the same set of addresses for each program in the DSM system. This only works for computers with common data and paging formats. This implementation does not put extra structure requirements on the program since it is just a series of bytes.
- Middleware: DSM is provided by some languages and middleware without hardware or paging support. For this implementation, the programming language, underlying system libraries, or middleware send the messages to keep the data synchronized between programs so that the programmer does not have to.

Efficiency

- DSM systems can perform almost as well as equivalent message-passing programs for systems that run on about 10 or less computers.
- There are many factors that affect the efficiency of DSM, including the implementation, design approach, and memory consistency model chosen.

Design and implementation issues

- Byte-oriented: This is implemented as a contiguous series of bytes. The language and programs determine the data structures
- Object-oriented: Language-level objects are used in this implementation. The memory is only accessed through class routines and therefore, OO semantics can be used when implementing this system
- Immutable data: Data is represented as a group of many tuples. Data can only be accessed through read, take, and write routines

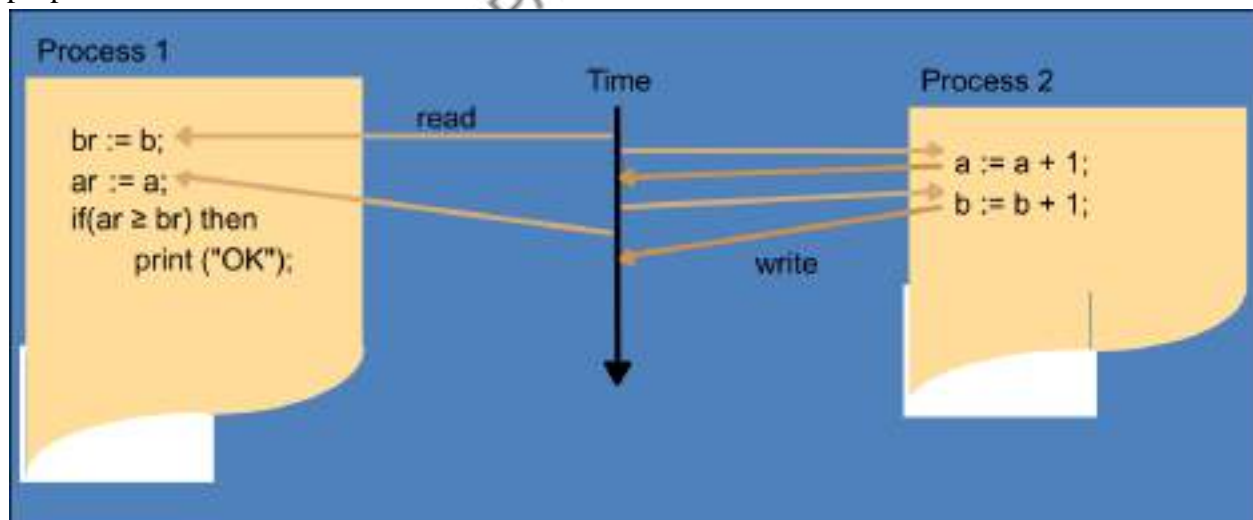
Memory consistency

- To use DSM, one must also implement a distributed synchronization service. This includes the use of locks, semaphores, and message passing
- Most implementations, data is read from local copies of the data but updates to data must be propagated to other copies of the data
- Memory consistency models determine when data updates are propagated and what level of inconsistency is acceptable



Memory consistency models

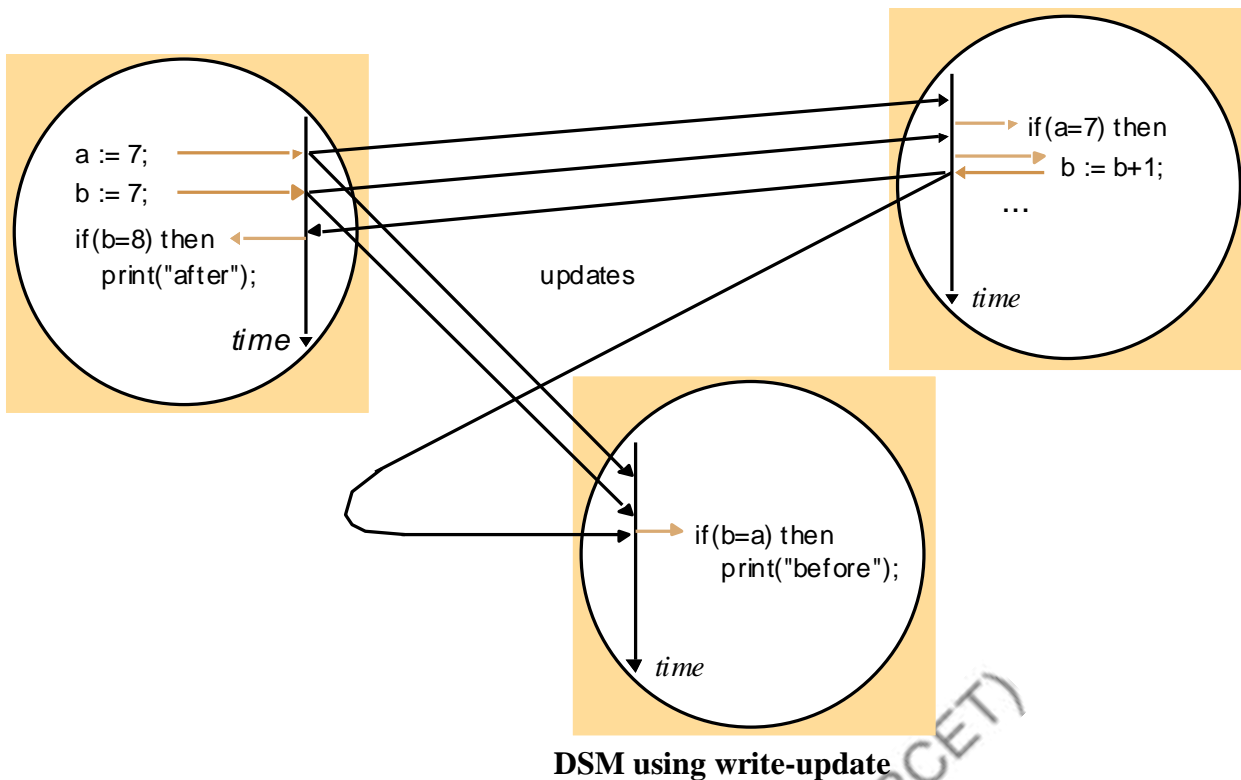
- Linearizability or atomic consistency is the strongest model. It ensures that reads and writes are made in the proper order. This results in a lot of underlying messaging being passed.
 - Variables can only be changed by a write operation
 - The order of operations is consistent with the real times at which the operations occurred in the actual execution
- Sequential consistency is strong, but not as strict. Reads and writes are done in the proper order in the context of individual programs.
 - The order of operations is consistent with the program order in which each individual client executed them
- Coherence has significantly weaker consistency. It ensures writes to individual memory locations are done in the proper order, but writes to separate locations can be done in improper order
- Weak consistency requires the programmer to use locks to ensure reads and writes are done in the proper order for data that needs it



Interleaving under sequential consistency

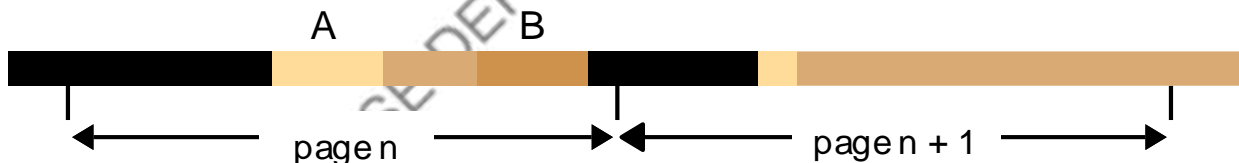
Update options

- Write-update: Each update is multicast to all programs. Reads are performed on local copies of the data
- Write-invalidate: A message is multicast to each program invalidating their copy of the data before the data is updated. Other programs can request the updated data



Granularity

- Granularity is the amount of data sent with each update
- If granularity is too small and a large amount of contiguous data is updated, the overhead of sending many small messages leads to less efficiency
- If granularity is too large, a whole page (or more) would be sent for an update to a single byte, thus reducing efficiency



Thrashing

- Thrashing occurs when network resources are exhausted, and more time is spent invalidating data and sending updates than is used doing actual work
- Based on system specifics, one should choose write-update or write-invalidate to avoid thrashing

Consistency models:

- **Strict consistency**
- **Sequential consistency**
- **Release Consistency**
- **Causal consistency**
- **Processor consistency**

UNIT V

Transactions and Concurrency Control: Introduction, Transactions, Nested Transactions, Locks, Optimistic concurrency control, Timestamp ordering, Comparison of methods for concurrency control.

Distributed Transactions: Introduction, Flat and Nested Distributed Transactions, Atomic commit protocols, Concurrency control in distributed transactions, Distributed deadlocks, Transaction recovery.

Transactions and Concurrency Control: Introduction

Banking transaction for a customer (e.g., at ATM or browser)

Transfer \$100 from saving to checking account;

Transfer \$200 from money-market to checking account;

Withdraw \$400 from checking account.

Transaction (invoked at client): /* Every step is an RPC */

1. savings.withdraw(100) /* includes verification */
2. checking.deposit(100) /* depends on success of 1 */
3. mnymkt.withdraw(200) /* includes verification */
4. checking. deposit(200) /* depends on success of 3 */
5. checking.withdraw(400) /* includes verification */
6. dispense(400)
7. commit

❖ All the following are RPCs from a client to the server

❖ Transaction calls that can be made at a client, and return values from the server:

Transactions

openTransaction() -> trans;

starts a new transaction and delivers a unique transaction identifier (TID) trans. This TID will be used in the other operations in the transaction.

closeTransaction(trans) -> (commit, abort);

ends a transaction: a commit return value indicates that the transaction has committed; an abort return value indicates that it has aborted.

abortTransaction(trans);

aborts the transaction.

❖ TID can be passed implicitly (for other operations between open and close) with CORBA

❖ deposit(amount)

- ❖ deposit amount in the account
- ❖ withdraw(amount)
- ❖ withdraw amount from the account
- ❖ getBalance() -> amount
- ❖ return the balance of the account
- ❖ setBalance(amount)
- ❖ set the balance of the account to amount
- ❖ Sequence of operations that forms a single step, transforming the server data from one consistent state to another.
 - ☐ All or nothing principle: a transaction either completes successfully, and the effects are recorded in the objects, or it has no effect at all. (even with multiple clients, or crashes)
- ❖ A transactions is indivisible (atomic) from the point of view of other transactions
 - ☐ No access to intermediate results/states of other transactions
 - ☐ Free from interference by operations of other transactions

But...

- ❖ Transactions could run concurrently, i.e., with multiple clients
- ❖ Transactions may be distributed, i.e., across multiple servers
- ❖ Atomicity: All or nothing
- ❖ Consistency: if the server starts in a consistent state, the transaction ends the server in a consistent state.
- ❖ Isolation: Each transaction must be performed without interference from other transactions, i.e., the non-final effects of a transaction must not be visible to other transactions.
- ❖ Durability: After a transaction has completed successfully, all its effects are saved in permanent storage.
- ❖ Atomicity: store tentative object updates (for later undo/redo) – many different ways of doing this
- ❖ Durability: store entire results of transactions (all updated objects) to recover from permanent server crashes.
- ❖ The effect of an operation refers to
- ❖ The value of an object set by a write operation
- ❖ The result returned by a read operation.
- ❖ Two operations are said to be conflicting operations, if their combined effect depends on the order they are executed, e.g., read-write, write-read, write-write (all on same variables). NOT read-read, NOT on different variables.
- ❖ Two transactions are serially equivalent if and only if all pairs of conflicting operations (pair containing one operation from each transaction) are executed in the same order (transaction order) for all objects (data) they both access.

- ❖ Why? Can start from original operation sequence and swap the order of non-conflicting operations to obtain a series of operations where one transaction finishes completely before the second transaction starts
- ❖ Why is the above result important? Because: Serial equivalence is the basis for concurrency control protocols for transactions.

<i>Operations of different transactions</i>			<i>Conflict</i>	<i>Reason</i>
<i>read</i>	<i>read</i>	No		Because the effect of a pair of <i>read</i> operations does not depend on the order in which they are executed
<i>read</i>	<i>write</i>	Yes		Because the effect of a <i>read</i> and a <i>write</i> operation depends on the order of their execution
<i>write</i>	<i>write</i>	Yes		Because the effect of a pair of <i>write</i> operations depends on the order of their execution

Concurrency control

- Lost update
 - 3 accounts (A, B, C)
 - » with balances 100, 200, 300
 - T1 transfers from A to B, for 10% increase
 - T2 transfers from C to B, for 10% increase
 - Both T1, T2 read balance of B (200)
 - T1 overwrites the update by T2
 - » Without seeing it
- Inconsistent retrievals
 - T1: transfers 10% of account A to account B
 - T2: computes sum of account balances
 - T2 computes sum before T1 updates B

Recoverability from aborts

- Servers must prevent a aborting Tx from affecting other concurrent Tx's.
 - Dirty reads:
 - » T2 sees result update by T1 on account A
 - » T2 performs its own update on A & then commits.

- » T1 aborts -> T2 has seen a “transient” value
 - T2 is not recoverable
- » If T2 delays its commit until T1’s outcome is resolved:
 - Abort(T1) -> Abort(T2)
 - However, if T3 has seen results of T2:
 - Abort(T2) -> Abort(T3) !
- Premature writes:
 - Assume server implements abort by maintaining the “before” image of all update operations
 - » T1 & T2 both updates account A
 - » T1 completes its work before T2
 - » If T1 commits & T2 aborts, the balance of A is correct
 - » If T1 aborts & T2 commits, the “before” image that is restored corresponds to the balance of A before T2
 - » If both T1 & T2 abort, the “before” image that is restored corresponds to the balance of A as set by T1
- Tx’s should delay both their reads & updates in order to avoid interference
 - Strict execution -> enforce isolation
- Servers should maintain tentative versions of objects in volatile memory
- Tx’s should delay both their reads & updates in order to avoid interference
 - Strict execution -> enforce isolation
- Servers should maintain tentative versions of objects in volatile memory

Concurrency Control: Locks

- Transactions:
 - Must be scheduled so that their effect on shared data is serially equivalent
 - Two types of approach
 - » Pessimistic → If something can go wrong, it will

Operations are synchronized before they are carried out

- » Optimistic → In general, nothing will go wrong

Operations are carried out, synchronization at the end of the transaction

- Locks (pessimistic)

- » can be used to ensuring serializability
 - » lock(x), unlock(x)
- Oldest and most widely used CC algorithm
- A process before read/write → requests the scheduler to grant a lock
- Upon finishing read/write → the lock is released
- In order to ensure serialized transaction Two Phase Locking (2PL) is used
- How Locks prevent consistency problems
 - Lost update and inconsistent retrieval:
 - Causes:
 - » are caused by the conflict between $r_i(x)$ and $w_j(x)$
 - » two transactions read a value and use it to compute new value
 - Prevention:
 - » delay the reads of later transactions until the earlier ones have completed
 - » Disadvantage of Locking
 - Deadlocks
- Strict 2PL avoids Cascading Aborts
 - A situation where a committed transaction has to be undone because it saw a file it shouldn't have seen.
- Problems of Locking
 - Deadlocks
 - Livelocks
 - » A transaction can't proceed for an indefinite amount of time while other transactions continue normally. It happens due to unfair locking.
 - Lock overhead
 - » If the system doesn't allow shared access--wastage of resources
 - Avoidance of Cascading Aborts may be costly
 - » Strict 2PL in fact, reduces the effect of concurrency
- Transaction managers (on server side) set locks on objects they need. A concurrent trans. cannot access locked objects.
- Two phase locking:
 - In the first (growing) phase of the transaction, new locks are only acquired, and in the second (shrinking) phase, locks are only released.

- A transaction is not allowed acquire *any* new locks, once it has released any one lock.

Deadlock with write locks

Deadlock with write locks

Transaction <i>T</i>		Transaction <i>U</i>	
Operations	Locks	Operations	Locks
<i>a.deposit(100);</i>	write lock <i>A</i>		
<i>b.withdraw(100)</i>		<i>b.deposit(200)</i>	write lock <i>B</i>
...	waits for <i>U</i> 's lock on <i>B</i>	<i>a.withdraw(200);</i>	waits for <i>T</i> 's lock on <i>A</i>
...		...	
...		...	

T locks *A* and waits for *U* to release the lock on *B*, *U* on the other hand locks *B* and waits for *T* to release the lock on *A*

➔ Circular hold and wait ➔ Deadlock

❖ Necessary conditions for deadlocks

- ☐ Non-shareable resources (exclusive lock modes)
- ☐ No preemption on locks
- ☐ Hold & Wait or Circular Wait

Naïve Deadlock Resolution Using Timeout

Transaction T		Transaction U	
Operations	Locks	Operations	Locks
<i>a.deposit(100);</i>	write lock _A		
<i>b.withdraw(100)</i>		<i>b.deposit(200)</i>	write lock _B
...	waits for U_s	<i>a.withdraw(200);</i>	waits for T's
	lock on <i>B</i>	...	lock on <i>A</i>
	(timeout elapses)	...	
<i>T's lock on A becomes vulnerable,</i>		<i>a.withdraw(200);</i>	write locks _A
<i>unlock_A, abort T</i>			unlock _A , <i>B</i>

Disadvantages?

Strategies to Fight Deadlock

- ☐ Lock timeout (costly and open to false positives)
- ☐ Deadlock Prevention: violate one of the necessary conditions for deadlock (from 2 slides ago), e.g., lock all objects before transaction starts, aborting entire transaction if any fails
- ☐ Deadlock Avoidance: Have transactions declare max resources they will request, but allow them to lock at any time (Banker's algorithm)
- ☐ Deadlock Detection: detect cycles in the wait-for graph, and then abort one or more of the transactions in cycle
- ☐ We have seen locking has some problems
- ☐ OCC based on the following simple idea:

Distributed Transactions

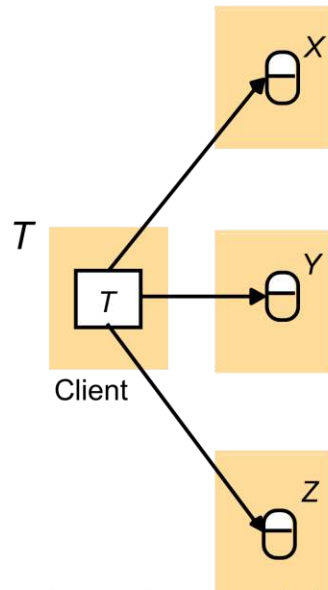
In previous chapter, we discussed transactions accessed objects at a single server. In the general case, a transaction will access objects located in different computers. Distributed transaction accesses objects managed by multiple servers.

The atomicity property requires that either all of the servers involved in the same transaction commit the transaction or all of them abort. Agreement among servers are necessary.

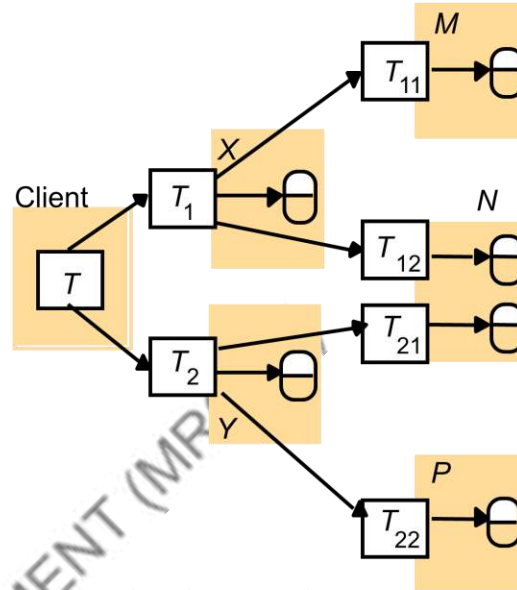
Transaction recovery is to ensure that all objects are recoverable. The values of the objects reflect all changes made by committed transactions and none of those made by aborted ones.

Figure 14.1
Distributed transactions

(a) Flat transaction

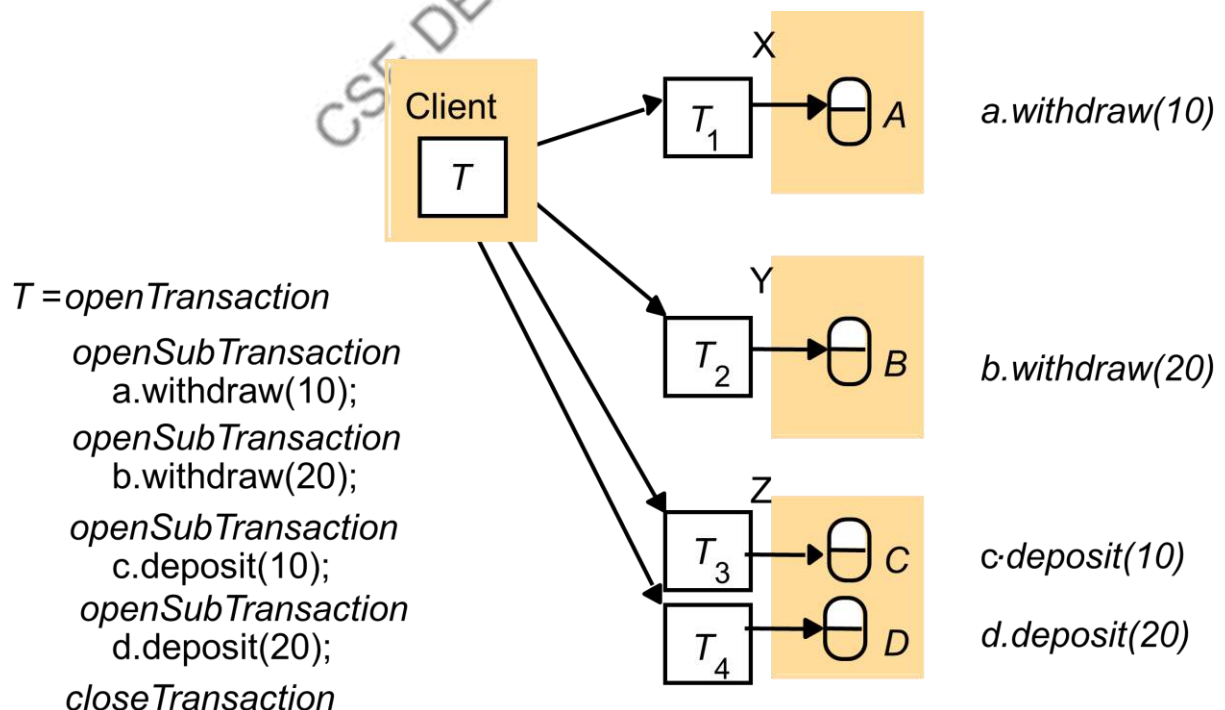


(b) Nested transactions



Flat transaction send out requests to different servers and each request is completed before client goes to the next one. Nested transaction allows sub-transactions at the same level to execute concurrently.

Instructor's Guide for: Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn.4
© Pearson Education 2005



A transaction comes to an end when the client requests that a transaction be committed or aborted.

Simple way is: coordinator to communicate the commit or abort request to all of the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they had carried it out.

Inadequate because when the client requests a commit, it does not allow a server to make a unilateral decision to abort a transaction. E.g. deadlock avoidance may force a transaction to abort at a server when locking is used. So any server may fail or abort and client is not aware.

Allow any participant to abort its part of a transaction. Due to atomicity, the whole transaction must also be aborted.

In the first phase, each participant votes for the transaction to be committed or aborted. Once voted to commit, not allowed to abort it. So before votes to commit, it must ensure that it will eventually be able to carry out its part, even if it fails and is replaced.

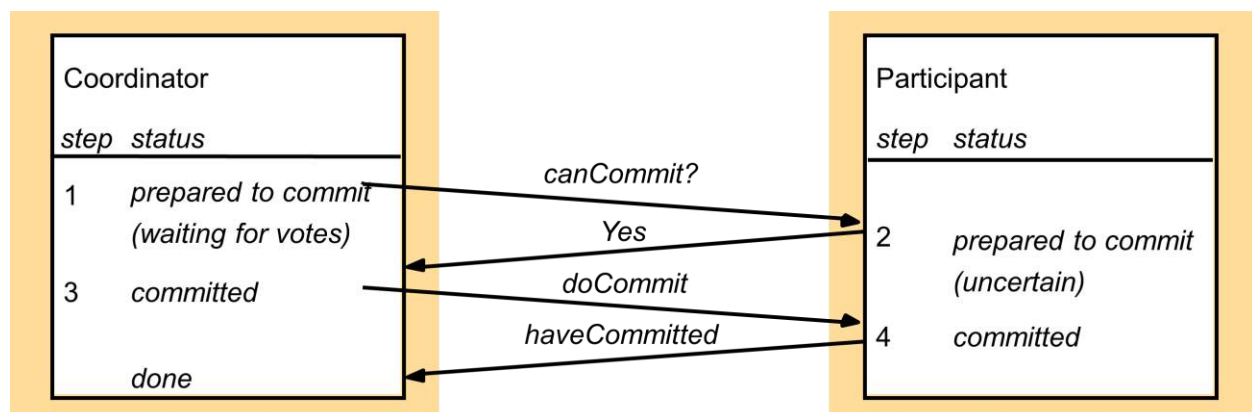
A participant is said to be in a **prepared** state if it will eventually be able to commit it. So each participant needs to save the altered objects in the permanent storage device together with its status-prepared.

Phase 1 (voting phase):

1. The coordinator sends a `canCommit?` request to each of the participants in the transaction.
2. When a participant receives a `canCommit?` request it replies with its vote (Yes or No) to the coordinator. Before voting Yes, it prepares to commit by saving objects in permanent storage. If the vote is No the participant aborts immediately.

Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).
 - (a) If there are no failures and all the votes are Yes the coordinator decides to commit the transaction and sends a `doCommit` request to each of the participants.
 - (b) Otherwise the coordinator decides to abort the transaction and sends `doAbort` requests to all participants that voted Yes.
4. Participants that voted Yes are waiting for a `doCommit` or `doAbort` request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a `haveCommitted` call as confirmation to the coordinator.



Consider when a participant has voted Yes and is waiting for the coordinator to report on the outcome of the vote by telling it to commit or abort.

Such a participant is uncertain and cannot proceed any further. The objects used by its transaction cannot be released for use by other transactions.

Participant makes a `getDecision` request to the coordinator to determine the outcome. If the coordinator has failed, the participant will not get the decision until the coordinator is replaced resulting in extensive delay for participant in uncertain state.

Timeout are used since exchange of information can fail when one of the servers crashes, or when messages are lost So process will not block forever.

Provided that all servers and communication channels do not fail, with N participants

N number of `canCommit?` Messages and replies

Followed by N `doCommit` messages

The cost in messages is proportional to $3N$

The cost in time is three rounds of message.

The cost of `haveCommitted` messages are not counted, which can function correctly without them- their role is to enable server to delete stale coordinator information.

Performance of two-phase commit protocol

Provided that all servers and communication channels do not fail, with N participants

N number of `canCommit?` Messages and replies

Followed by N `doCommit` messages

The cost in messages is proportional to $3N$

The cost in time is three rounds of message.

The cost of `haveCommitted` messages are not counted, which can function correctly without them- their role is to enable server to delete stale coordinator information.

When a participant has voted *Yes* and is waiting for the coordinator to report on the outcome of the vote, such participant is in uncertain stage. If the coordinator has failed, the participant will not be able to get the decision until the coordinator is replaced, which can result in extensive delays for participants in the uncertain state.

Concurrency Control in Distributed Transactions

Concurrency control for distributed transactions: each server applies local concurrency control to its own objects, which ensure transactions serializability locally.

However, the members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. Thus global serializability is required.

Locks

Lock manager at each server decide whether to grant a lock or make the requesting transaction wait.

However, it cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction.

A lock managers in different servers set their locks independently of one another. It is possible that different servers may impose different orderings on transactions.

Timestamp ordering concurrency control

In a single server transaction, the coordinator issues a unique timestamp to each transaction when it starts. Serial equivalence is enforced by committing the versions of objects in the order of the timestamps of transactions that accessed them.

In distributed transactions, we require that each coordinator issue globally unique time stamps. The coordinators must agree as to the ordering of their timestamps. $\langle \text{local timestamp, server-id} \rangle$, the agreed ordering of pairs of timestamps is based on a comparison in which the server-id is less significant.

The timestamp is passed to each server whose objects perform an operation in the transaction.

To achieve the same ordering at all the servers, The servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner. E.g. If T commits after U at server X, T must commits after U at server Y.

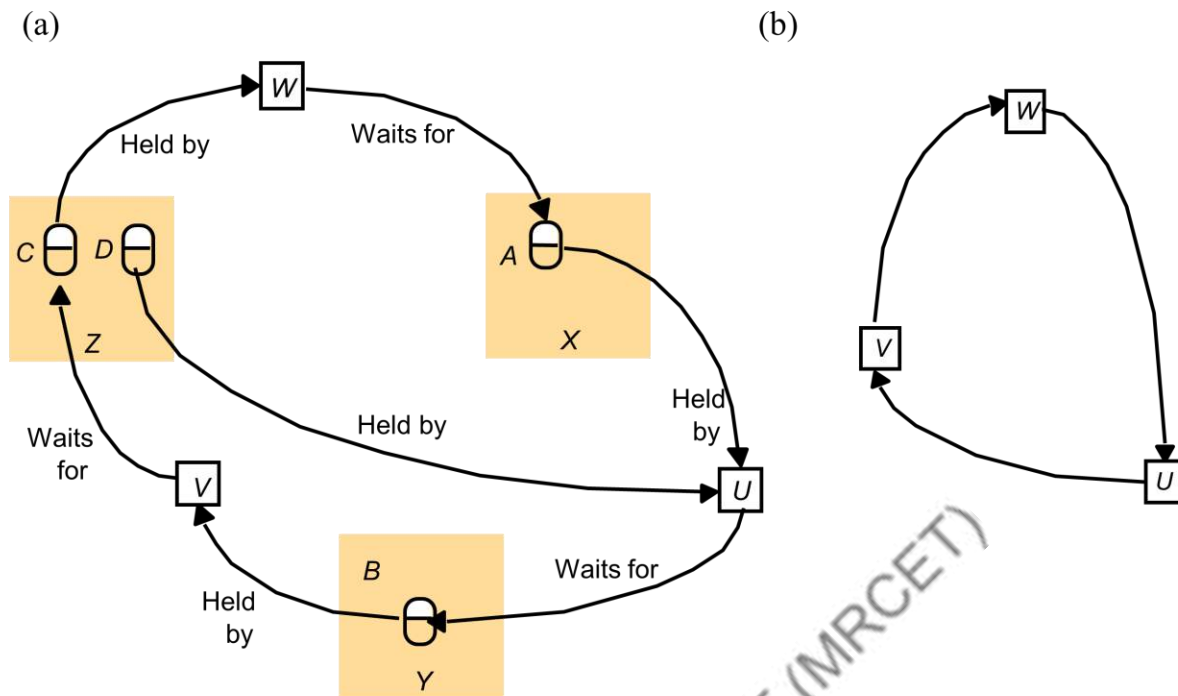
Conflicts are resolved as each operation is performed. If the resolution of a conflict requires a transaction to be aborted, the coordinator will be informed and it will abort the transaction at all the participants.

Distributed Deadlock

Deadlocks can arise within a single server when locking is used for concurrency control. Servers must either prevent or detect and resolve deadlocks.

Using timeout to resolve deadlock is a clumsy approach. Why? Another way is to detect deadlock by detecting cycles in a wait for graph

Figure 14.14
Distributed deadlock

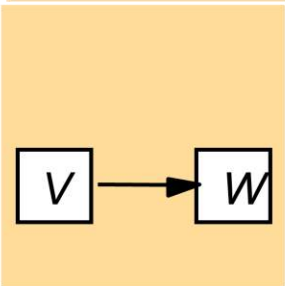
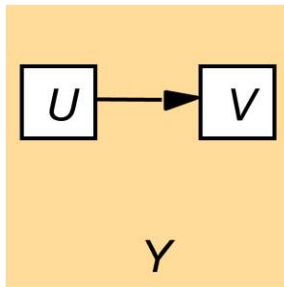
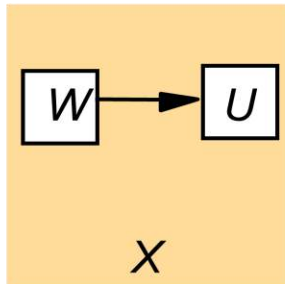


Instructor's Guide for Coulouris, Dollimore and Kindberg: Distributed Systems: Concepts and Design Edn.4
© Pearson Education 2005

- z A deadlock that is detected but is not really a deadlock is called a phantom deadlock.
- z As the procedure of sending local wait-for graph to one place will take some time, there is a chance that one of the transactions that holds a lock will meanwhile have released it, in which case the deadlock will no longer exist.

Figure 14.14
Local and global wait-for graphs

local wait-for graph



Global wait for graph is held in part by each of the several servers involved. Communication between these servers is required to find cycles in the graph.

Simple solution: one server takes on the role of global deadlock detector. From time to time, each server sends the latest copy of its local wait-for graph.

Disadvantages: poor availability, lack of fault tolerance and no ability to scale. The cost of frequent transmission of local wait-for graph is high.

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn.4
© Pearson Education 2005

At about the same time, T waits for U ($T \rightarrow U$) and W waits for V ($W \rightarrow V$). Two probes occur, two deadlocks detected by different servers.

We want to ensure that only one transaction is aborted in the same deadlock since different servers may choose different transaction to abort leading to unnecessary abort of transactions.

So using priorities to determine which transaction to abort will result in the same transaction to abort even if the cycles are detected by different servers.

Using priority can also reduce the number of probes. For example, we only initiate probe when higher priority transaction starts to wait for lower priority transaction.

If we say the priority order from high to low is: T, U, V and W. Then only the probe of $T \rightarrow U$ will be sent and not the probe of $W \rightarrow V$.

Transaction recovery

Atomic property of transactions can be described in two aspects:

Durability: objects are saved in permanent storage and will be available indefinitely thereafter. Acknowledgement of a client's commit request implies that all the effects of the transaction have been recorded in permanent storage as well as in the server's volatile object.

Failure atomicity: the effects of transactions are atomic even when the server crashes.

Both can be realized by recovery manager.

Recovery manager

Tasks of a recovery manager:

Save objects in permanent storage (in a recovery file) for committed transactions;

To restore the server's objects after a crash;

To reorganize the recovery file to improve the performance of recovery;

To reclaim storage space in the recovery file.

Figure 14.18
Types of entry in a recovery file

Type of entry	Description of contents of entry
Object	A value of an object.
Transaction status	Transaction identifier, transaction status (<i>prepared</i> , <i>committed</i> , <i>aborted</i>) and other status values used for the two-phase commit protocol.
Intentions list	Transaction identifier and a sequence of intentions, each of which consists of <identifier of object>, < Position of value of object>.

Intention list records all of its currently active transactions. A list of a particular transaction contains a list of the references and the values of all the objects that are altered. When committed, the committed version of each object is replaced by the tentative version made by that transaction. When a transaction aborts, the server uses the intention list to delete all the tentative versions of objects.

When a participant says it is prepared to commit, its recovery manager must have saved both its intention list for that transaction and the objects in that intention list in its recovery file, so it will be able to carry out the commitment later on, even if it crashes in the interim.

Instructor's Guide for Coulouris, Dollimore and Kindberg Distributed Systems: Concepts and Design Edn.4
© Pearson Education 2005

Log with entries relating to two-phase commit protocol

In phase 1, when the coordinator is prepared to commit and has already added a prepared status entry, its recovery manager adds a coordinator entry. Before a participant can vote Yes, it must have already prepared to commit and must have already added a prepared status entry. When it votes Yes, its recovery manager records a participant entry and adds an uncertain status. When a participant votes No, it adds an abort status to recovery file.

In phase 2, the recovery manager of the coordinator adds either a committed or an aborted, according to the decision. Recovery manager of participants add a commit or abort status to their recovery files according to

message received from coordinator. When a coordinator has received a confirmation from all its participants, its recovery manager adds a done status.

When a server is replaced after a crash, the recovery manager has to deal with the two-phase commit protocol in addition to restore the objects.

For any transaction where the server has played the coordinator role, it should find a coordinator entry and a set of transaction status entries. For any transaction where the server has played the participant role, it should find a participant entry and a set of set of transaction status entries. In both cases, the most recent transaction status entry, that is the one nearest the end of log determine the status at the time of failure.

The action of the recovery manager with respect to the two-phase commit protocol for any transaction depends on whether the server was the coordinator or a participant and on its status at the time of failure as shown in the following table.

Figure 14.22
Recovery of the two-phase commit protocol

<i>Role</i>	<i>Status</i>	<i>Action of recovery manager</i>
Coordinator	<i>prepared</i>	No decision had been reached before the server failed. It sends <i>abortTransaction</i> to all the servers in the participant list and adds the transaction status <i>aborted</i> in its recovery file. Same action for state <i>aborted</i> . If there is no participant list, the participants will eventually timeout and abort the transaction.
Coordinator	<i>committed</i>	A decision to commit had been reached before the server failed. It sends a <i>doCommit</i> to all the participants in its participant list (in case it had not done so before) and resumes the two-phase protocol at step 4 (Fig 13.5).
Participant	<i>committed</i>	The participant sends a <i>haveCommitted</i> message to the coordinator (in case this was not done before it failed). This will allow the coordinator to discard information about this transaction at the next checkpoint.
Participant	<i>uncertain</i>	The participant failed before it knew the outcome of the transaction. It cannot determine the status of the transaction until the coordinator informs it of the decision. It will send a <i>getDecision</i> to the coordinator to determine the status of the transaction. When it receives the reply it will commit or abort accordingly.
Participant	<i>prepared</i>	The participant has not yet voted and can abort the transaction.
Coordinator	<i>done</i>	No action is required.

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

III Year B. Tech. CSE –II Semester

(R15A0518)

OBJECT ORIENTED ANALYSIS AND DESIGN

TEXT BOOKS:

1. Grady Booch, James Rumbaugh, Ivar Jacobson: The Unified Modeling Language User Guide, Pearson Education.

REFERENCES :

1. Grady Booch, James Rumbaugh and Ivar Jacobson, “The Unified Modeling Languages User Guide”, Addison Wesley, 2004.

UNIT I

Introduction to UML

Syllabus : Importance of modeling, principles of modeling, object oriented modeling, conceptual model of the UML, Architecture, and Software Development Life Cycle.

Object -oriented modeling languages appeared sometime between the mid 1970s and the late 1980s as methodologists, faced with a new genre of object-oriented programming languages and increasingly complex applications, began to experiment with alternative approaches to analysis and design. The number of object-oriented methods increased from fewer than 10 to more than 50 during the period between 1989 and 1994. Many Importance of modeling, principles of modeling, object oriented modeling, conceptual model of the UML, Architecture, and Software Development Life Cycle.

Brief History of the UML

users of these methods had trouble finding a modeling language that met their needs completely, thus fueling the so- called method wars. Learning from experience, new generations of these methods began to appear, with a few clearly prominent methods emerging, most notably Booch, Jacobson's OOSE (Object-Oriented Software Engineering), and Rumbaugh's OMT (Object Modeling Technique). Other important methods included Fusion, Shlaer-Mellor, and Coad-Yourdon. Each of these was a complete method, although each was recognized as having strengths and weaknesses. In simple terms, the Booch method was particularly expressive during the design and construction phases of projects, OOSE provided excellent support for use cases as a way to drive requirements capture, analysis, and high-level design, and OMT-2 was most useful for analysis and data-intensive information systems. The behavioral component of many object-oriented methods, including the Booch method and OMT, was the language of statecharts, invented by David Harel. Prior to this object-oriented adoption, statecharts were used mainly in the realm of functional decomposition and structured analysis, and led to the development of executable models and tools that generated full running code.

A critical mass of ideas started to form by the mid 1990s, when Grady Booch (Rational Software Corporation), Ivar Jacobson (Objectory), and James Rumbaugh (General Electric) began to adopt ideas from each other's methods, which collectively were becoming recognized as the leading object-oriented methods worldwide. As the primary authors of the Booch, OOSE, and OMT methods, we were motivated to create a unified modeling language for three reasons. First, our methods were already evolving toward each other independently. It made sense to continue that evolution together rather than apart, eliminating the potential for any unnecessary and gratuitous differences that would further confuse users. Second, by unifying our methods, we could bring some stability to the object-oriented marketplace, allowing projects to settle on one mature modeling language and letting tool builders focus on delivering more useful features. Third, we expected that our collaboration would yield improvements for all three earlier methods, helping us to capture lessons learned and to address problems that none of our methods previously handled well.

As we began our unification, we established three goals for our work:

1. To model systems, from concept to executable artifact, using object- oriented techniques
2. To address the issues of scale inherent in complex, mission-critical systems
3. To create a modeling language usable by both humans and machines

Devising a language for use in object- oriented analysis and design is not unlike designing a programming language. First, we had to constrain the problem: Should the language encompass requirements specification? Should the language be sufficient to permit visual programming? Second, we had to strike a

balance between expressiveness and simplicity. Too simple a language would limit the breadth of problems that could be solved; too complex a language would overwhelm the mortal developer. In the case of unifying existing methods, we also had to be sensitive to the installed base. Make too many changes, and we would confuse existing users; resist advancing the language, and we would miss the opportunity of engaging a much broader set of users and of making the language simpler. The UML definition strives to make the best trade-offs in each of these areas.

The UML effort started officially in October 1994, when Rumbaugh joined Booch at Rational. Our project's initial focus was the unification of the Booch and OMT methods. The version 0.8 draft of the Unified Method (as it was then called) was released in October 1995. Around the same time, Jacobson joined Rational and the scope of the UML project was expanded to incorporate OOSE. Our efforts resulted in the release of the UML version 0.9 documents in June 1996. Throughout 1996, we invited and received feedback from the general software engineering community. During this time, it also became clear that many software organizations saw the UML as strategic to their business. We established a UML consortium, with several organizations willing to dedicate resources to work toward a strong and complete UML definition. Those partners contributing to the UML 1.0 definition included Digital Equipment Corporation, Hewlett-Packard, I-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational, Texas Instruments, and Unisys. This collaboration resulted in the UML 1.0, a modeling language that was well-defined, expressive, powerful, and applicable to a wide spectrum of problem domains. UML 1.0 was offered for standardization to the Object Management Group (OMG) in January 1997, in response to their request for proposal for a standard modeling language.

Between January 1997 and July 1997, the original group of partners was expanded to include virtually all of the other submitters and contributors of the original OMG response, including Andersen Consulting, Ericsson, ObjecTime Limited, Platinum Technology, PTech, Reich Technologies, Softeam, Sterling Software, and Taskon. A semantics task force was formed, led by Cris Kobryn of MCI Systemhouse and administered by Ed Eykholt of Rational, to formalize the UML specification and to integrate the UML with other standardization efforts. A revised version of the UML (version 1.1) was offered to the OMG for standardization in July 1997. In September 1997, this version was accepted by the OMG Analysis and Design Task Force (ADTF) and the OMG Architecture Board and then put up for vote by the entire OMG membership. UML 1.1 was adopted by the OMG on November 14, 1997.

A successful software organization is one that consistently deploys quality software that meets the needs of its users. An organization that can develop such software in a timely and predictable fashion, with an efficient and effective use of resources, both human and material, is one that has a sustainable business.

There's an important implication in this message: The primary product of a development team is not beautiful documents, world-class meetings, great slogans, or Pulitzer prize—winning lines of source code. Rather, it is good software that satisfies the evolving needs of its users and the business. Everything else is secondary.

Unfortunately, many software organizations confuse "secondary" with "irrelevant." To deploy software that satisfies its intended purpose, you have to meet and engage users in a disciplined fashion, to expose the real requirements of your system. To develop software of lasting quality, you have to craft a solid architectural foundation that's resilient to change. To develop software rapidly, efficiently, and effectively, with a minimum of software scrap and rework, you have to have the right people, the right tools, and the right focus. To do all this consistently and predictably, with an appreciation for the lifetime costs of the system, you must have a sound development process that can adapt to the changing needs of your business and technology.

Modeling is a central part of all the activities that lead up to the deployment of good software. We build models to communicate the desired structure and behavior of our system. We build models to visualize and control the system's architecture. We build models to better understand the system we are building, often exposing opportunities for simplification and reuse. We build models to manage risk.

The Importance of Modeling

Unsuccessful software projects fail in their own unique ways, but all successful projects are alike in many ways. There are many elements that contribute to a successful software organization; one common thread is the use of modeling.

Modeling is a proven and well-accepted engineering technique. We build architectural models of houses and high rises to help their users visualize the final product. We may even build mathematical models in order to analyze the effects of winds or earthquakes on our buildings.

Modeling is not just a part of the building industry. It would be inconceivable to deploy a new aircraft or **an automobile without first building models• from computer models to physical wind tunnel models to** full-scale prototypes. New electrical devices, from microprocessors to telephone switching systems require some degree of modeling in order to better understand the system and to communicate those ideas to others. In the motion picture industry, storyboarding, which is a form of modeling, is central to any production. In the fields of sociology, economics, and business management, we build models so that we can validate our theories or try out new ones with minimal risk and cost.

What, then, is a model? Simply put,

A model is a simplification of reality.

A model provides the blueprints of a system. Models may encompass detailed plans, as well as more general plans that give a 30,000-foot view of the system under consideration.

Why do we model? There is one fundamental reason.

We build models so that we can better understand the system we are developing.

Through modeling, we achieve four aims.

1. Models help us to visualize a system as it is or as we want it to be.
2. Models permit us to specify the structure or behavior of a system.
3. Models give us a template that guides us in constructing a system.
4. Models document the decisions we have made.

Modeling is not just for big systems. Even the software equivalent of a dog house can benefit from some modeling. However, it's definitely true that the larger and more complex the system, the more important modeling becomes, for one very simple reason:

We build models of complex systems because we cannot comprehend such a system in its entirety.

Principles of Modeling

The use of modeling has a rich history in all the engineering disciplines. That experience suggests four basic principles of modeling. First,

The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.

In other words, choose your models well. The right models will brilliantly illuminate the most wicked development problems, offering insight that you simply could not gain otherwise; the wrong models will mislead you, causing you to focus on irrelevant issues.

Second,

Every model may be expressed at different levels of precision.

If you are building a high rise, sometimes you need a 30,000- foot **view• for instance, to help** your investors visualize its look and feel. Other times, you **need to get down to the level of the studs•** for instance, when there's a tricky pipe run or an unusual structural element.

Third,

The best models are connected to reality.

A physical model of a building that doesn't respond in the same way as do real materials has only limited value; a mathematical model of an aircraft that assumes only ideal conditions and perfect manufacturing can mask some potentially fatal characteristics of the real aircraft. It's best to have models that have a clear connection to reality, and where that connection is weak, to know exactly how those models are divorced from the real world. All models simplify reality; the trick is to be sure that your simplifications don't mask any important details.

In software, the Achilles heel of structured analysis techniques is the fact that there is a basic disconnect between its analysis model and the system's design model. Failing to bridge this chasm causes the system as conceived and the system as built to diverge over time. In object-oriented systems, it is possible to connect all the nearly independent views of a system into one semantic whole.

Fourth,

No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

If you are constructing a building, there is no single set of blueprints that reveal all its details. At the very least, you'll need floor plans, elevations, electrical plans, heating plans, and plumbing plans.

Object-Oriented Modeling

In software, there are several ways to approach a model. The two most common ways are from an algorithmic perspective and from an object-oriented perspective.

The traditional view of software development takes an algorithmic perspective. In this approach, the main building block of all software is the procedure or function. This view leads developers to focus on issues of control and the decomposition of larger algorithms into smaller ones. There's nothing inherently evil about such a point of view except that it tends to yield brittle systems. As requirements change (and they will) and the system grows (and it will), systems built with an algorithmic focus turn out to be very hard to maintain.

The contemporary view of software development takes an object-oriented perspective. In this approach, the main building block of all software systems is the object or class. Simply put, an object is a thing, generally drawn from the vocabulary of the problem space or the solution space; a class is a description of a set of common objects. Every object has identity (you can name it or otherwise distinguish it from other objects), state (there's generally some data associated with it), and behavior (you can do things to the object, and it can do things to other objects, as well).

The Unified Modeling Language (UML) is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system.

The UML is appropriate for modeling systems ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems. It is a very expressive language, addressing all the views needed to develop and then deploy such systems. Even though it is expressive, the UML is not difficult to understand and to use. Learning to apply the UML effectively starts with forming a conceptual model of the language, which requires learning three major elements: the UML's basic building blocks, the rules that dictate how these building blocks may be put together, and some common mechanisms that apply throughout the language.

The UML is only a language and so is just one part of a software development method. The UML is process independent, although optimally it should be used in a process that is use case driven, architecture-centric, iterative, and incremental.

An Overview of the UML

The UML is a language for

- Visualizing
- Specifying
- Constructing
- Documenting

the artifacts of a software-intensive system.

The UML Is a Language

A language provides a vocabulary and the rules for combining words in that vocabulary for the purpose of communication. A *modeling* language is a language whose vocabulary and rules focus on the conceptual and physical representation of a system. A modeling language such as the UML is thus a standard language for software blueprints.

The UML Is a Language for Visualizing

For many programmers, the distance between thinking of an implementation and then pounding it out in code is close to zero. You think it, you code it. In fact, some things are best cast directly in code. Text is a wonderfully minimal and direct way to write expressions and algorithms.

information would be lost forever or, at best, only partially recreatable from the implementation, once that developer moved on. Writing models in the UML addresses the third issue: An explicit model facilitates communication.

The UML Is a Language for Specifying

In this context, *specifying* means building models that are precise, unambiguous, and complete. In particular, the UML addresses the specification of all the important analysis, design, and implementation decisions that must be made in developing and deploying a software-intensive system.

The UML Is a Language for Constructing

The UML is not a visual programming language, but its models can be directly connected to a variety of programming languages. This means that it is possible to map from a model in the UML to a programming language such as Java, C++, or Visual Basic, or even to tables in a relational database or the persistent store of an object-oriented database

The UML Is a Language for Documenting

A healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include (but are not limited to)

- Requirements
- Architecture
- Design
- Source code
- Project plans
- Tests
- Prototypes
- Releases

Where Can the UML Be Used?

The UML is intended primarily for software-intensive systems. It has been used effectively for such domains as

- Enterprise information systems
- Banking and financial services
- Telecommunications
- Transportation
- Defense/aerospace
- Retail
- Medical electronics
- Scientific
- Distributed Web-based services

The UML is not limited to modeling software. In fact, it is expressive enough to model nonsoftware systems, such as workflow in the legal system, the structure and behavior of a patient healthcare system, and the design of hardware.

A Conceptual Model of the UML

To understand the UML, you need to form a conceptual model of the language, and this requires learning three major elements: the UML's basic building blocks, the rules that dictate how those building blocks may be put together, and some common mechanisms that apply throughout the UML.

Building Blocks of the UML

The vocabulary of the UML encompasses three kinds of building blocks:

1. Things

2. Relationships
3. Diagrams

Things are the abstractions that are first-class citizens in a model; relationships tie these things together; diagrams group interesting collections of things.

Things in the UML

There are four kinds of things in the UML:

1. Structural things
2. Behavioral things
3. Grouping things
4. Annotational things

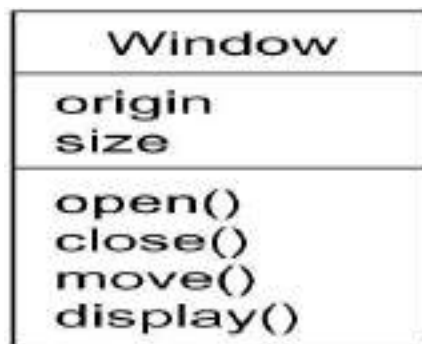
These things are the basic object-oriented building blocks of the UML. You use them to write well-formed models.

Structural Things

Structural things are the nouns of UML models. These are the mostly static parts of a model, representing elements that are either conceptual or physical. In all, there are seven kinds of structural things.

First, a *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces. Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations, as in [Figure](#)

Figure Classes



Second, an *interface* is a collection of operations that specify a service of a class or component. An interface therefore describes the externally visible behavior of that element. An interface might represent the complete behavior of a class or component or only a part of that behavior. An interface defines a set of operation specifications (that is, their signatures) but never a set of operation implementations. Graphically, an interface is rendered as a circle together with its name. An interface rarely stands alone. Rather, it is typically attached to the class or component that realizes the interface, as in [Figure](#).

Figure Interfaces



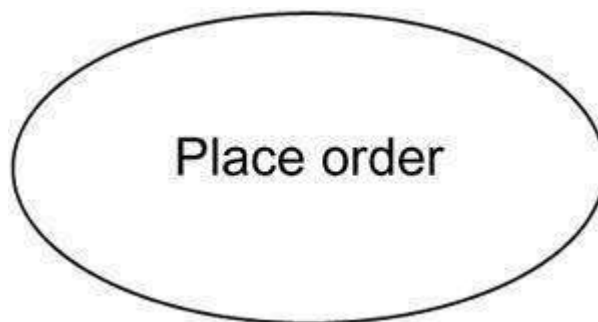
Third, a *collaboration* defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class might participate in several collaborations. These collaborations therefore represent the implementation of patterns that make up a system. Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name, as in [Figure](#).

Figure Collaborations



Fourth, a *use case* is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor. A use case is used to structure the behavioral things in a model. A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse with solid lines, usually including only its name, as in [Figure](#).

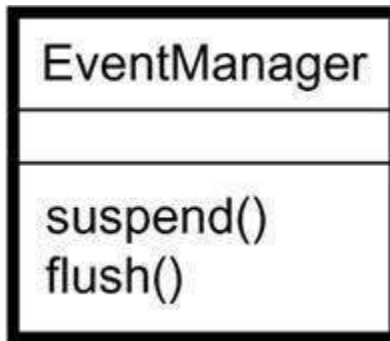
Figure Use Cases



Fifth, an *active class* is a class whose objects own one or more processes or threads and therefore can initiate control activity. An active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but

with heavy lines, usually including its name, attributes, and operations, as in [Figure](#).

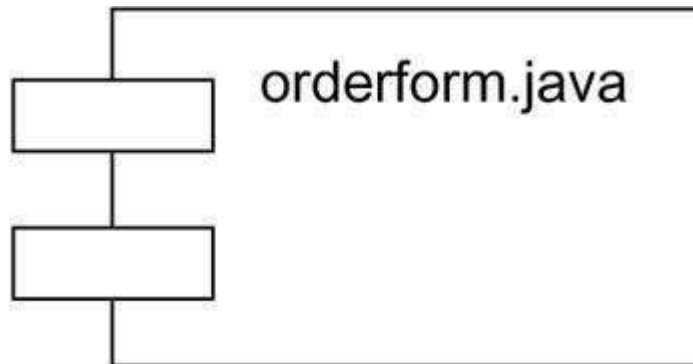
Figure Active Classes



The remaining two elements—component, and nodes—are also different. They represent **physical** things, whereas the previous five things represent conceptual or logical things.

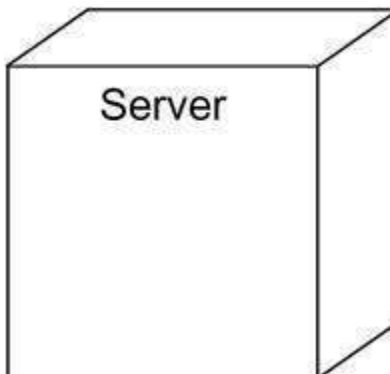
Sixth, a *component* is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. In a system, you'll encounter different kinds of deployment components, such as COM+ components or Java Beans, as well as components that are artifacts of the development process, such as source code files. A component typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations. Graphically, a component is rendered as a rectangle with tabs, usually including only its name, as in [Figure](#).

Figure Components



Seventh, a *node* is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. A set of components may reside on a node and may also migrate from node to node. Graphically, a node is rendered as a cube, usually including only its name, as in [Figure](#).

Figure Nodes



These seven elements• classes, interfaces, collaborations, use cases, active classes, components, and nodes• are the basic structural things that you may include in a UML model. There are also variations on these seven, such as actors, signals, and utilities (kinds of classes), processes and threads (kinds of active classes), and applications, documents, files, libraries, pages, and tables (kinds of components).

Behavioral Things

Behavioral things are the dynamic parts of UML models. These are the verbs of a model, representing behavior over time and space. In all, there are two primary kinds of behavioral things.

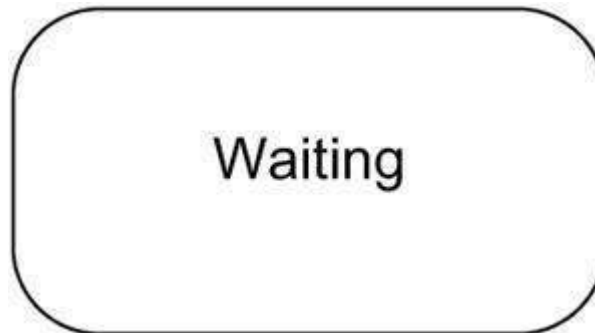
First, an *interaction* is a behavior that comprises a set of messages exchanged among a set of objects within a particular context to accomplish a specific purpose. The behavior of a society of objects or of an individual operation may be specified with an interaction. An interaction involves a number of other elements, including messages, action sequences (the behavior invoked by a message), and links (the connection between objects). Graphically, a message is rendered as a directed line, almost always including the name of its operation, as in [Figure](#).

Figure Messages



Second, a *state machine* is a behavior that specifies the sequences of states an object or an interaction goes through during its lifetime in response to events, together with its responses to those events. The behavior of an individual class or a collaboration of classes may be specified with a state machine. A state machine involves a number of other elements, including states, transitions (the flow from state to state), events (things that trigger a transition), and activities (the response to a transition). Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates, if any, as in [Figure](#).

Figure States



These two elements• interactions and state machines• are the basic behavioral things that you may include in a UML model. Semantically, these elements are usually connected to various structural elements, primarily classes, collaborations, and objects.

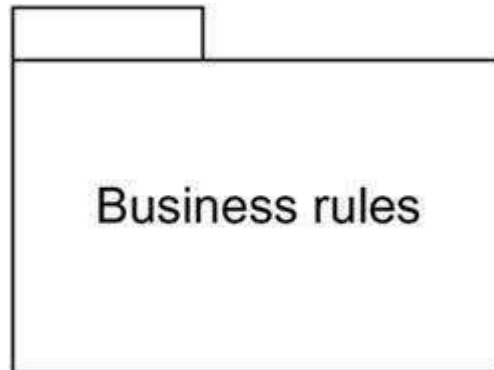
Grouping Things

Grouping things are the organizational parts of UML models. These are the boxes into which a model can be decomposed. In all, there is one primary kind of grouping thing, namely, packages.

A *package* is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package. Unlike components (which exist at run time), a package is purely conceptual (meaning that it exists only at development time).

Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents, as in [Figure](#).

Figure Packages

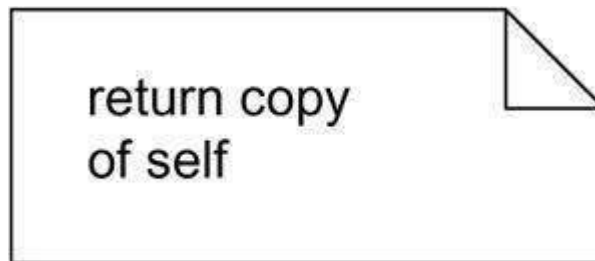


Packages are the basic grouping things with which you may organize a UML model. There are also variations, such as frameworks, models, and subsystems (kinds of packages).

Annotational Things

Annotational things are the explanatory parts of UML models. These are the comments you may apply to describe, illuminate, and remark about any element in a model. There is one primary kind of annotational thing, called a note. A *note* is simply a symbol for rendering constraints and comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment, as in [Figure](#).

Figure Notes



Relationships in the UML

There are four kinds of relationships in the UML:

1. Dependency
2. Association
3. Generalization
4. Realization

These relationships are the basic relational building blocks of the UML. You use them to write well-formed models.

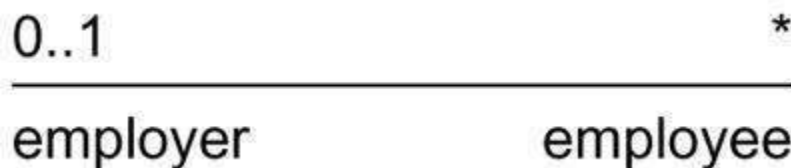
First, a *dependency* is a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label, as in [Figure](#).

Figure Dependencies



Second, an *association* is a structural relationship that describes a set of links, a link being a connection among objects. Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names, as in [Figure](#).

Figure Associations



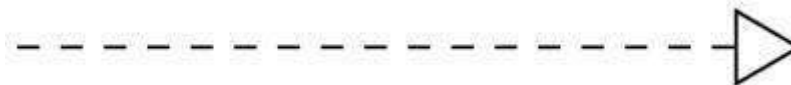
Third, a *generalization* is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In this way, the child shares the structure and the behavior of the parent. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent, as in [Figure](#).

Figure Generalizations



Fourth, a *realization* is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. You'll encounter realization relationships in two places: between interfaces and the classes or components that realize them, and between use cases and the collaborations that realize them. Graphically, a realization relationship is rendered as a cross between a generalization and a dependency relationship, as in [Figure](#).

Figure Realization



These four elements are the basic relational things you may include in a UML model. There are also variations on these four, such as refinement, trace, include, and extend (for dependencies). *The five views of an architecture are discussed in the following section.*

Diagrams in the UML

A *diagram* is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships). the UML includes nine such diagrams:

1. Class diagram
2. Object diagram

3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

A *class diagram* shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modeling object- oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system.

An *object diagram* shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams, but from the perspective of real or prototypical cases.

A *use case diagram* shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system. These diagrams are especially important in organizing and modeling the behaviors of a system.

Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams. An shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. Interaction diagrams address the dynamic view of a system. A *sequence diagram* is an interaction diagram that emphasizes the time-ordering of messages;

A *collaboration diagram* is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other.

A *statechart diagram* shows a state machine, consisting of states, transitions, events, and activities. Statechart diagrams address the dynamic view of a system. They are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

An *activity diagram* is a special kind of a statechart diagram that shows the flow from activity to activity within a system. Activity diagrams address the dynamic view of a system. They are especially important in modeling the function of a system and emphasize the flow of control among objects.

A *component diagram* shows the organizations and dependencies among a set of components. Component diagrams address the static implementation view of a system. They are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

A *deployment diagram* shows the configuration of run-time processing nodes and the components that live

on them. Deployment diagrams address the static deployment view of an architecture. They are related to component diagrams in that a node typically encloses one or more components.

Rules of the UML

The UML's building blocks can't simply be thrown together in a random fashion. Like any language, the UML has a number of rules that specify what a well-formed model should look like. A *well-formed model* is one that is semantically self-consistent and in harmony with all its related models.

The UML has semantic rules for

Names	What you can call things, relationships, and diagrams
Scope	The context that gives specific meaning to a name
Visibility	How those names can be seen and used by others
Integrity	How things properly and consistently relate to one another
Execution	What it means to run or simulate a dynamic model

Models built during the development of a software-intensive system tend to evolve and may be viewed by many stakeholders in different ways and at different times. For this reason, it is common for the development team to not only build models that are well-formed, but also to build models that are elided, incomplete, inconsistent.

Common Mechanisms in the UML

It is made simpler by the presence of four common mechanisms that apply consistently throughout the language.

1. Specifications
2. Adornments
3. Common divisions
4. Extensibility mechanisms

Specifications

The UML's specifications provide a semantic backplane that contains all the parts of all the models of a system, each part related to one another in a consistent fashion. The UML's diagrams are thus simply visual projections into that backplane, each diagram revealing a specific interesting aspect of the system.

Adornments

Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element. For example, the notation for a class is intentionally designed to be easy to draw, because classes are the most common element found in modeling object-oriented systems. The class notation also exposes the most important aspects of a class, namely its name, attributes, and operations.

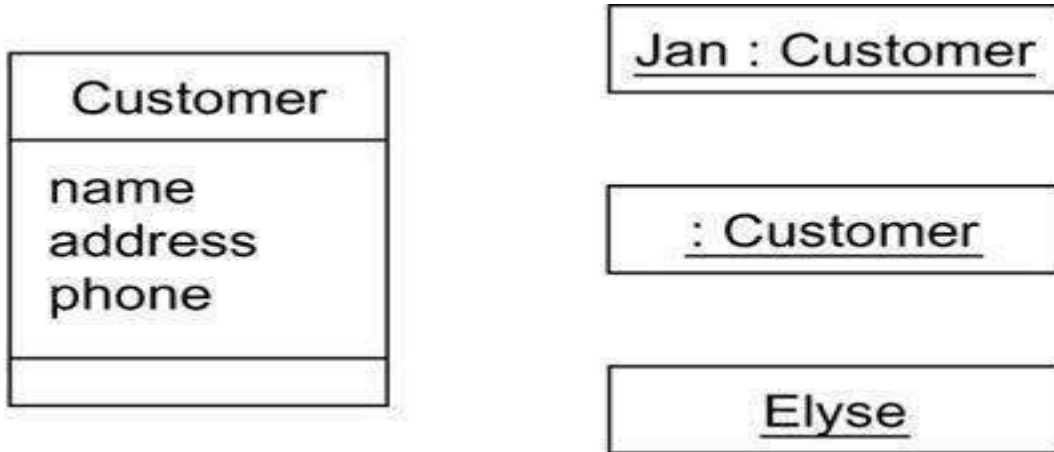
Every element in the UML's notation starts with a basic symbol, to which can be added a variety of adornments specific to that symbol.

Common Divisions

In modeling object-oriented systems, the world often gets divided in at least a couple of ways.

First, there is the division of class and object. A class is an abstraction; an object is one concrete manifestation of that abstraction. In the UML, you can model classes as well as objects, as shown in [Figure](#).

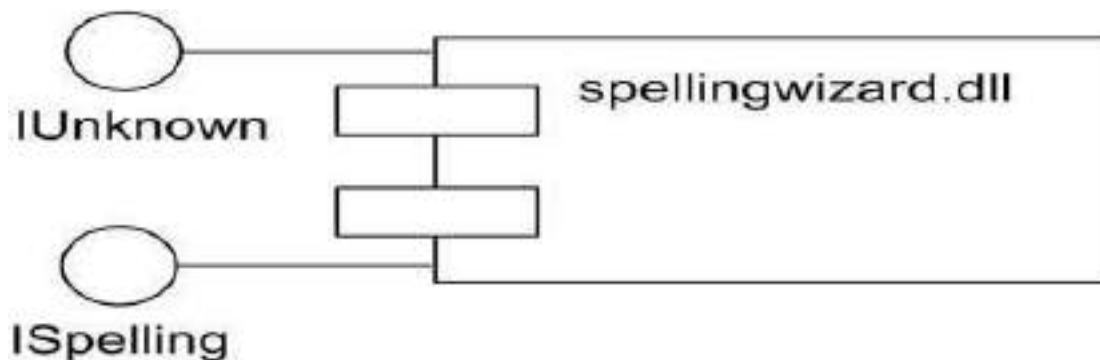
Figure Classes And Objects



In this figure, there is one class, named **Customer**, together with three objects: **Jan** (which is marked explicitly as being a **Customer** object), **:Customer** (an anonymous **Customer** object), and **Elyse** (which in its specification is marked as being a kind of **Customer** object, although it's not shown explicitly here).

Second, there is the separation of interface and implementation. An interface declares a contract, and an implementation represents one concrete realization of that contract, responsible for faithfully carrying out the interface's complete semantics. In the UML, you can model both interfaces and their implementations, as shown in [Figure](#).

Figure Interfaces And Implementations



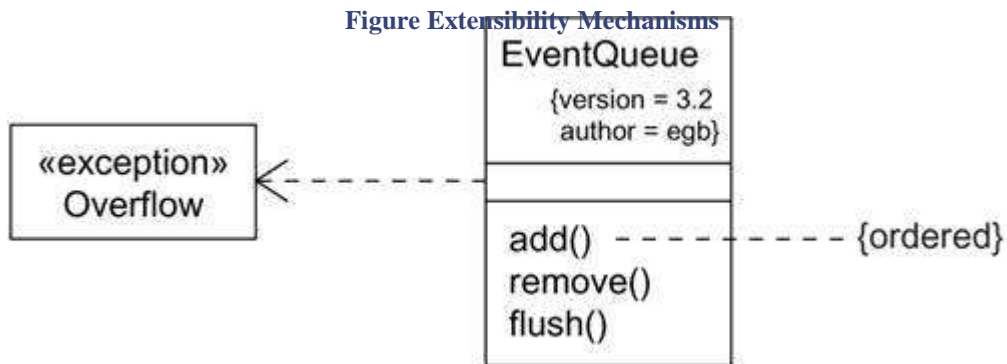
In this figure, there is one component named **spellingwizard.dll** that implements two interfaces, **IUnknown** and **ISpelling**. Almost every building block in the UML has this same kind of interface/implementation dichotomy. For example, you can have use cases and the collaborations that realize them, as well as operations and the methods that implement them.

Extensibility Mechanisms

The UML's extensibility mechanisms include

- Stereotypes
- Tagged values
- Constraints

A *stereotype* extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but that are specific to your problem. For example, if you are working in a programming language, such as Java or C++, you will often want to model exceptions. In these languages, exceptions are just classes, although they are treated in very special ways. Typically, you only want to allow them to be thrown and caught, nothing else. You can make exceptions first class citizens in your **models** meaning that they are treated like basic building blocks by marking them with an appropriate stereotype, as for the class **Overflow** in [Figure](#).



A *tagged value* extends the properties of a UML building block, allowing you to create new information in that element's specification. For example, if you are working on a shrink-wrapped product that undergoes many releases over time, you often want to track the version and author of certain critical abstractions. Version and author are not primitive UML concepts. They can be added to any building block, such as a class, by introducing new tagged values to that building block. In [Figure](#), for example, the class **EventQueue** is extended by marking its version and author explicitly.

A *constraint* extends the semantics of a UML building block, allowing you to add new rules or modify existing ones. For example, you might want to constrain the **EventQueue** class so that all additions are done in order. As [Figure](#) shows, you can add a constraint that explicitly marks these for the operation **add**.

Architecture

A system's architecture is perhaps the most important artifact that can be used to manage these different viewpoints and so control the iterative and incremental development of a system throughout its life cycle.

Architecture is the set of significant decisions about

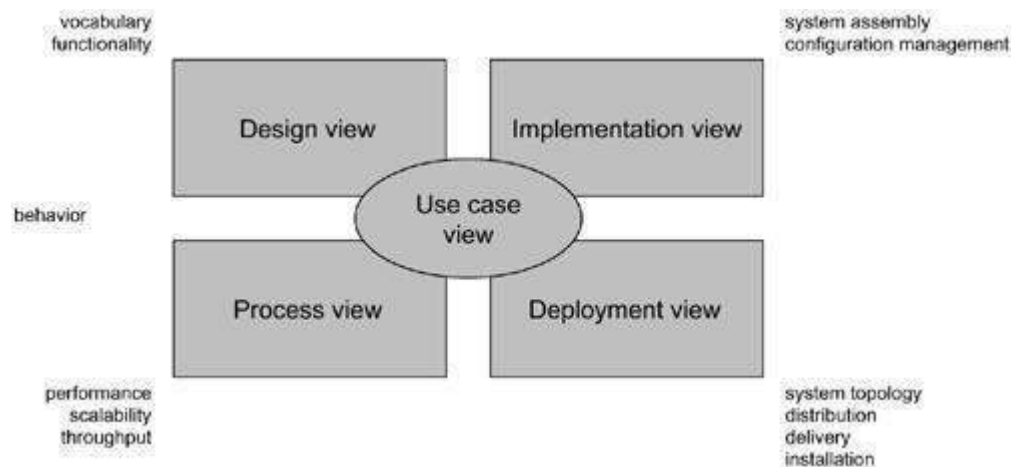
- The organization of a software system

- The selection of the structural elements and their interfaces by which the system is composed
- Their behavior, as specified in the collaborations among those elements
- The composition of these structural and behavioral elements into progressively larger subsystems
- The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations, and their composition

Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns.

As Figure illustrates, the architecture of a software-intensive system can best be described by five interlocking views. Each view is a projection into the organization and structure of the system, focused on a particular aspect of that system.

Figure Modeling a System's Architecture



The *use case view* of a system encompasses the use cases that describe the behavior of the system as seen by its end users, analysts, and testers. This view doesn't really specify the organization of a software system. Rather, it exists to specify the forces that shape the system's architecture. With the UML, the static aspects of this view are captured in use case diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

The *design view* of a system encompasses the classes, interfaces, and collaborations that form the vocabulary of the problem and its solution. This view primarily supports the functional requirements of the system, meaning the services that the system should provide to its end users. With the UML, the static aspects of this view are captured in class diagrams and object diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

The *process view* of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability, and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as for the design view, but with a focus on the active classes that represent these threads and processes.

The *implementation view* of a system encompasses the components and files that are used to assemble and release the physical system. This view primarily addresses the configuration management of the system's releases, made up of somewhat independent components and files that can be assembled in various ways to produce a running system. With the UML, the static aspects of this view are captured in component diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

The *deployment view* of a system encompasses the nodes that form the system's hardware topology on which the system executes. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system. With the UML, the static aspects of this view are captured in deployment diagrams; the dynamic aspects of this view are captured in interaction diagrams, statechart diagrams, and activity diagrams.

Each of these five views can stand alone so that different stakeholders can focus on the issues of the system's architecture that most concern them. These five views also interact with **one another**• nodes in the deployment view hold components in the implementation view that, in turn, represent the physical realization of classes, interfaces, collaborations, and active classes from the design and process views. The UML permits you to express every one of these five views and their interactions.

Software Development Life Cycle

The UML is largely process-independent, meaning that it is not tied to any particular software development life cycle. However, to get the most benefit from the UML, you should consider a process that is

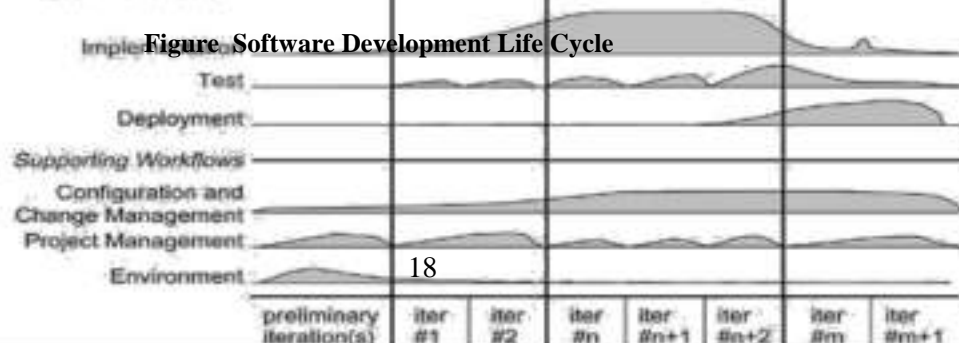
- Use case driven
- Architecture-centric
- Iterative and incremental

Use case driven means that use cases are used as a primary artifact for establishing the desired behavior of the system, for verifying and validating the system's architecture, for testing, and for communicating among the stakeholders of the project.

Architecture-centric means that a system's architecture is used as a primary artifact for conceptualizing, constructing, managing, and evolving the system under development.

An *iterative process* is one that involves managing a stream of executable releases. An is one that involves the continuous integration of the system's architecture to produce these releases, with each new release embodying incremental improvements over the other. Together, an iterative and incremental process is *risk-driven*, meaning that each new release is focused on attacking and reducing the most significant risks to the success of the project.

This use case driven, architecture-centric, and iterative/incremental process can be broken into phases. A *phase* is the span of time between two major milestones of the process, when a well-defined set of objectives are met, artifacts are completed, and decisions are made whether to move into the next phase. As Figure shows, there are four phases in the software development life cycle: inception, elaboration, construction, and transition. In the diagram, workflows are plotted against these phases, showing their varying degrees of focus over time.



Inception is the first phase of the process, when the seed idea for the development is brought up to the **point of being• at least internally• sufficiently well-founded** to warrant entering into the elaboration phase.

Elaboration is the second phase of the process, when the product vision and its architecture are defined. In this phase, the system's requirements are articulated, prioritized, and baselined. A system's requirements may range from general vision statements to precise evaluation criteria, each specifying particular functional or nonfunctional behavior and each providing a basis for testing.

Construction is the third phase of the process, when the software is brought from an executable architectural baseline to being ready to be transitioned to the user community. Here also, the system's requirements and especially its evaluation criteria are constantly reexamined against the business needs of the project, and resources are allocated as appropriate to actively attack risks to the project.

Transition is the fourth phase of the process, when the software is turned into the hands of the user community. Rarely does the software development process end here, for even during this phase, the system is continuously improved, bugs are eradicated, and features that didn't make an earlier release are added.

One element that distinguishes this process and that cuts across all four phases is an iteration. An *iteration* is a distinct set of activities, with a baselined plan and evaluation criteria that result in a release, either internal or external. This means that the software development life cycle can be characterized as involving a continuous stream of executable releases of the system's architecture. It is this emphasis on architecture as an important artifact that drives the UML to focus on modeling the different views of a system's architecture.

UNIT II

Basic Structural Modeling and Advanced Structural Modeling

Syllabus :Classes, Relationships, common Mechanisms and diagrams.Advanced classes, advanced relationships, Interfaces, Types and Roles, Packages.Classes, Terms, concepts, modeling techniques for Class & Object Diagrams.

Class

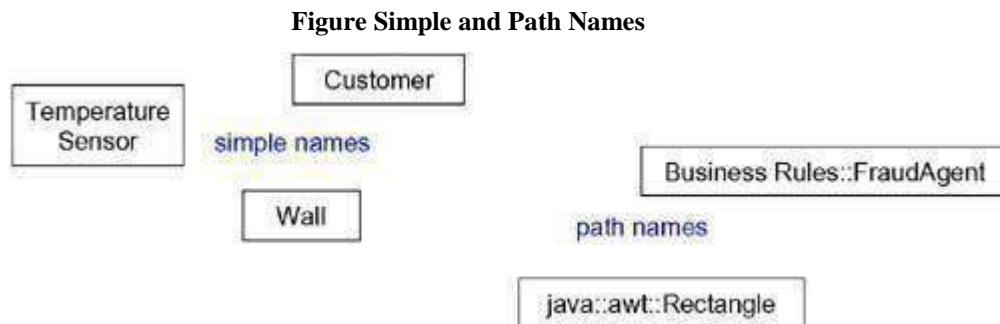
Classes are the most important building block of any object-oriented system. A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. A class implements one or more interfaces.

Terms and Concepts

A *class* is a description of a set of objects that share the same attributes, operations,relationships, and semantics. Graphically, a class is rendered as a rectangle.

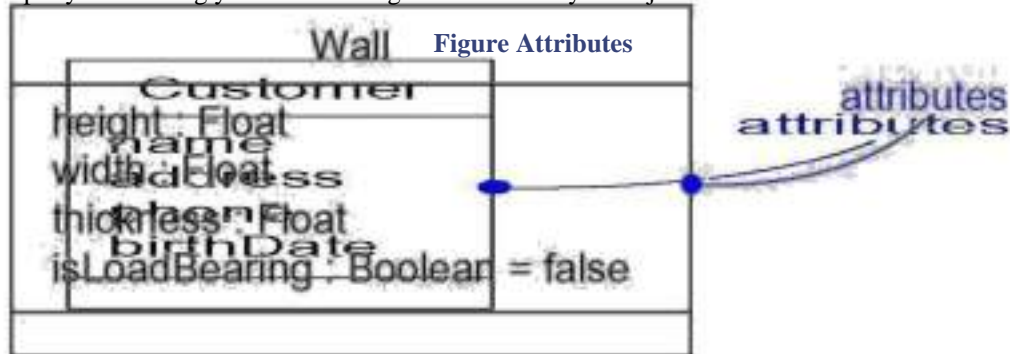
Names

Every class must have a name that distinguishes it from other classes. A *name* is a textual string. That name alone is known as a *simple name*; a *path name* is the class name prefixed by the name of the package in which that class lives. A class may be drawn showing only its name, as Figure shows.



Attributes

An *attribute* is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing you are modeling that is shared by all objects of that class



You can further specify an attribute by stating its class and possibly a default initial value, as shown Figure

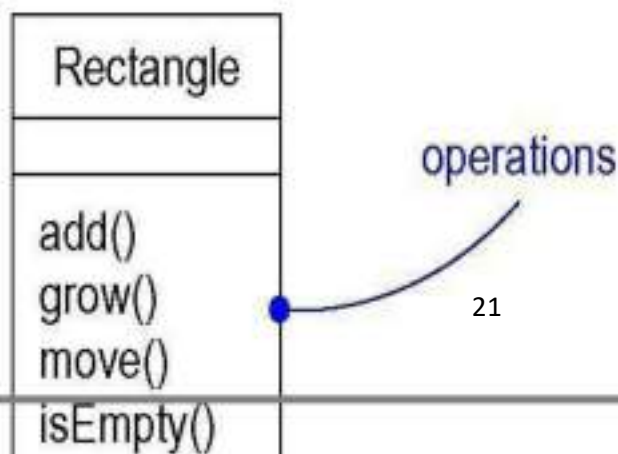
FigureAttributes and Their Class

Operations

An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class. A class may have any number of operations or no operations at all.

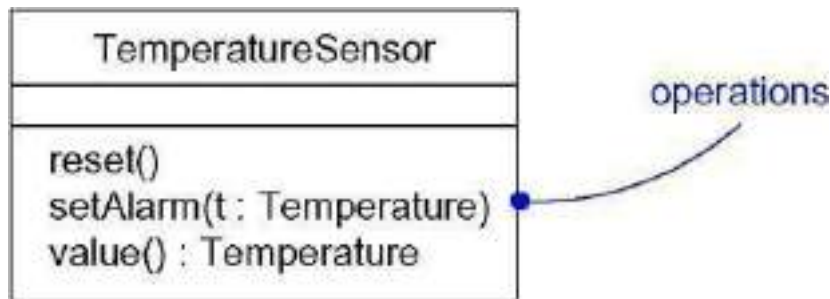
For example, in a windowing library such as the one found in Java's **awt** package, all objects of the class **Rectangle** can be moved, resized, or queried for their properties. Often (but not always), invoking an operation on an object changes the object's data or state. Graphically, operations are listed in a compartment just below the class attributes. Operations may be drawn showing only their names, as in Figure.

Figure Operations



You can specify an operation by stating its signature, covering the name, type, and default value of all parameters and (in the case of functions) a return type, as shown in Figure.

Figure Operations and Their Signatures



Organizing Attributes and Operations

When drawing a class, you don't have to show every attribute and every operation at once. In fact, in most cases, you can't (there are too many of them to put in one figure) and you probably shouldn't (only a subset of these attributes and operations are likely to be relevant to a specific view). For these reasons, you can elide a class, meaning that you can choose to show only some or none of a class's attributes and operations. An empty compartment doesn't necessarily mean there are no attributes or operations, just that you didn't choose to show them. You can explicitly specify that there are more attributes or properties than shown by ending each list with an ellipsis ("...").

To better organize long lists of attributes and operations, you can also prefix each group with a descriptive category by using stereotypes, as shown in Figure .

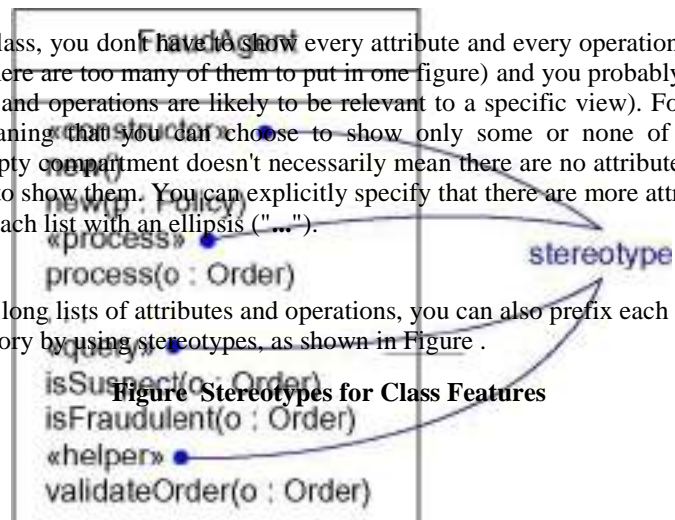


Figure Stereotypes for Class Features

Responsibilities

A *responsibility* is a contract or an obligation of a class. When you create a class, you are making a statement that all objects of that class have the same kind of state and the same kind of behavior. At a more abstract level, these corresponding attributes and operations are just the features by which the class's responsibilities are carried out. A **Wall** class is responsible for knowing about height, width, and thickness; a **FraudAgent** class, as you might find in a credit card application, is responsible for processing orders and determining if they are legitimate, suspect, or fraudulent; a **TemperatureSensor** class is responsible for measuring temperature and raising an alarm if the temperature reaches a certain point.

When you model classes, a good starting point is to specify the responsibilities of the things in your vocabulary. Techniques like CRC cards and use case-based analysis are especially helpful here. A class may have any number of responsibilities, although, in practice, every well-structured class has at least one responsibility and at most just a handful. As you refine your models, you will translate these responsibilities into a set of attributes and operations that best fulfill the class's responsibilities.

Graphically, responsibilities can be drawn in a separate compartment at the bottom of the class icon, as shown in Figure.

Figure Responsibilities



Attributes, operations, and responsibilities are the most common features you'll need when you create abstractions. In fact, for most models you build, the basic form of these three features will be all you need to convey the most important semantics of your classes.

When you build models, you will soon discover that almost every abstraction you create is some kind of class. Sometimes, you will want to separate the implementation of a class from its specification, and this can be expressed in the UML by using interfaces.

When you start building more complex models, you will also find yourself encountering the same kinds of classes over and over again, such as classes that represent concurrent processes and threads, or classes that represent physical things, such as applets, Java Beans, COM+ objects, files, Web pages, and hardware. Because these kinds of classes are so common and because they represent important architectural abstractions, the UML provides active classes (representing processes and threads), components (representing physical software components), and nodes (representing hardware devices).

Finally, classes rarely stand alone. Rather, when you build models, you will typically focus on groups

of classes that interact with one another. In the UML, these societies of classes form collaborations and are usually visualized in class diagrams.

Common Modeling Techniques

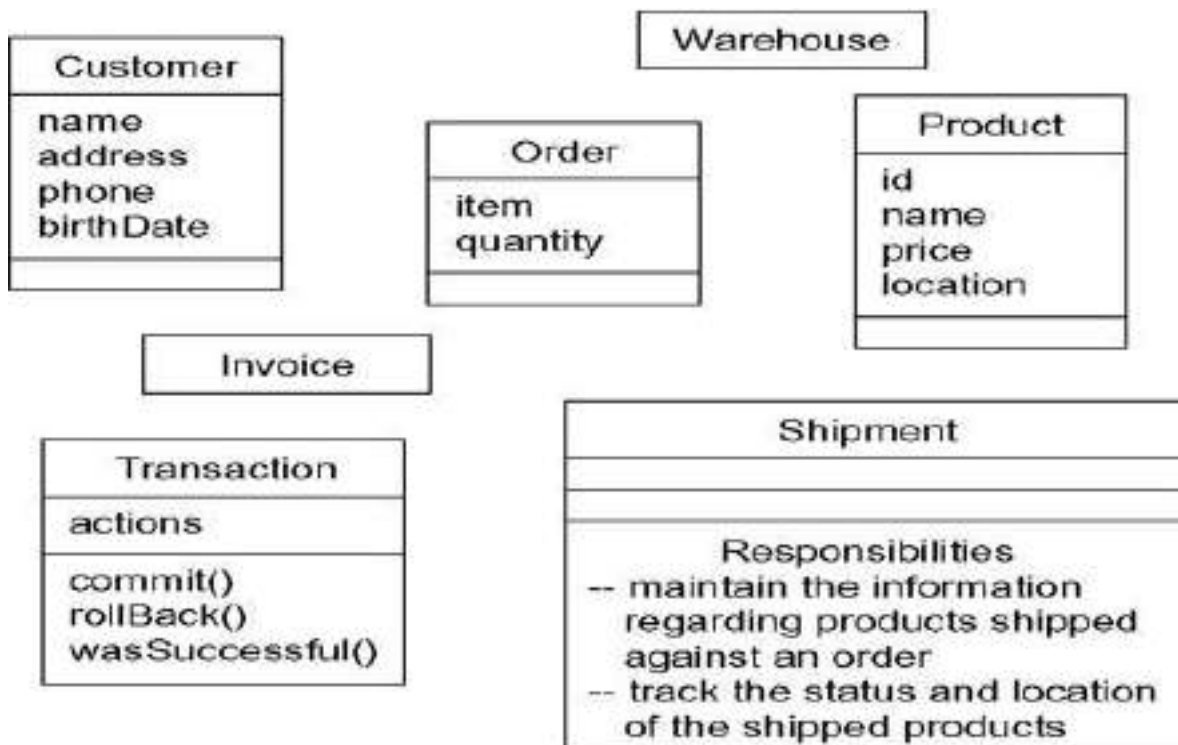
Modeling the Vocabulary of a System

To model the vocabulary of a system,

- Identify those things that users or implementers use to describe the problem or solution. Use CRC cards and use case-based analysis to help find these abstractions.
- For each abstraction, identify a set of responsibilities. Make sure that each class is crisply defined and that there is a good balance of responsibilities among all your classes.
- Provide the attributes and operations that are needed to carry out these responsibilities for each class.

Figure shows a set of classes drawn from a retail system, including **Customer**, **Order**, and **Product**. This figure includes a few other related abstractions drawn from the vocabulary of the problem, such as **Shipment** (used to track orders), **Invoice** (used to bill orders), and **Warehouse** (where products are located prior to shipment). There is also one solution-related abstraction, **Transaction**, which applies to orders and shipments.

Figure Modeling the Vocabulary of a System



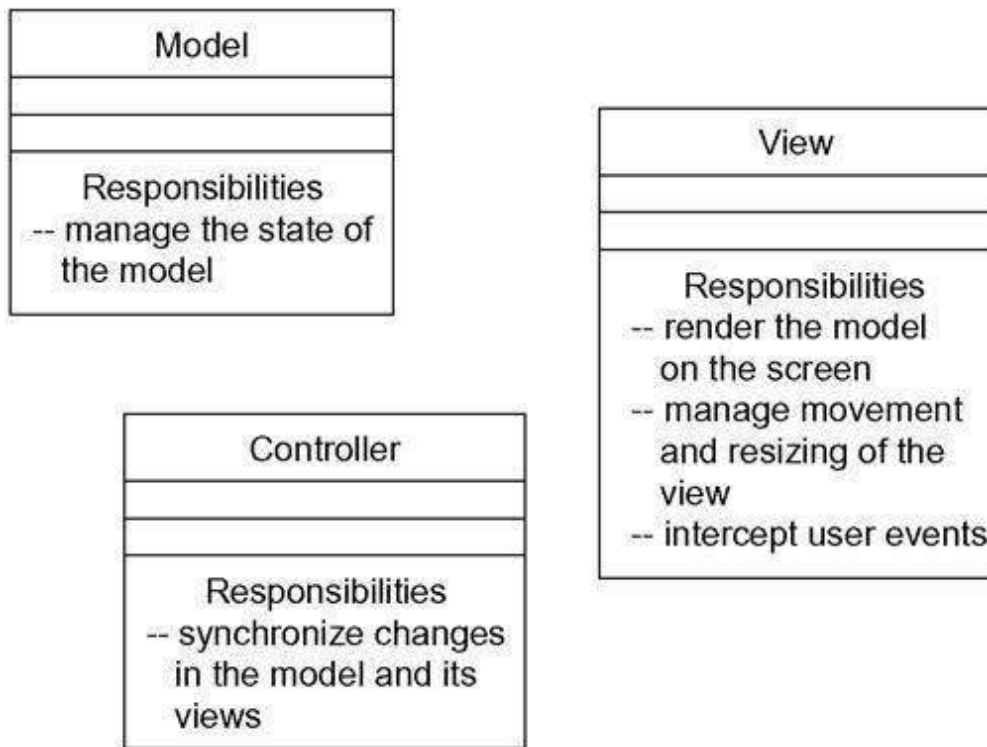
Modeling the Distribution of Responsibilities in a System

To model the distribution of responsibilities in a system,

- Identify a set of classes that work together closely to carry out some behavior.
- Identify a set of responsibilities for each of these classes.
- Look at this set of classes as a whole, split classes that have too many responsibilities into smaller abstractions, collapse tiny classes that have trivial responsibilities into larger ones, and reallocate responsibilities so that each abstraction reasonably stands on its own.
- Consider the ways in which those classes collaborate with one another, and redistribute their responsibilities accordingly so that no class within a collaboration does too much or too little.

For example, Figure shows a set of classes drawn from Smalltalk, showing the distribution of responsibilities among **Model**, **View**, and **Controller** classes. Notice how all these classes work together such that no one class does too much or too little.

Figure Modeling the Distribution of Responsibilities in a System



Modeling Nonsoftware Things

To model nonsoftware things,

- Model the thing you are abstracting as a class.
- If you want to distinguish these things from the UML's defined building blocks, create a new building block by using stereotypes to specify these new semantics and to give a distinctive visual cue.
- If the thing you are modeling is some kind of hardware that itself contains software, consider modeling it as a kind of node, as well, so that you can further expand on its structure.

As Figure shows, it's perfectly normal to abstract humans (like **AccountsReceivableAgent**) and hardware (like **Robot**) as classes, because each represents a set of objects with a common structure and a common behavior.

Figure Modeling Nonsoftware Things



Modeling Primitive Types

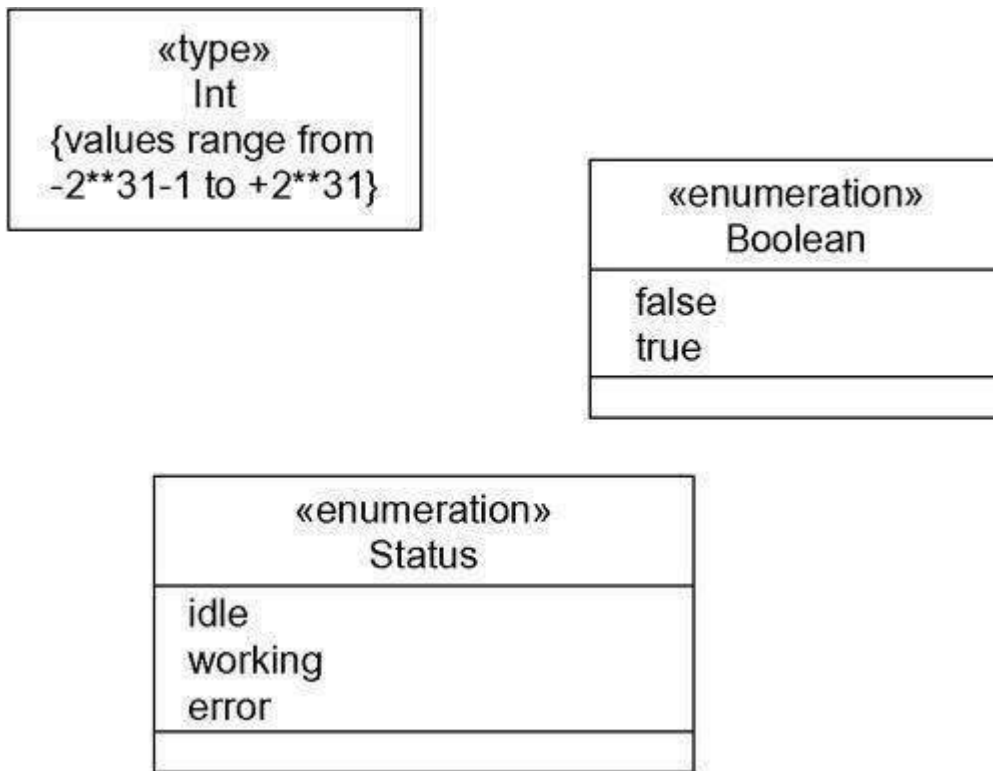
At the other extreme, the things you model may be drawn directly from the programming language you are using to implement a solution. Typically, these abstractions involve primitive types, such as integers, characters, strings, and even enumeration types, that you might create yourself.

To model primitive types,

- Model the thing you are abstracting as a type or an enumeration, which is rendered using class notation with the appropriate stereotype.
- If you need to specify the range of values associated with this type, use constraints.

As Figure shows, these things can be modeled in the UML as types or enumerations, which are rendered just like classes but are explicitly marked via stereotypes. Things like integers (represented by the class **Int**) are modeled as types, and you can explicitly indicate the range of values these things can take on by using a constraint. Similarly, enumeration types, such as **Boolean** and **Status**, can be modeled as enumerations, with their individual values provided as attributes.

Figure Modeling Primitive Types



Relationships

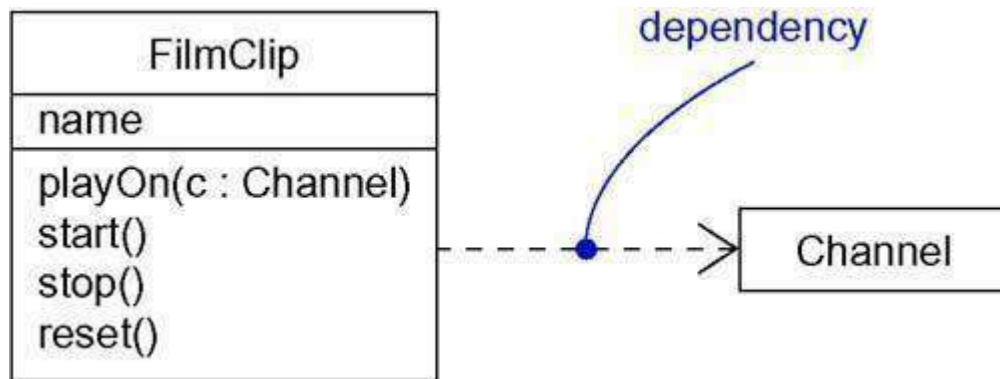
Terms and Concepts

A *relationship* is a connection among things. In object-oriented modeling, the three most important relationships are dependencies, generalizations, and associations. Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the kinds of relationships.

Dependency

A *dependency* is a using relationship that states that a change in specification of one thing (for example, class **Event**) may affect another thing that uses it (for example, class **Window**), but not necessarily the reverse. Graphically, a dependency is rendered as a dashed directed line, directed to the thing being depended on. Use dependencies when you want to show one thing using another.

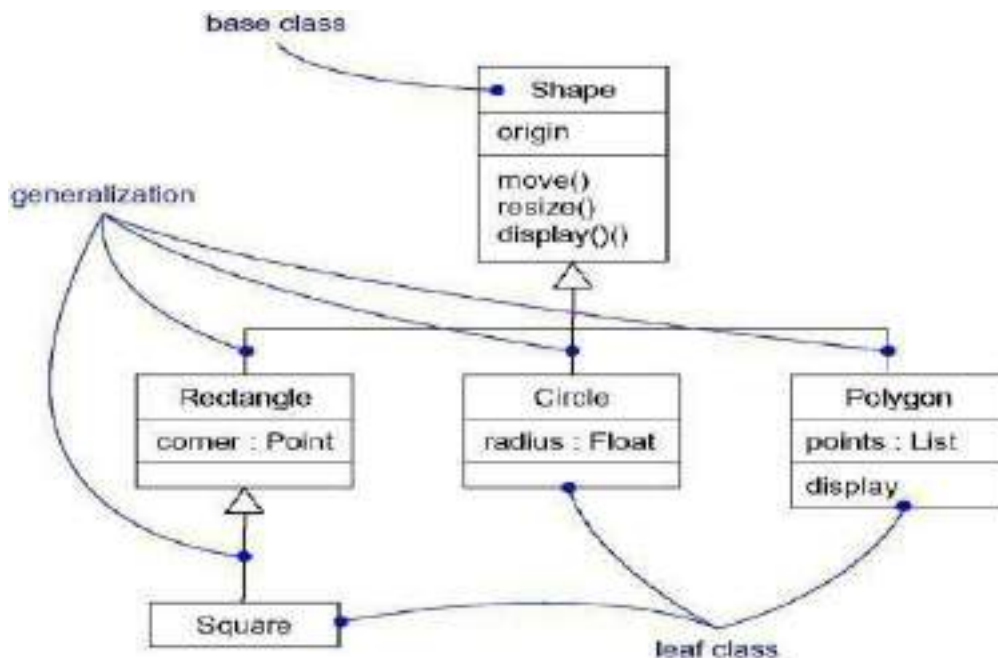
Figure Dependencies



Generalization

A *generalization* is a relationship between a general thing (called the superclass or parent) and a more specific kind of that thing (called the subclass or child). Generalization is sometimes called an "is-a-kind-of" relationship: one thing (like the class **BayWindow**) is-a-kind-of a more general thing (for example, the class **Window**). Generalization means that objects of the child may be used anywhere the parent may appear, but not the reverse. In other words, generalization means that the child is substitutable for the parent. A child inherits the properties of **its parents, especially their attributes and operations. Often• but not always• the child has attributes** and operations in addition to those found in its parents. An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent; this is known as polymorphism. Graphically, generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent, as shown in [Figure](#). Use generalizations when you want to show parent/child relationships.

Figure Generalization



A class may have zero, one, or more parents. A class that has no parents and one or more children is called a root class or a base class. A class that has no children is called a leaf class. A class that has exactly one parent is said to use single inheritance; a class with more than one parent is said to use multiple inheritance.

Association

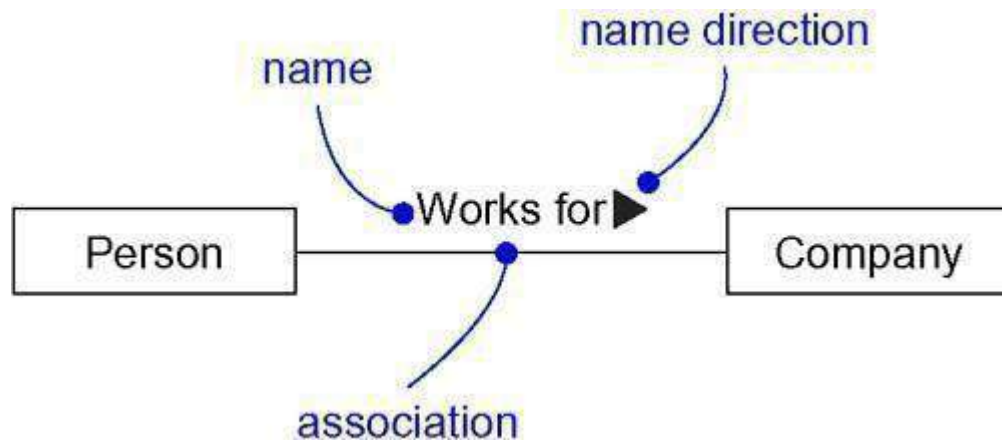
An *association* is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa. It's quite legal to have both ends of an association circle back to the same class. This means that, given an object of the class, you can link to other objects of the same class. An association that connects exactly two classes is called a binary association. Although it's not as common, you can have associations that connect more than two classes; these are called n-ary associations. Graphically, an association is rendered as a solid line connecting the same or different classes. Use associations when you want to show structural relationships.

Beyond this basic form, there are four adornments that apply to associations.

Name

An association can have a name, and you use that name to describe the nature of the relationship. So that there is no ambiguity about its meaning, you can give a direction to the name by providing a direction triangle that points in the direction you intend to read the name, as shown in Figure.

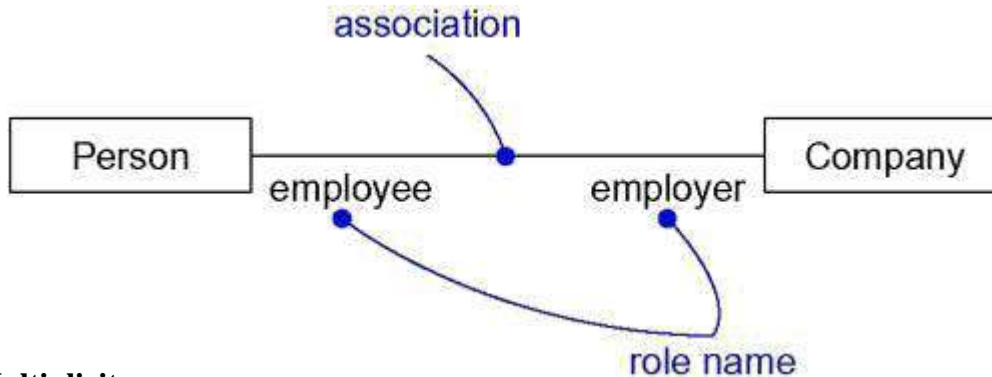
Figure Association Names



Role

When a class participates in an association, it has a specific role that it plays in that relationship; a role is just the face the class at the near end of the association presents to the class at the other end of the association. You can explicitly name the role a class plays in an association. In Figure, a **Person** playing the role of **employee** is associated with a **Company** playing the role of **employer**.

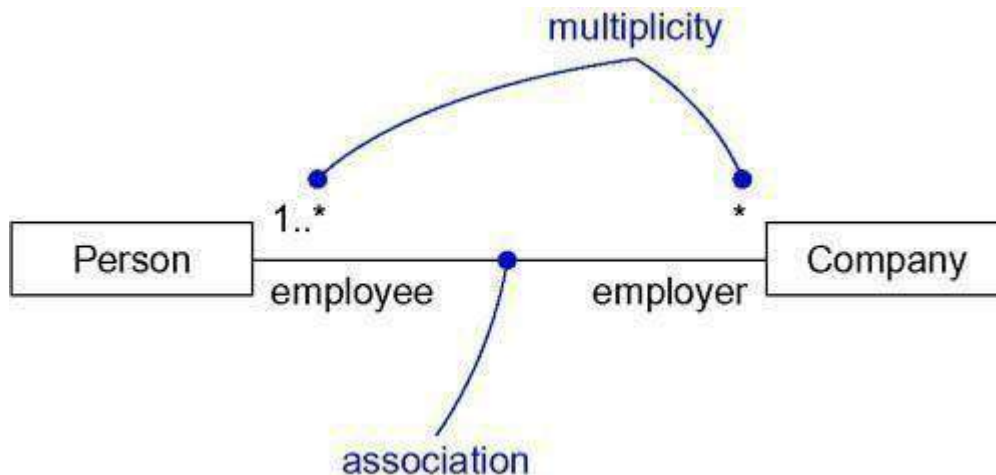
Figure Roles



Multiplicity

An association represents a structural relationship among objects. In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association. This "how many" is called the multiplicity of an association's role, and is written as an expression that evaluates to a range of values or an explicit value as in Figure. When you state a multiplicity at one end of an association, you are specifying that, for each object of the class at the opposite end, there must be that many objects at the near end. You can show a multiplicity of exactly one (**1**), zero or one (**0..1**), many (**0..***), or one or more (**1..***). You can even state an exact number (for example, **3**).

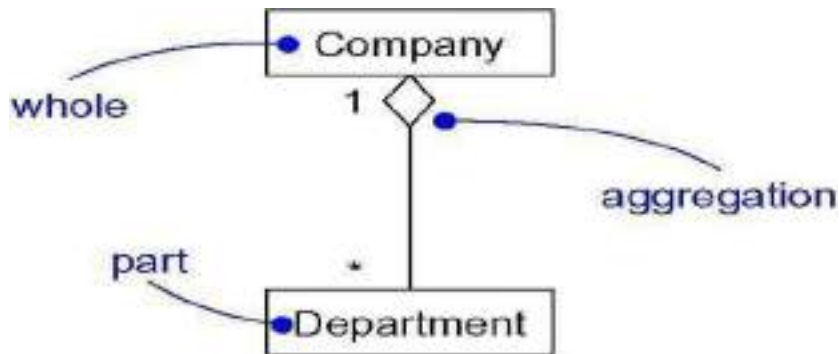
Figure Multiplicity



Aggregation

A plain association between two classes represents a structural relationship between peers, meaning that both classes are conceptually at the same level, no one more important than the other. Sometimes, you will want to model a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts"). This kind of relationship is called aggregation, which represents a "has-a" relationship, meaning that an object of the whole has objects of the part. Aggregation is really just a special kind of association and is specified by adorning a plain association with an open diamond at the whole end, as shown in Figure.

Figure Aggregation



Other Features

Plain, unadorned dependencies, generalizations, and associations with names, multiplicities, and roles are the most common features you'll need when creating abstractions. In fact, for most of the models you build, the basic form of these three relationships will be all you need to convey the most important semantics of your relationships.

Common Modeling Techniques

Modeling Simple Dependencies

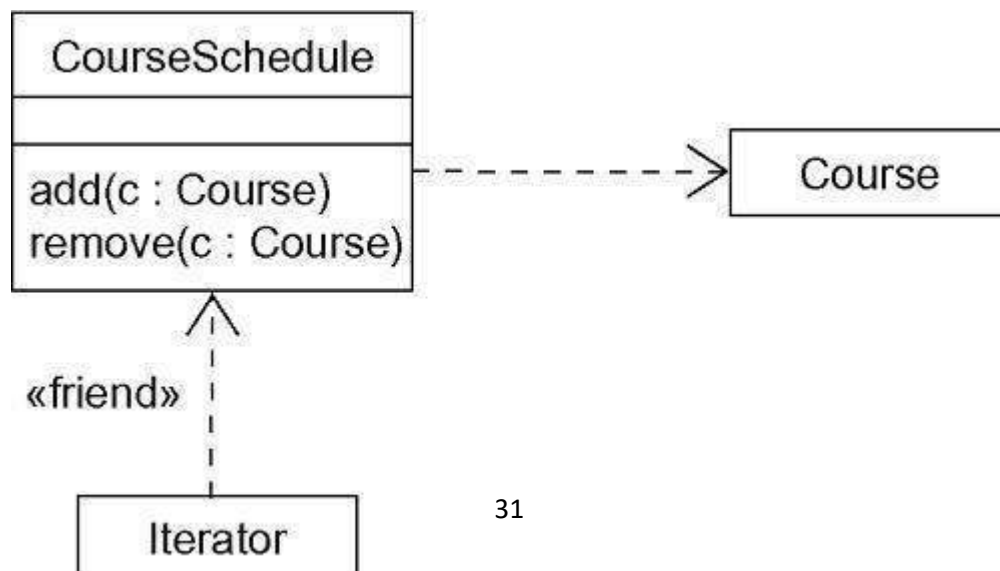
The most common kind of dependency relationship is the connection between a class that only uses another class as a parameter to an operation.

To model this using relationship,

- Create a dependency pointing from the class with the operation to the class used as a parameter in the operation.

For example, Figure shows a set of classes drawn from a system that manages the assignment of students and instructors to courses in a university. This figure shows a dependency from **CourseSchedule** to **Course**, because **Course** is used in both the **add** and **remove** operations of **CourseSchedule**.

Figure Dependency Relationships



Modeling Single Inheritance

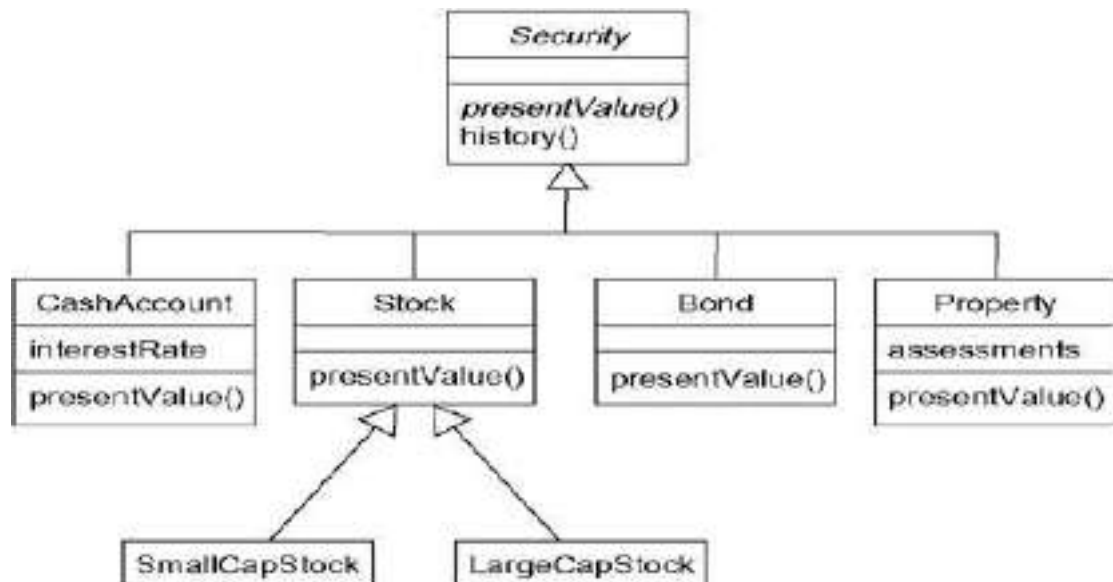
In modeling the vocabulary of your system, you will often run across classes that are structurally or behaviorally similar to others. You could model each of these as distinct and unrelated abstractions. A better way would be to extract any common structural and behavioral features and place them in more-general classes from which the specialized ones inherit.

To model inheritance relationships,

- Given a set of classes, look for responsibilities, attributes, and operations that are common to two or more classes.
- Elevate these common responsibilities, attributes, and operations to a more general class. If necessary, create a new class to which you can assign these elements (but be careful about introducing too many levels).
- Specify that the more-specific classes inherit from the more-general class by placing a generalization relationship that is drawn from each specialized class to its more-general parent.

For example, Figure shows a set of classes drawn from a trading application. You will find a generalization relationship from four classes—**CashAccount**, **Stock**, **Bond**, and **Property**—to the more-general class named **Security**. **Security** is the parent, and **CashAccount**, **Stock**, **Bond**, and **Property** are all children. Each of these specialized children is a kind of **Security**. You'll notice that **Security** includes two operations: **presentValue** and **history**. Because **Security** is their parent, **CashAccount**, **Stock**, **Bond**, and **Property** all inherit these two operations, and for that matter, any other attributes and operations of **Security** that may be elided in this figure.

Figure Inheritance Relationships

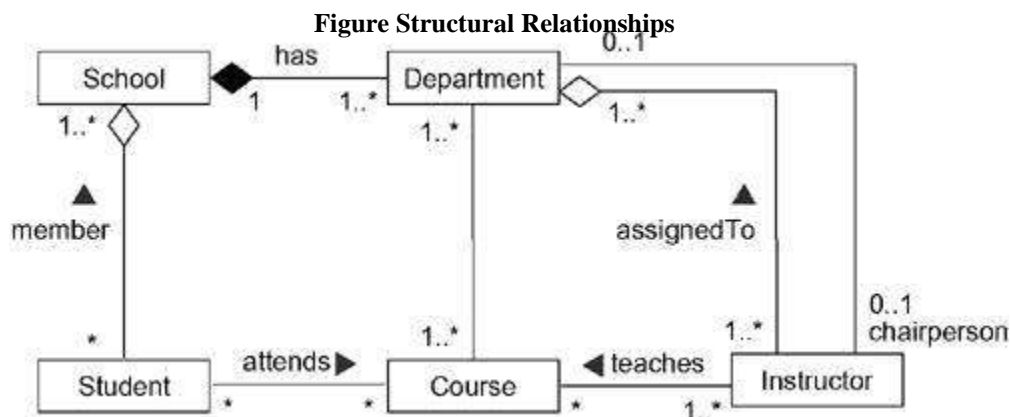


Modeling Structural Relationships

When you model with dependencies or generalization relationships, you are modeling classes that represent different levels of importance or different levels of abstraction. Given a dependency between two classes, one class depends on another but the other class has no knowledge of the one. Given a generalization relationship between two classes, the child inherits from its parent but the parent has no specific knowledge of its children. In short, dependency and generalization relationships are one-sided.

To model structural relationships,

- For each pair of classes, if you need to navigate from objects of one to objects of another, specify an association between the two. This is a data-driven view of associations.
- For each pair of classes, if objects of one class need to interact with objects of the other class other than as parameters to an operation, specify an association between the two. This is more of a behavior-driven view of associations.
- For each of these associations, specify a multiplicity (especially when the multiplicity is not *, which is the default), as well as role names (especially if it helps to explain the model).
- If one of the classes in an association is structurally or organizationally a whole compared with the classes at the other end that look like parts, mark this as an aggregation by adorning the association at the end near the whole.



The relationships between **School** and the classes **Student** and **Department** are a bit different. Here you'll see aggregation relationships. A school has zero or more students, each student may be a registered member of one or more schools, a school has one or more departments, each department belongs to exactly one school. You could leave off the aggregation adornments and use plain associations, but by specifying that **School** is a whole and that **Student** and **Department** are some of its parts, you make clear which one is organizationally superior to the other. Thus, schools are somewhat defined by the students and departments they have. Similarly, students and departments don't really stand alone outside the school to which they belong. Rather, they get some of their identity from their school.

You'll also see that there are two associations between **Department** and **Instructor**. One of these associations specifies that every instructor is assigned to one or more departments and that each department has one or more instructors. This is modeled as an aggregation because organizationally,

departments are at a higher level in the school's structure than are instructors. The other association specifies that for every department, there is exactly one instructor who is the department chair. The way this model is specified, an instructor can be the chair of no more than one department and some instructors are not chairs of any department.

Common Mechanisms

Terms and Concepts

A *note* is a graphical symbol for rendering constraints or comments attached to an element or a collection of elements. Graphically, a note is rendered as a rectangle with a dog-eared corner, together with a textual or graphical comment.

A *stereotype* is an extension of the vocabulary of the UML, allowing you to create new kinds of building blocks similar to existing ones but specific to your problem. Graphically, a stereotype is rendered as a name enclosed by guillemets and placed above the name of another element. As an option, the stereotyped element may be rendered by using a new icon associated with that stereotype.

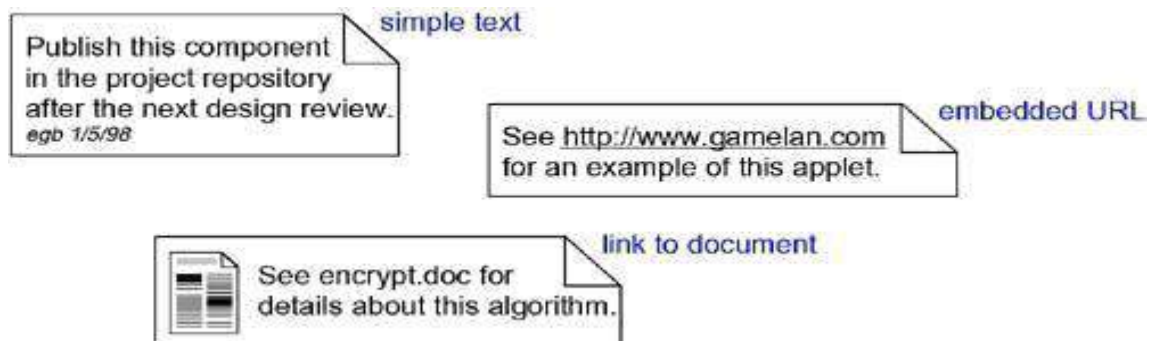
A *tagged value* is an extension of the properties of a UML element, allowing you to create new information in that element's specification. Graphically, a tagged value is rendered as a string enclosed by brackets and placed below the name of another element.

A *constraint* is an extension of the semantics of a UML element, allowing you to add new rules or to modify existing ones. Graphically, a constraint is rendered as a string enclosed by brackets and placed near the associated element or connected to that element or elements by dependency relationships. As an alternative, you can render a constraint in a note.

Notes

A note that renders a comment has no semantic impact, meaning that its contents do not alter the meaning of the model to which it is attached. This is why notes are used to specify things like requirements, observations, reviews, and explanations, in addition to rendering constraints.

Figure Notes

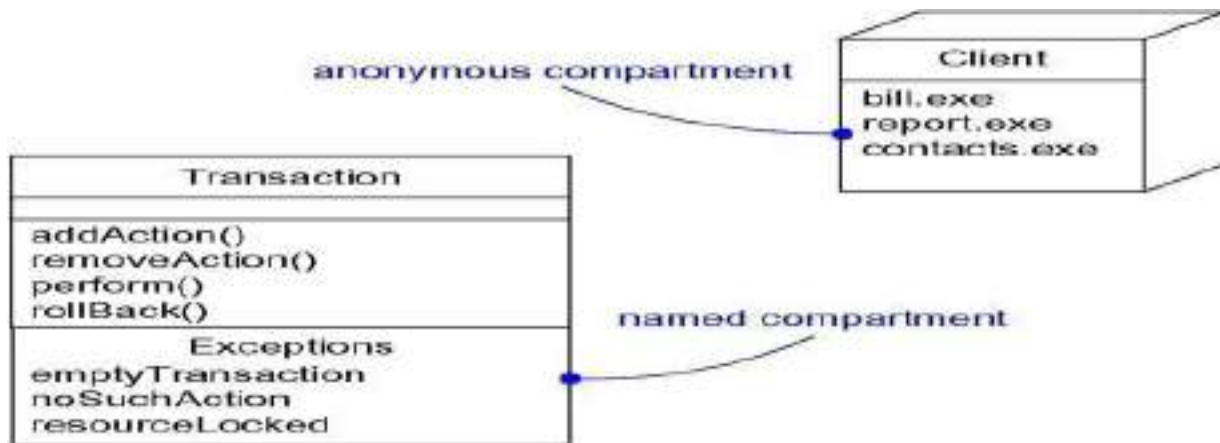


Other Adornments

Adornments are textual or graphical items that are added to an element's basic notation and are used to visualize details from the element's specification. For example, the basic notation for an association is a line, but this may be adorned with such details as the role and multiplicity of each end. In using the UML, the general rule to follow is this: Start with the basic notation for each element and then add other adornments only as they are necessary to convey specific information that is important to your model.

Most adornments are rendered by placing text near the element of interest or by adding a graphic symbol to the basic notation. However, sometimes you'll want to adorn an element with more detail than can be accommodated by simple text or graphics. In the case of such things as classes, components, and nodes, you can add an extra compartment below the usual compartments to provide this information, as [Figure](#) shows.

Figure Extra Compartments



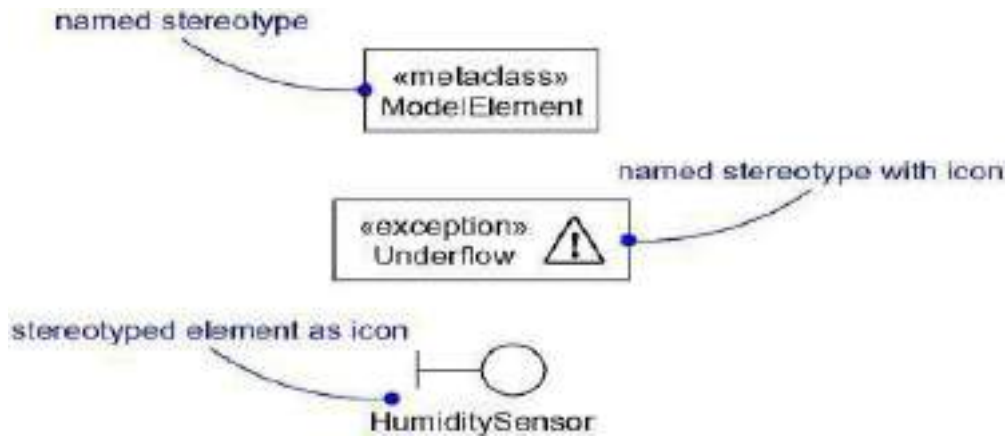
Stereotypes

The UML provides a language for structural things, behavioral things, grouping things, and notational things.

A stereotype is not the same as a parent class in a parent/child generalization relationship. Rather, you can think of a stereotype as a metatype, because each one creates the equivalent of a new class in the UML's metamodel. For example, if you are modeling a business process, you'll want to introduce things like workers, documents, and policies.

In its simplest form, a stereotype is rendered as a name enclosed by guillemets (for example, »name) and placed above the name of another element. As a visual cue, you may define an icon for the stereotype and render that icon to the right of the name (if you are using the basic notation for the element) or use that icon as the basic symbol for the stereotyped item. All three of these approaches are illustrated in [Figure](#).

Figure Stereotypes

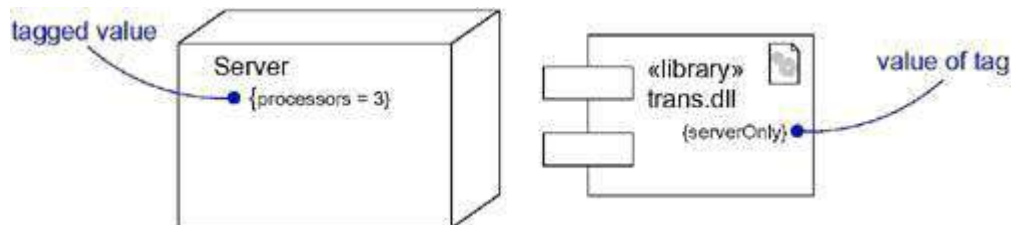


Tagged Values

Every thing in the UML has its own set of properties: classes have names, attributes, and operations; associations have names and two or more ends (each with its own properties); and so on. With stereotypes, you can add new things to the UML; with tagged values, you can add new properties.

You can define tags for existing elements of the UML, or you can define tags that apply to individual stereotypes so that everything with that stereotype has that tagged value. A tagged value is not the same as a class attribute. Rather, you can think of a tagged value as metadata because its value applies to the element itself, not its instances. For example, as Figure shows, you might want to specify the number of processors installed on each kind of node in a deployment diagram, or you might want to require that every component be stereotyped as a library if it is intended to be deployed on a client or a server.

Figure Tagged Values

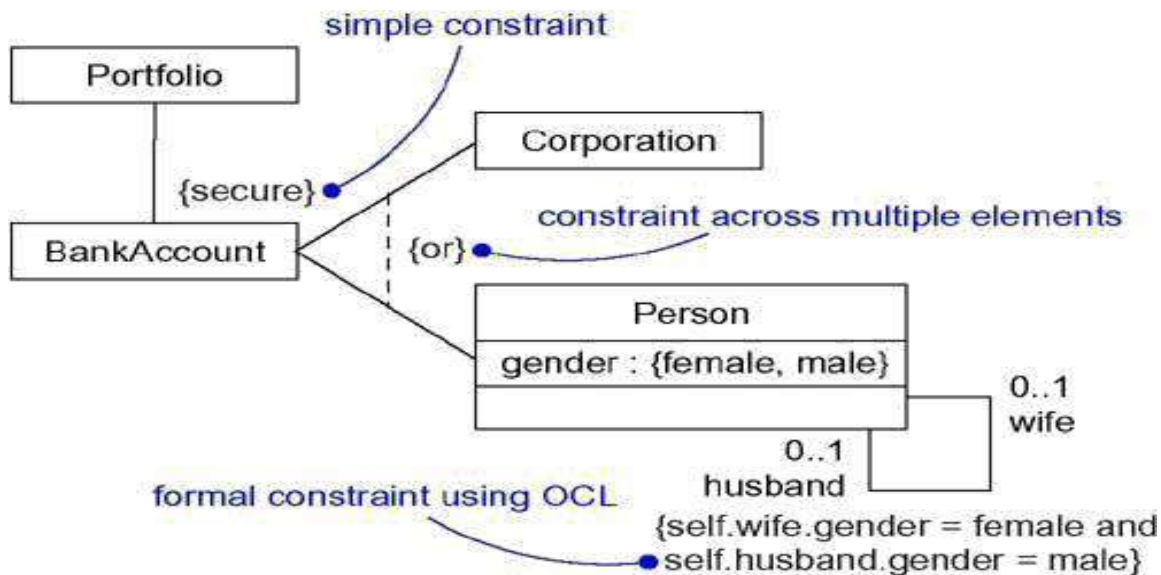


Constraints

Everything in the UML has its own semantics. Generalization implies the Liskov substitution principle, and multiple associations connected to one class denote distinct relationships. With constraints, you can add new semantics or change existing rules. A constraint specifies conditions that must be held true for the model to be well-formed.

For example, as Figure shows, you might want to specify that, across a given association, communication is encrypted. Similarly, you might want to specify that among a set of associations, only one is manifest at a time.

Figure Constraint



A constraint is rendered as a string enclosed by brackets and placed near the associated element. This notation is also used as an adornment to the basic notation of an element to visualize parts of an element's specification that have no graphical cue. For example, some properties of associations (order and changeability) are rendered using constraint notation.

Standard Elements

The UML defines a number of standard stereotypes for classifiers, components, relationships, and other modeling elements. There is one standard stereotype, mainly of interest to tool builders, that lets you model stereotypes themselves.

Stereotype	Specifies that the classifier is a stereotype that may be applied to other elements
documentation	Specifies a comment, description, or explanation of the element to which it attached

The UML also specifies one standard tagged value that applies to all modeling elements.

You'll use this tagged value when you want to attach a comment directly to the specification of an element, such as a class.

Common Modeling Techniques

Modeling Comments

To model a comment

To model a comment, Put your comment as text in a note and place it adjacent to the element to

which it refers. You can show a more explicit relationship by connecting a note to its elements using a dependency relationship.

- Remember that you can hide or make visible the elements of your model as you see fit. This means that you don't have to make your comments visible everywhere the elements to which it is attached are visible. Rather, expose your comments in your diagrams only insofar as you need to communicate that information in that context
- If your comment is lengthy or involves something richer than plain text, consider putting your comment in an external document and linking or embedding that document in a note attached to your model
- As your model evolves, keep those comments that record significant decisions that cannot be **inferred from the model itself, and• unless they are of historic interest• discard the others**

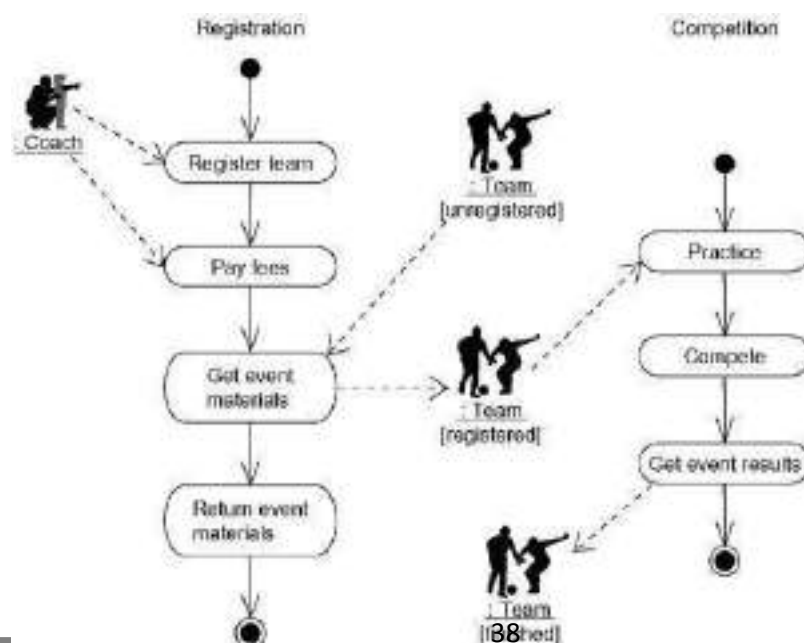
For example, Figure shows a model that's a work in progress of a class hierarchy, showing some requirements that shape the model as well as some notes from a design review.

Modeling New Building Blocks

To model new building blocks,

- Make sure there's not already a way** to express what you want by using basic UML. If you have a common **modeling problem, chance are there's already some standard stereotype that will do** what you want.
- If you're convinced there's no other way to express these semantics, identify the primitive thing in the UML that's most like what you want to model (for example, class, interface, component, node, association, and so on) and define a new stereotype for that thing. Remember that you can define hierarchies of stereotypes so that you can have general kinds of stereotypes along with their specializations (but as with any hierarchy, use this sparingly).
- Specify the common properties and semantics that go beyond the basic element being stereotyped by defining a set of tagged values and constraints for the stereotype
- If you want these stereotype elements to have a distinctive visual cue, define a new icon for the stereotype

Figure Modeling New Building Blocks.



Modeling New Properties

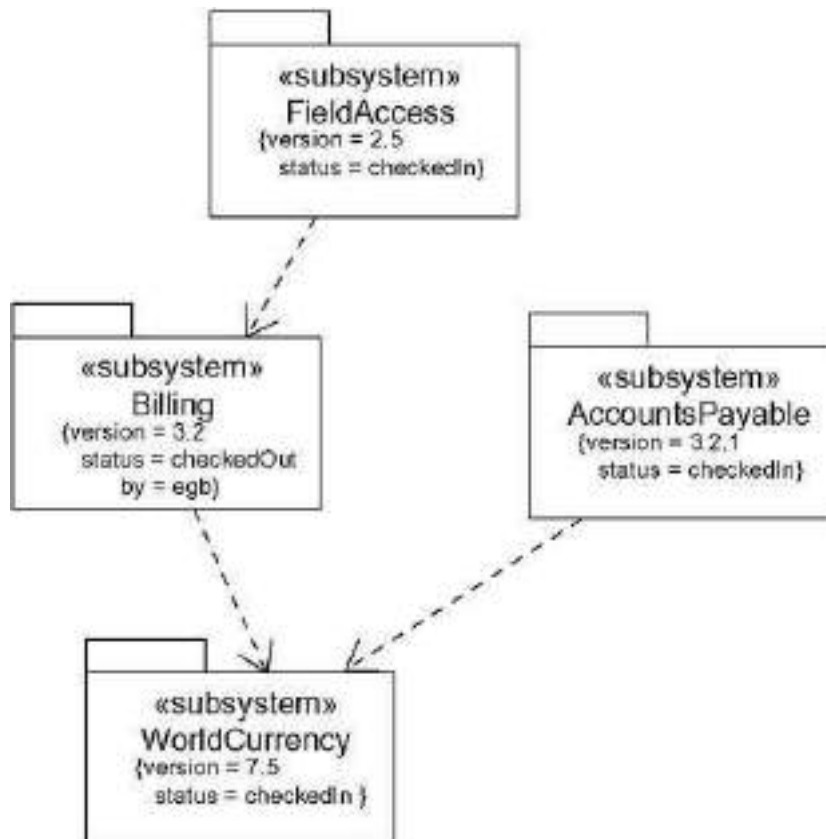
To model new properties,

- ❑ First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already some standard tagged value that will do what you want.
- ❑ If you're convinced there's no other way to express these semantics, add this new property to an individual element or a stereotype. The rules of generalization **apply** tagged values defined for one kind of element apply to its children.

For example, suppose you want to tie the models you create to your project's configuration management system. Among other things, this means keeping track of the version number, current check in/check out status, and perhaps even the creation and modification dates of each subsystem. Because this is process-specific information, it is not a basic part of the UML, although you can add this information as tagged values. Furthermore, this information is not just a class attribute either. A subsystem's version number is part of its metadata, not part of the model.

Figure shows four subsystems, each of which has been extended to include its version number and status. In the case of the **Billing** subsystem, one other tagged value is shown• the person who has currently checked out the subsystem.

Figure Modeling New Properties



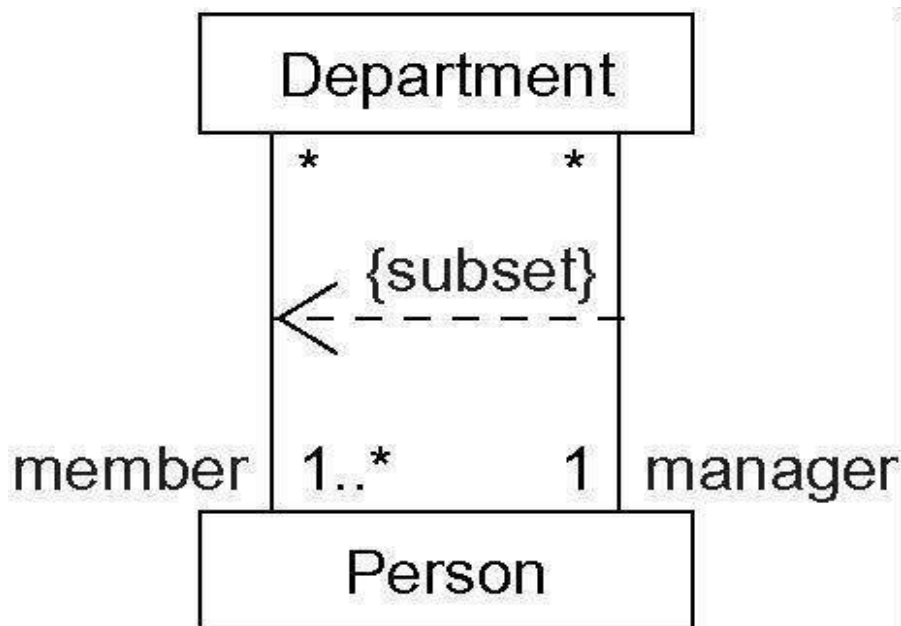
Modeling New Semantics

To model new semantics,

- First, make sure there's not already a way to express what you want by using basic UML. If you have a common modeling problem, chances are that there's already some standard constraint that will do what you want.
- If you're convinced there's no other way to express these semantics, write your new semantics as text in a constraint and place it adjacent to the element to which it refers. You can show a more explicit relationship by connecting a constraint to its elements using a dependency relationship.
- If you need to specify your semantics more precisely and formally, write your new semantics using OCL.

For example, Figure models a small part of a corporate human resources system.

Figure Modeling New Semantics



This diagram shows that each person may be a member of zero or more departments and that each department must have at least one person as a member. This diagram goes on to indicate that each department must have exactly one person as a manager and every person may be the manager of zero or more departments. All of these semantics can be expressed using simple uml. However, to assert that a manager must also be a member of the department is something that cuts across multiple associations and cannot be expressed using simple uml. To state this invariant, you have to write a constraint that shows the manager as a subset of the members of the department, connecting the two associations and the constraint by a dependency from the subset to the superset.

Diagrams

A *system* is a collection of subsystems organized to accomplish a purpose and described by a set of models, possibly from different viewpoints.

A *subsystem* is a grouping of elements, of which some constitute a specification of the behavior offered by the other contained elements.

A *model* is a semantically closed abstraction of a system, meaning that it represents a complete and self-consistent simplification of reality, created in order to better understand the system. In the context of architecture, a *view* is a projection into the organization and structure of a system's model, focused on one aspect of that system. A *diagram* is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).

In modeling real systems, no matter what the problem domain, you'll find yourself creating the same kinds of diagrams, because they represent common views into common models. Typically, you'll view the static parts of a system using one of the four following diagrams.

1. Class diagram
2. Object diagram
3. Component diagram
4. Deployment diagram

You'll often use five additional diagrams to view the dynamic parts of a system.

1. Use case diagram
2. Sequence diagram
3. Collaboration diagram
4. Statechart diagram
5. Activity diagram

The UML defines these nine kinds of diagrams.

Structural Diagrams

The UML's structural diagrams are roughly organized around the major groups of things you'll find when modeling a system.

- | | | |
|---|---------------------|--------------------------------------|
| 1 | Class diagram | Class, interfaces and collaborations |
| 2 | Objects diagram | Objects |
| 3 | Component diagram | Components |
| 4 | Development diagram | Nodes |

Class Diagram

A *class diagram* shows a set of classes, interfaces, and collaborations and their relationships. Class diagrams are the most common diagram found in modeling object-oriented systems. You use class diagrams to illustrate the static design view of a system. Class diagrams that include active classes are used to address the static process view of a system.

Object Diagram

An *object diagram* shows a set of objects and their relationships. You use object diagrams to illustrate data structures, the static snapshots of instances of the things found in class diagrams. Object diagrams address the static design view or static process view of a system just as do class diagrams, but from the perspective of real or prototypical cases.

Component Diagram

A *component diagram* shows a set of components and their relationships. You use component diagrams to illustrate the static implementation view of a system. Component diagrams are related to class diagrams in that a component typically maps to one or more classes, interfaces, or collaborations.

Deployment Diagram

A *deployment diagram* shows a set of nodes and their relationships. You use deployment diagrams to illustrate the static deployment view of an architecture. Deployment diagrams are related to component diagrams in that a node typically encloses one or more components.

Behavioral Diagrams

The UML's five behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system.

The UML's behavioral diagrams are roughly organized around the major ways you can model the dynamics of a system.

- | | | |
|---|-----------------------|--|
| 1 | Use case diagram | Organizes the behaviors of the system |
| 2 | Sequence diagram | Focused on the time ordering of messages |
| 3 | Collaboration diagram | Focused on the structural organization of objects that send and receive messages |
| 4 | State chart diagram | Focused on the changing state of a system driven by events |
| 5 | Activity diagram | Focused on the flow of control from activity to activity |

Use Case Diagram

A *use case diagram* shows a set of use cases and actors (a special kind of class) and their relationships. You apply use case diagrams to illustrate the static use case view of a system. Use case diagrams are especially important in organizing and modeling the behaviors of a system.

Interaction diagram is the collective name given to sequence diagrams and collaboration diagrams. All sequence diagrams and collaborations are interaction diagrams, and an interaction diagram is either a sequence diagram or a collaboration diagram.

Sequence Diagram

A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. A sequence diagram shows a set of objects and the messages sent and received by those objects. The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes. You use sequence diagrams to illustrate the dynamic view of a system.

Collaboration Diagram

A *collaboration diagram* is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. A collaboration diagram shows a set of objects, links among those objects, and messages sent and received by those objects. The objects are typically named or anonymous instances of classes, but may also represent instances of other things, such as collaborations, components, and nodes. You use collaboration diagrams to illustrate the dynamic view of a system.

State Chart Diagram

A *statechart diagram* shows a state machine, consisting of states, transitions, events, and activities. You use statechart diagrams to illustrate the dynamic view of a system. They are especially important in modeling the behavior of an interface, class, or collaboration. Statechart diagrams emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

Activity Diagram

An *activity diagram* shows the flow from activity to activity within a system. An activity shows a set of activities, the sequential or branching flow from activity to activity, and objects that act and are acted upon. You use activity diagrams to illustrate the dynamic view of a system. Activity diagrams are especially important in modeling the function of a system. Activity diagrams emphasize the flow of control among objects.

Common Modeling Techniques

Modeling Different Views of a System

To model a system from different views,

- Decide which views you need to best express the architecture of your system and to expose the technical risks to your project. The five views of an architecture described earlier are a good starting point.
- For each of these views, decide which artifacts you need to create to capture the essential details of that view. For the most part, these artifacts will consist of various UML diagrams.

- As part of your process planning, decide which of these diagrams you'll want to put under some sort of formal or semi-formal control. These are the diagrams for which you'll want to schedule reviews and to preserve as documentation for the project.
- Allow room for diagrams that are thrown away. Such transitory diagrams are still useful for exploring the implications of your decisions and for experimenting with changes.

For example, if you are modeling a simple monolithic application that runs on a single machine, you might need only the following handful of diagrams.

Use case view	Use case diagrams
Design view	Class diagrams (for structural modeling) Interaction diagrams (for behavioral modeling)
Process view	None required
Implementation view	None required
Deployment view	None required

Finally, if you are modeling a complex, distributed system, you'll need to employ the full range of the UML's diagrams in order to express the architecture of your system and the technical risks to your project, as in the following.

Use case view	Use case diagrams Activity diagrams (for behavioral modeling)
Design view	Class diagrams (for structural modeling) Interaction diagrams (for behavioral modeling) statechart diagram (for behavioral modeling)
Process view	Class diagrams (for structural modeling) Interaction diagrams (for behavioral modeling)
Implementation view	Component diagram
Deployment view	Component diagram

Modeling Different Levels of Abstraction

To model a system at different levels of abstraction by presenting diagrams with different levels of detail,

- Consider the needs of your readers, and start with a given model.
- If your reader is using the model to construct an implementation, she'll need diagrams that are at a lower level of abstraction, which means that they'll need to reveal a lot of detail. If she is using the model to present a conceptual model to an end user, she'll need diagrams that are at a higher level of abstraction, which means that they'll hide a lot of detail.
- Depending on where you land in this spectrum of low-to-high levels of abstraction, create a diagram at the right level of abstraction by hiding or revealing the following four categories of things from your model:

1. Building blocks and relationships:

Hide those that are not relevant to the intent of your diagram or the needs of your reader.

2. Adornments:

Reveal only the adornments of these building blocks and relationships that are essential to understanding your intent.

3. Flow:

In the context of behavioral diagrams, expand only those messages or transitions that are essential to understanding your intent.

4. Stereotypes:

In the context of stereotypes used to classify lists of things, such as attributes and operations, reveal only those stereotyped items that are essential to understanding your intent.

The main advantage of this approach is that you are always modeling from a common semantic repository. The main disadvantage of this approach is that changes from diagrams at one level of abstraction may make obsolete diagrams at a different level of abstraction.

To model a system at different levels of abstraction by creating models at different levels of abstraction,

- Consider the needs of your readers and decide on the level of abstraction that each should view, forming a separate model for each level.
- In general, populate your models that are at a high level of abstraction with simple abstractions and your models that are at a low level of abstraction with detailed abstractions. Establish trace dependencies among the related elements of different models.
- In practice, if you follow the five views of an architecture, there are four common situations you'll encounter when modeling a system at different levels of abstraction:

1. Use cases and their realization:

Use cases in a use case model will trace to collaborations in a design model.

2. Collaborations and their realization:

Collaborations will trace to a society of classes that work together to carry out the collaboration.

3. Components and their design:

Components in an implementation model will trace to the elements in a design model.

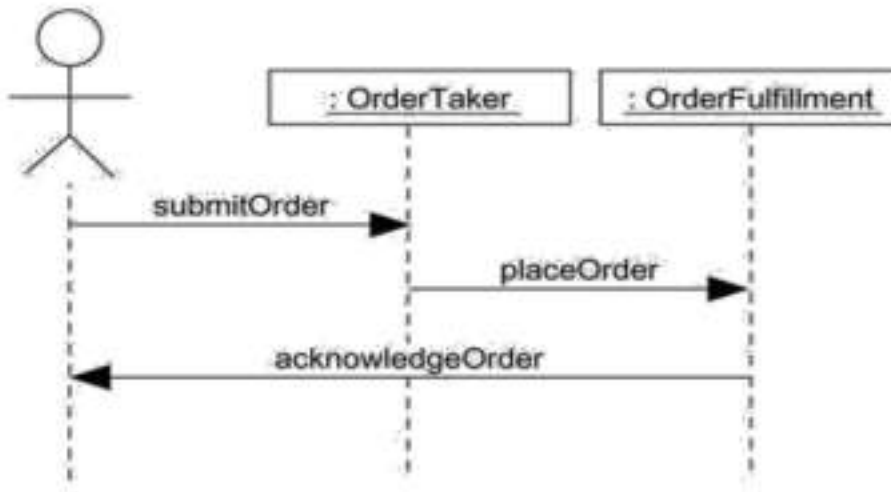
4. Nodes and their components:

Nodes in a deployment model will trace to components in an implementation model.

The main advantage of the approach is that diagrams at different levels of abstraction remain more loosely coupled. This means that changes in one model will have less direct effect on other models. The main disadvantage of this approach is that you must spend resources to keep these models and their diagrams synchronized. This is especially true when your models parallel different phases of the software development life cycle, such as when you decide to maintain an analysis model separate from a design model.

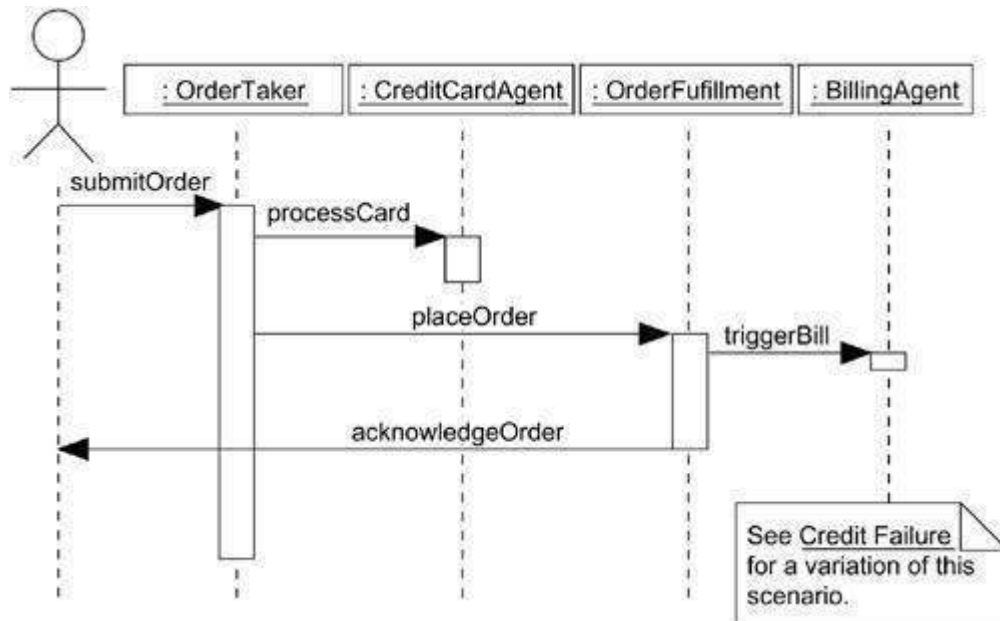
For example, suppose you are modeling a system for Web commerce• one of the main use cases of such a system would be for placing an order. If you're an analyst or an end user, you'd probably create some interaction diagrams at a high level of abstraction that show the action of placing an order, as in Figure.

Figure Interaction Diagram at a High Level of Abstraction



On the other hand, a programmer responsible for implementing this scenario would have to build on this diagram, expanding certain messages and adding other players in this interaction, as in Figure .

Figure Interaction at a Low Level of Abstraction



Both of these diagrams work against the same model, but at different levels of detail. It's reasonable to have many diagrams such as these, especially if your tools make it easy to navigate from one diagram to another.

Modeling Complex Views

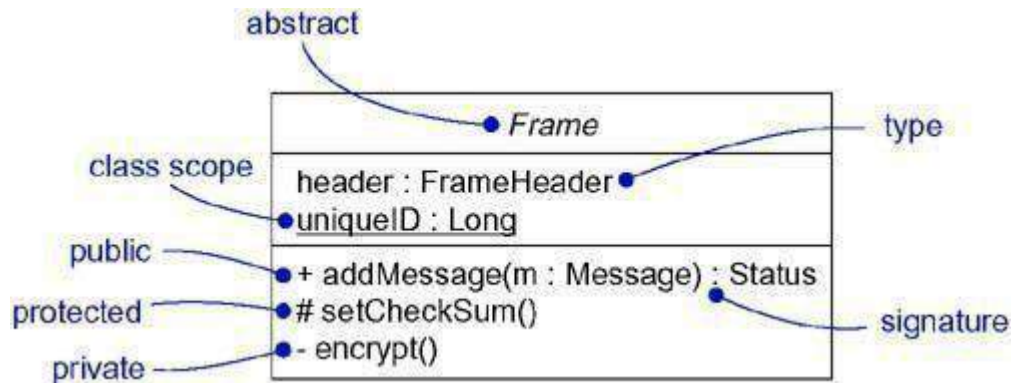
To model complex views,

- First, convince yourself there's no meaningful way to present this information at a higher level of abstraction, perhaps eliding some parts of the diagram and retaining the detail in other parts.
- If you've hidden as much detail as you can and your diagram is still complex, consider grouping some of the elements in packages or in higher level collaborations, then render only those packages or collaborations in your diagram.

- If your diagram is still complex, use notes and color as visual cues to draw the reader's attention to the points you want to make.
- If your diagram is still complex, print it in its entirety and hang it on a convenient large wall. You lose the interactivity an online version of the diagram brings, but you can step back from the diagram and study it for common patterns.

Classifiers (and especially classes) have a number of advanced features beyond the simpler properties of attributes and operations described in the previous section: You can model multiplicity, visibility, signatures, polymorphism, and other characteristics. In the UML, you can model the semantics of a class so that you can state its meaning to whatever degree of formality you like.

Figure Advanced Classes



Advanced classes

Terms and Concepts

A *classifier* is a mechanism that describes structural and behavioral features. Classifiers include classes, interfaces, datatypes, signals, components, nodes, use cases, and subsystems.

Classifiers

When you model, you'll discover abstractions that represent things in the real world and things in your solution. For example, if you are building a Web-based ordering system, the vocabulary of your project will likely include a **Customer** class (representing people who order products) and a **Transaction** class (an implementation artifact, representing an atomic action). In the deployed system, you might have a **Pricing** component, with instances living on every client node. Each of these abstractions will have instances; separating the essence and the instance of the things in your world is an important part of modeling.

The most important kind of classifier in the UML is the class. A class is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Classes are not the only kind of classifier, however. The UML provides a number of other kinds of classifiers to help you model.

Interface	A collection of operations that are used to specify a service of a class or a component
-----------	---

Datatype	A type whose values have no identity, including primitive built-in types (such as numbers and strings), as well as enumeration types (such as Boolean)
Signal	The specification of an asynchronous stimulus communicated between instances
Component re	A physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces
Node	A physical element that exists at run time and that represents a computational resource, generally having at least some memory and often processing capability
Use case	A description of a set of a sequence of actions, including variants, that a system performs that yields an observable result of value to a particular actor
Subsystem	A grouping of elements of which some constitute a specification of the behavior offered by the other contained elements

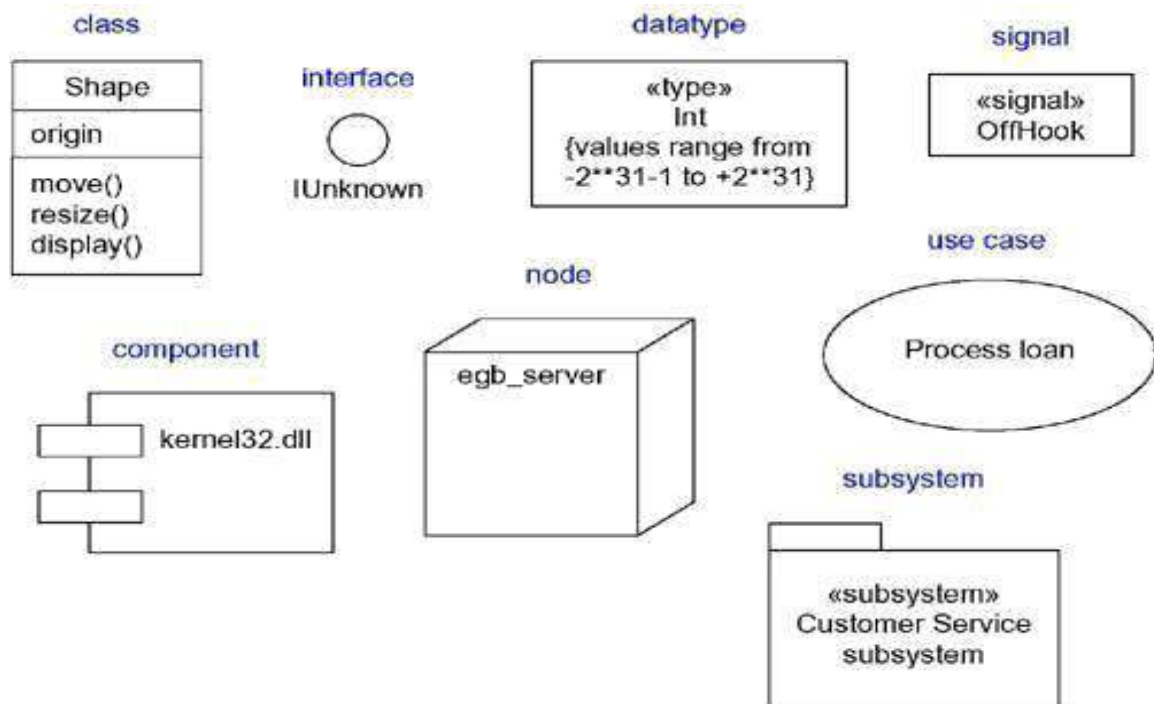
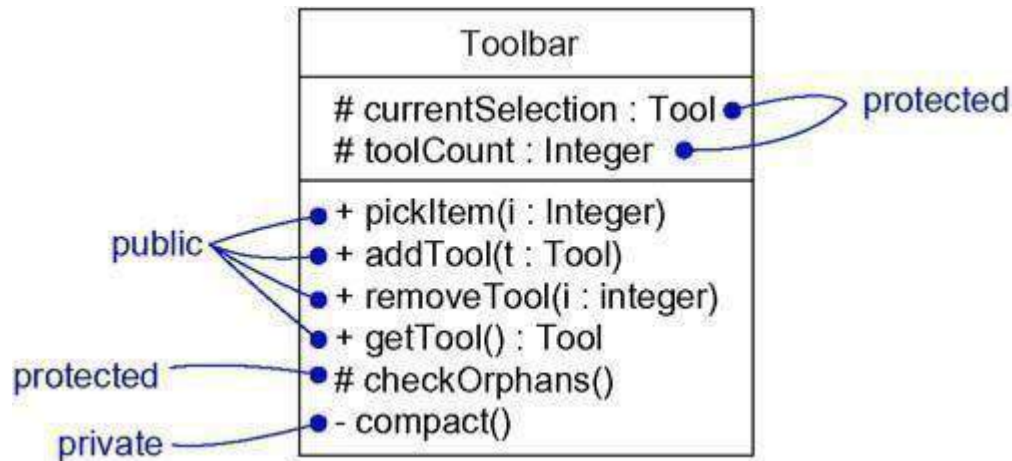


Figure Classifiers

Visibility
1. public
2. private
3. protected

Figure Visibility



When you specify the visibility of a classifier's features, you generally want to hide all its implementation details and expose only those features that are necessary to carry out the responsibilities of the abstraction. That's the very basis of information hiding, which is essential to building solid, resilient systems. If you don't explicitly adorn a feature with a visibility symbol, you can usually assume that it is public.

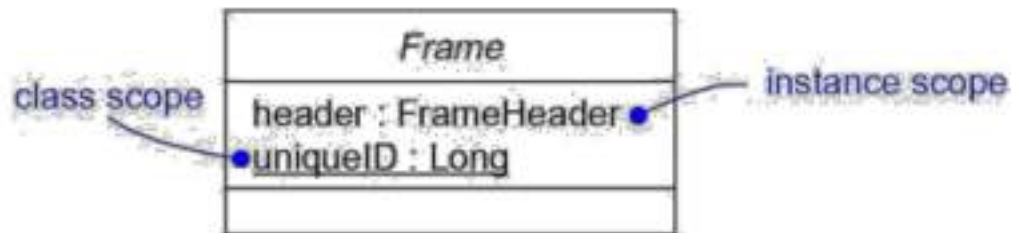
Scope

Another important detail you can specify for a classifier's attributes and operations is its owner scope. The owner scope of a feature specifies whether the feature appears in each instance of the classifier or whether there is just a single instance of the feature for all instances of the classifier. In the UML, you can specify two kinds of owner scope.

1. instance	Each instance of the classifier holds its own value for the feature.
2. classifier	There is just one value of the feature for all instances of the classifier.

As Figure (a simplification of the first figure) shows, a feature that is classifier scoped is rendered by underlining the feature's name. No adornment means that the feature is instance scoped.

Figure Owner Scope

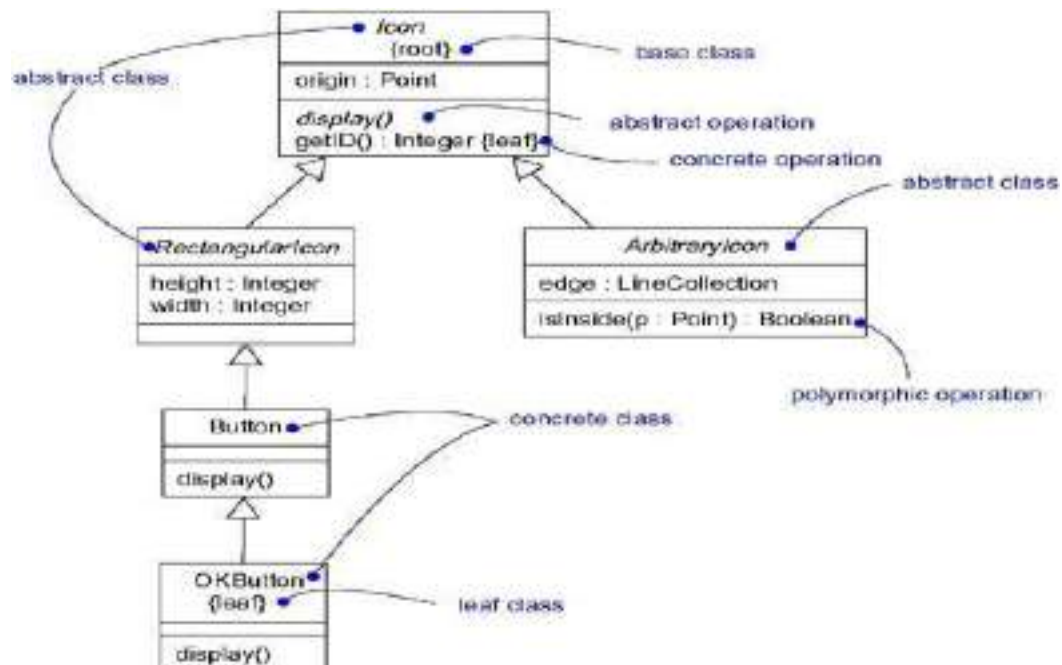


In general, most features of the classifiers you model will be instance scoped. The most common use of classifier scoped features is for private attributes that must be shared among a set of instances (and with the guarantee that no other instances have access to that attribute), such as for generating unique IDs among all instances of a given classifier, and for operations that create instances of the class.

Abstract, Root, Leaf, and Polymorphic Elements

You use generalization relationships to model a lattice of classes, with more-generalized abstractions at the top of the hierarchy and more-specific ones at the bottom. Within these hierarchies, it's common to **specify that certain classes are abstract** meaning that they may not have any direct instances. In the UML, you specify that a class is abstract by writing its name in italics. For example, as Figure shows, *Icon*, *RectangularIcon*, and *ArbitraryIcon* are all abstract classes. By contrast, a concrete class (such as *Button* and *OKButton*) is one that may have direct instances.

Figure Abstract and Concrete Classes and Operations



Whenever you use a class, you'll probably want to inherit features from other, more-general, classes, and have other, more-specific, classes inherit features from it. These are the normal semantics you get from classes in the UML. However, you can also specify that a class may have no children. Such an element is called a leaf class and is specified in the UML by writing the property **leaf** below the class's name. For example, in the figure, **OKButton** is a leaf class, so it may have no children.

Less common but still useful is the ability to specify that a class may have no parents. Such an element is called a root class, and is specified in the UML by writing the property **root** below the class's name. For example, in the figure, **Icon** is a root class. Especially when you have multiple, independent inheritance lattices, it's useful to designate the head of each hierarchy in this manner.

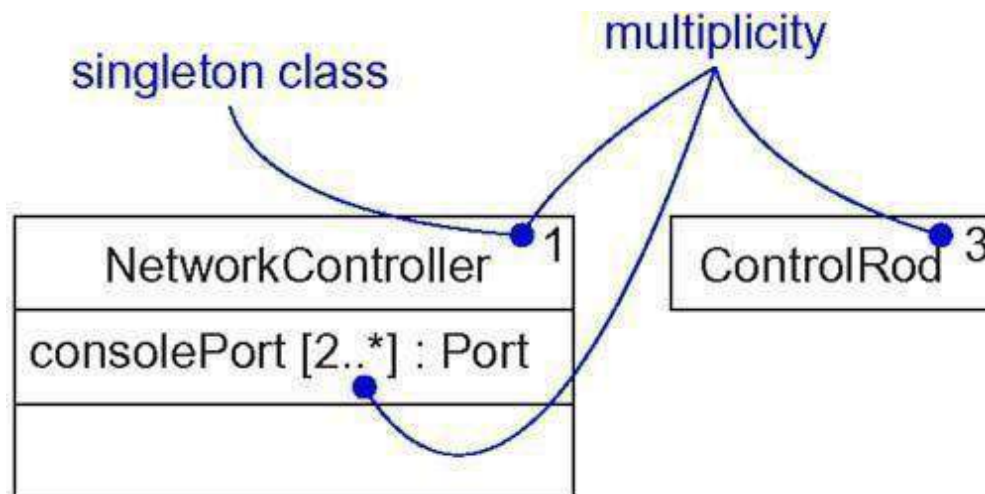
Operations have similar properties. Typically, an operation is polymorphic, which means that, in a hierarchy of classes, you can specify operations with the same signature at different points in the hierarchy. Ones in the child classes override the behavior of ones in the parent classes. When a message is dispatched at run time, the operation in the hierarchy that is invoked is chosen **polymorphically**—that is, a match is determined at run time according to the type of the object. For example, **display** and **isInside** are both polymorphic operations. Furthermore, the operation **Icon::display()** is abstract, meaning that it is incomplete and requires a child to supply an implementation of the operation. In the UML, you specify an abstract operation by writing its name in italics, just as you do for a class. By contrast, **Icon::getID()** is a leaf operation, so designated by the property **leaf**. This means that the operation is not polymorphic and may not be overridden.

Multiplicity

Whenever you use a class, it's reasonable to assume that there may be any number of instances of that class (unless, of course, it is an abstract class and so it may not have any direct instances, although there may be any number of instances of its concrete children). Sometimes, though, you'll want to restrict the number of instances a class may have. Most often, you'll want to specify zero instances (in which case, the class is a utility class that exposes only class-scoped attributes and operations), one instance (a singleton class), a specific number of instances, or many instances (the default case).

The number of instances a class may have is called its multiplicity. Multiplicity is a specification of the range of allowable cardinalities an entity may assume. In the UML, you can specify the multiplicity of a class by writing a multiplicity expression in the upper-right corner of the class icon. For example, in Figure, **NetworkController** is a singleton class. Similarly, there are exactly three instances of the class **ControlRod** in the system.

Figure Multiplicity



Multiplicity applies to attributes, as well. You can specify the multiplicity of an attribute by writing a suitable expression in brackets just after the attribute name. For example, in the figure, there are two or more **consolePort** instances in the instance of **NetworkController**.

Attributes

At the most abstract level, when you model a class's structural features (that is, its attributes), you simply write each attribute's name. That's usually enough information for the average reader to understand the intent of your model.

In its full form, the syntax of an attribute in the UML is

**[visibility] name [multiplicity] [: type] [= initial-value]
[{property-string}]**

For example, the following are all legal attribute declarations:

origin	Name only
+ origin	Visibility and name
origin : Point	Name and type
head : *Item	Name and complex type
name [0..1] : String	Name, multiplicity, and type
origin : Point = (0,0)	Name, type, and initial value
id : Integer {frozen}	Name and property

There are three defined properties that you can use with attributes.

1. changeable	There are no restrictions on modifying the attribute's value.
2. addOnly	For attributes with a multiplicity greater than one, additional values may be added, but once created, a value may not be removed or altered.
3. frozen	The attribute's value may not be changed after the object is initialized.

Unless otherwise specified, attributes are always **changeable**. You'll mainly want to use **frozen** when modeling constant or write-once attributes.

Operations

At the most abstract level, when you model a class's behavioral features (that is, its operations and its signals), you will simply write each operation's name. That's usually enough information for the average reader to understand the intent of your model. As the previous sections have described, however, you can also specify the visibility and scope of each operation. There's still more: You can also specify the parameters, return type, concurrency semantics, and other properties of each operation. Collectively, the name of an operation plus its parameters (including its return type, if any) is called the operation's signature.

In its full form, the syntax of an operation in the UML is

**[visibility] name [(parameter-list)] [: return-type]
[{property-string}]**

For example, the following are all legal operation declarations:

display	Name only
+ display	Visibility and name
set(n : Name, s : String)	Name and parameters
getID() : Integer	Name and return type
restart() {guarded}	Name and property

In an operation's signature, you may provide zero or more parameters, each of which follows the syntax

[direction] name : type [= default-value]

Direction may be any of the following values:

in	An input parameter; may not be modified
out	An output parameter; may be modified to communicate information to the caller
inout	An input parameter; may be modified

In addition to the **leaf** property described earlier, there are four defined properties that you can use with operations.

1. isQuery	Execution of the operation leaves the state of the system unchanged. In other words, the operation is a pure function that has no side effects.
2. sequential	Callers must coordinate outside the object so that only one flow is in the object at a time. In the presence of multiple flows of control, the semantics and integrity of the object cannot be guaranteed.
3. guarded	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by sequentializing all calls to all of the object's guarded operations. In effect, exactly one operation at a time can be invoked on the object, reducing this to sequential semantics.
4. concurrent	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic. Multiple calls from concurrent flows of control may occur simultaneously to one object on any concurrent operation, and all may proceed concurrently with correct semantics;

concurrent operations must be designed so that they perform correctly in the case of a concurrent sequential or guarded operation on the same object.

The last three properties (**sequential**, **guarded**, **concurrent**) address the concurrency semantics of an operation, properties that are relevant only in the presence of active objects, processes, or threads.

Template Classes

A template is a parameterized element. In such languages as C++ and Ada, you can write template classes, each of which defines a family of classes (you can also write template functions, each of which defines a family of functions). A template includes slots for classes, objects, and values, and these slots serve as the template's parameters. You can't use a template directly; you have to instantiate it first. Instantiation involves binding these formal template parameters to actual ones. For a template class, the result is a concrete class that can be used just like any ordinary class.

For example, the following C++ code fragment declares a parameterized **Map** class.

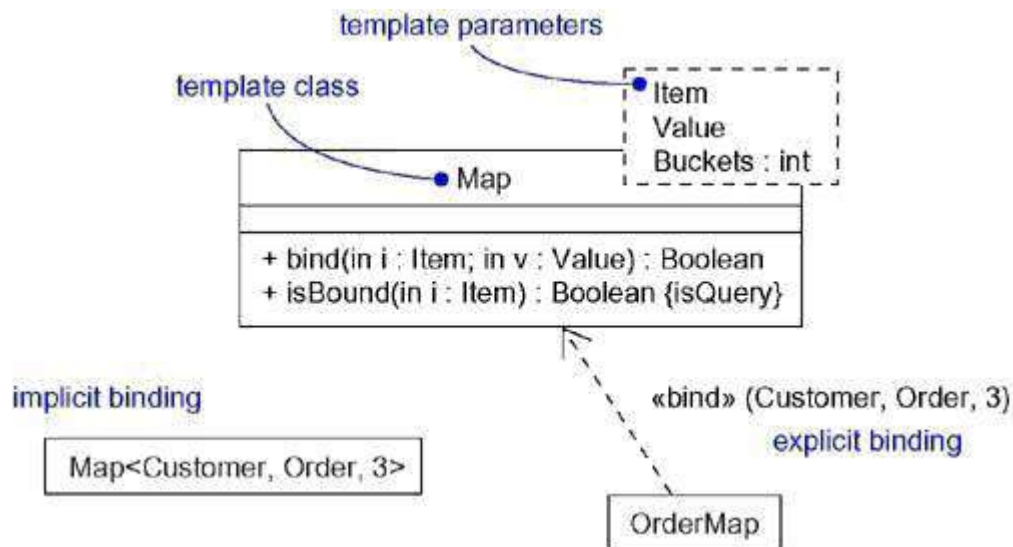
```
template<class Item, class Value, int Buckets> class Map {
public:
    virtual Boolean bind(const Item&, const Value&); virtual
    Boolean isBound(const Item&) const;
    ...
};
```

You might then instantiate this template to map **Customer** objects to **Order** objects.

```
m : Map<Customer, Order, 3>;
```

You can model template classes in the UML as well. As Figure shows, you render a template class just as you do an ordinary class, but with an additional dashed box in the upper-right corner of the class icon, which lists the template parameters.

Figure Template Classes



As the figure goes on to show, you can model the instantiation of a template class in two ways. First, you can do so implicitly, by declaring a class whose name provides the binding. Second, you can do so explicitly, by using a dependency stereotyped as **bind**, which specifies that the source instantiates the target template using the actual parameters.

Standard Elements

All of the UML's extensibility mechanisms apply to classes. Most often, you'll use tagged values to extend class properties (such as specifying the version of a class) and stereotypes to specify new kinds of components (such as model-specific components).

The UML defines four standard stereotypes that apply to classes.

1. metaclass	Specifies a classifier whose objects are all classes
2. powertype	Specifies a classifier whose objects are the children of a given parent
3. stereotype	Specifies that the classifier is a stereotype that may be applied to other elements
4. utility	Specifies a class whose attributes and operations are all class scoped

Common Modeling Techniques

Modeling the Semantics of a Class

The most common purpose for which you'll use classes is to model abstractions that are drawn from the problem you are trying to solve or from the technology you are using to implement a solution to that problem. Once you've identified those abstractions, the next thing you'll need to do is specify their semantics.

To model the semantics of a class, choose among the following possibilities, arranged from informal to formal.

- Specify the responsibilities of the class. A responsibility is a contract or obligation of a type or class and is rendered in a note (stereotyped as **responsibility**) attached to the class, or in an extra compartment in the class icon.
- Specify the semantics of the class as a whole using structured text, rendered in a note (stereotyped as **semantics**) attached to the class.
- Specify the body of each method using structured text or a programming language, rendered in a note attached to the operation by a dependency relationship.
- Specify the pre- and postconditions of each operation, plus the invariants of the class as a whole, using structured text. These elements are rendered in notes (stereotyped as **precondition**, **postcondition**, and **invariant**) attached to the operation or class by a dependency relationship.
- Specify a state machine for the class. A state machine is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events.
- Specify a collaboration that represents the class. A collaboration is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. A collaboration has a structural part, as well as a dynamic part, so you can use collaborations to specify all dimensions of a class's semantics.

- Specify the pre- and postconditions of each operation, plus the invariants of the class as a whole, using a formal language such as OCL.

Advanced Relationships

Terms and Concepts

A *relationship* is a connection among things. In object-oriented modeling, the four most important relationships are dependencies, generalizations, associations, and realizations. Graphically, a relationship is rendered as a path, with different kinds of lines used to distinguish the different relationships.

Dependency

A *dependency* is a using relationship, specifying that a change in the specification of one thing (for example, class **SetTopController**) may affect another thing that uses it (for example, class **ChannelIterator**), but not necessarily the reverse. Graphically, a dependency is rendered as a dashed line, directed to the thing that is depended on. Apply dependencies when you want to show one thing using another.

A plain, unadorned dependency relationship is sufficient for most of the using relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines a number of stereotypes that may be applied to dependency relationships. There are 17 such stereotypes, all of which can be organized into six groups.

First, there are eight stereotypes that apply to dependency relationships among classes and objects in class diagrams.

1. specifies	Specifies that the source instantiates the target template using the given actual bind parameters
---------------------	--

You'll use **bind** when you want to model the details of template classes. For example, the relationship between a template container class and an instantiation of that class would be modeled as a **bind** dependency. **Bind** includes a list of actual arguments that map to the formal arguments of the template.

2. derive	Specifies that the source may be computed from the target
------------------	---

You'll use **derive** when you want to model the relationship between two attributes or two associations, one of which is concrete and the other is conceptual. For example, a **Person** class might have the attribute **BirthDate** (which is concrete), as well as the attribute **Age** (which can be derived from **BirthDate**, so is not separately manifest in the class). You'd show the relationship between **Age** and **BirthDate** by using a **derive** dependency, showing **Age** derived from **BirthDate**.

3. friend	Specifies that the source is given special visibility into the target
------------------	---

You'll use **friend** when you want to model relationships such as found with C++ friend classes.

4. instanceOf	Specifies that the source object is an instance of the target classifier
5. instantiate	Specifies that the source creates instances of the target

These last two stereotypes let you model class/object relationships explicitly. You'll use **instanceOf** when you want to model the relationship between a class and an object in the same diagram, or between a class and its metaclass. You'll use **instantiate** when you want to specify which element creates objects of

another.

- | | |
|----|--|
| 6. | Specifies that the target is a powertype of the source; a powertype is a classifier whose objects are all the children of a given parent |
|----|--|

You'll use **powertype** when you want to model classes that cover other classes, such as you'll find when modeling databases.

- | | |
|------------------|---|
| 7. refine | Specifies that the source is at a finer degree of abstraction than the target |
|------------------|---|

You'll use **refine** when you want to model classes that are essentially the same but at different levels of abstraction. For example, during analysis, you might encounter a **Customer** class which, during design, you refine into a more detailed **Customer** class, complete with its implementation.

- | | |
|----|--|
| 8. | Specifies that the semantics of the source element depends on the semantics of the use public part of the target |
|----|--|

You'll apply **use** when you want to explicitly mark a dependency as a using relationship, in contrast to the shades of dependencies other stereotypes provide.

Continuing, there are two stereotypes that apply to dependency relationships among packages.

- | | |
|----|---|
| 1. | Specifies that the source package is granted the right to reference the elements of the access target package |
| 2. | A kind of access that specifies that the public contents of the target package enter the |

You'll use **access** and **import** when you want to model the relationships among packages. Between two peer packages, the elements in one cannot reference the elements in the other unless there's an explicit **access** or **import** dependency. For example, suppose a target package **T** contains the class **C**. If you specify an access dependency from **S** to **T**, then the elements of **S** can reference **C**, using the fully qualified name **T::C**. If you specify an import dependency from **S** to **T**, then the elements of **S** can reference **C** using just its simple name.

- | | |
|------------------|--|
| 1. extend | Specifies that the target use case extends the behavior of the source |
| 2. | Specifies that the source use case explicitly incorporates the behavior of another include use case at a location specified by the source |

You'll use **extend** and **include** (and simple generalization) when you want to decompose use cases into reusable parts.

- | | |
|----------------|---|
| 1. | Specifies that the target is the same object as the source but at a later point in time |
| become | and with possibly different values, state, or roles |
| 2. call | Specifies that the source operation invokes the target operation |
| 3. copy | Specifies that the target object is an exact, but independent, copy of the source |

You'll use **become** and **copy** when you want to show the role, state, or attribute value of one object at different points in time or space. You'll use **call** when you want to model the calling dependencies among operations.

One stereotype you'll encounter in the context of state machines is

?send	Specifies that the source operation sends the target event
--------------	--

You'll use **send** when you want to model an operation (such as found in the action associated with a state transition) dispatching a given event to a target object (which in turn might have an associated state machine). The **send** dependency in effect lets you tie independent state machines together.

Finally, one stereotype that you'll encounter in the context of organizing the elements of your system into subsystems and models is

?trace	Specifies that the target is an historical ancestor of the source
---------------	---

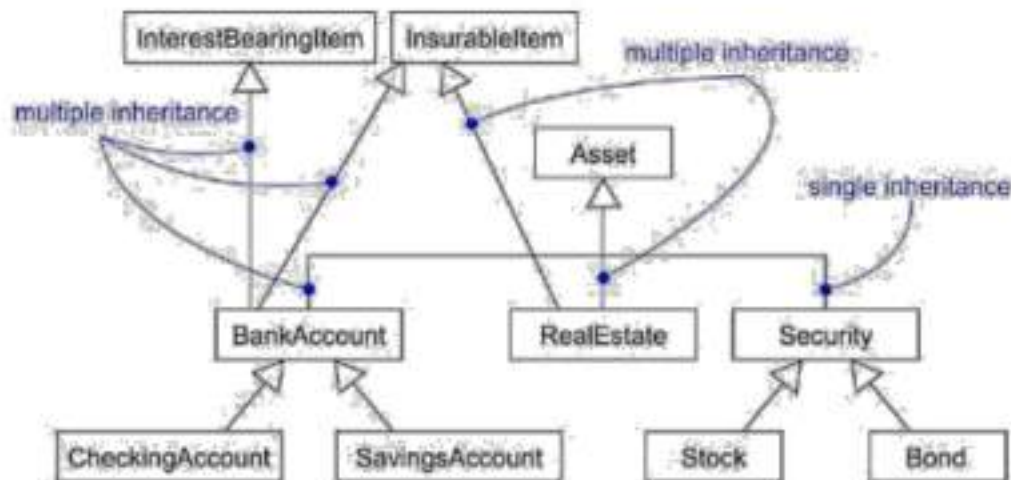
You'll use **trace** when you want to model the relationships among elements in different models. For example, in the context of a system's architecture, a use case in a use case model (representing a functional requirement) might trace to a package in the corresponding design model (representing the artifacts that realize that use case).

Generalization

A *generalization* is a relationship between a general thing (called the superclass or parent) and a more specific kind of that thing (called the subclass or child). For example, you might encounter the general class **Window** with its more specific kind, **MultiPaneWindow**. With a generalization relationship from the child to the parent, the child (**MultiPaneWindow**) will inherit all the structure and behavior of the parent (**Window**). The child may even add new structure and behavior, or it may modify the behavior of the parent. In a generalization relationship, instances of the child may be used anywhere instances of the parent apply • meaning that the child is substitutable for the parent.

Most of the time, you'll find single inheritance sufficient. A class that has exactly one parent is said to use single inheritance. There are times, however, when multiple inheritance is better, and you can model those relationships, as well, in the UML. For example, Figure shows a set of classes drawn from a financial services application. You see the class **Asset** with three children: **BankAccount**, **RealEstate**, and **Security**. Two of these children (**BankAccount** and **Security**) have their own children. For example, **Stock** and **Bond** are both children of **Security**.

Figure Multiple Inheritance



Two of these children (**BankAccount** and **RealEstate**) inherit from multiple parents. **RealEstate**, for example, is a kind of **Asset**, as well as a kind of **InsurableItem**, and **BankAccount** is a kind of **Asset**, as well as a kind of **InterestBearingItem** and an **InsurableItem**.

Parents, such as **InterestBearingItem** and **InsurableItem**, are called mixins because they don't stand alone but, rather, are intended to be mixed in with other parents (such as **Asset**) to form children from these various bits of structure and behavior.

A plain, unadorned generalization relationship is sufficient for most of the inheritance relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines one stereotype and four constraints that may be applied to generalization relationships.

First, there is the one stereotype.

? implementation	Specifies that the child inherits the implementation of the parent but does not make public nor support its interfaces, thereby violating substitutability
-------------------------	--

You'll use **implementation** when you want to model private inheritance, such as found in C++.

Next, there are four standard constraints that apply to generalization relationships.

1. complete	Specifies that all children in the generalization have been specified in the model (although some may be elided in the diagram) and that no additional children are permitted
2. incomplete	Specifies that not all children in the generalization have been specified (even if some are elided) and that additional children are permitted

Unless otherwise stated, you can assume that any diagram shows only a partial view of an inheritance lattice and so is elided. However, elision is different from the completeness of a model. Specifically, you'll use the **complete** constraint when you want to show explicitly that you've fully specified a hierarchy in the model (although no one diagram may show that hierarchy); you'll use **incomplete** to show explicitly that you have not stated the full specification of the hierarchy in the model (although one diagram may show everything in the model).

3. disjoint	Specifies that objects of the parent may have no more than one of the children as a type
--------------------	--

4. overlapping	Specifies that objects of the parent may have more than one of the children as a type
--------------------------	---

These two constraints apply only in the context of multiple inheritance. You'll use **disjoint** and **overlapping** when you want to distinguish between static classification (**disjoint**) and dynamic classification (**overlapping**).

Association

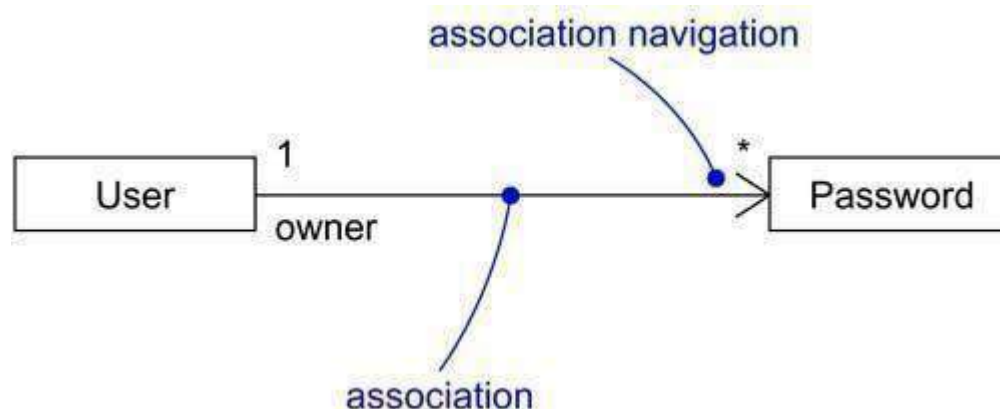
An *association* is a structural relationship, specifying that objects of one thing are connected to objects of another. For example, a **Library** class might have a one-to-many association to a **Book** class, indicating that each **Book** instance is owned by one **Library** instance. Furthermore, given a **Book**, you can find its owning **Library**, and given a **Library**, you can navigate to all its **Books**. Graphically, an association is rendered as a solid line connecting the same or different classes. You use associations when you want to show structural relationships.

There are four basic adornments that apply to an association: a name, the role at each end of the association, the multiplicity at each end of the association, and aggregation. For advanced uses, there are a number of other properties you can use to model subtle details, such as navigation, qualification, and various flavors of aggregation.

Navigation

Given a plain, unadorned association between two classes, such as **Book** and **Library**, it's possible to navigate from objects of one kind to objects of the other kind. Unless otherwise specified, navigation across an association is bidirectional. However, there are some circumstances in which you'll want to limit navigation to just one direction. For example, as Figure 10-3 shows, when modeling the services of an operating system, you'll find an association between **User** and **Password** objects. Given a **User**, you'll want to be able to find the corresponding **Password** objects; but given a **Password**, you don't want to be able to identify the corresponding **User**. You can explicitly represent the direction of navigation by adorning an association with an arrowhead pointing to the direction of traversal.

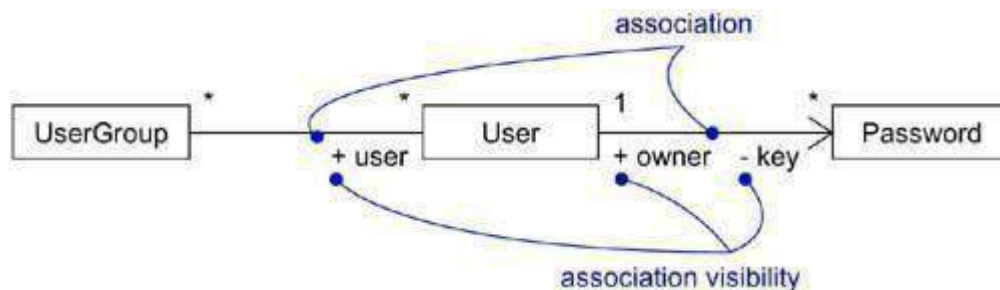
Figure Navigation



Visibility

Given an association between two classes, objects of one class can see and navigate to objects of the other, unless otherwise restricted by an explicit statement of navigation. However, there are circumstances in which you'll want to limit the visibility across that association relative to objects outside the association. For example, as Figure shows, there is an association between **UserGroup** and **User** and another between **User** and **Password**. Given a **User** object, it's possible to identify its corresponding **Password** objects. However, a **Password** is private to a **User**, so it shouldn't be accessible from the outside (unless, of course, the **User** explicitly exposes access to the **Password**, perhaps through some public operation). Therefore, as the figure shows, given a **UserGroup** object, you can navigate to its **User** objects (and vice versa), but you cannot in turn see the **User** object's **Password** objects; they are private to the **User**. In the UML, you can specify three levels of visibility for an association end, just as you can for a class's features by appending a visibility symbol to a role name. Unless otherwise noted, the visibility of a role is public. Private visibility indicates that objects at that end are not accessible to any objects outside the association; protected visibility indicates that objects at that end are not accessible to any objects outside the association, except for children of the other end.

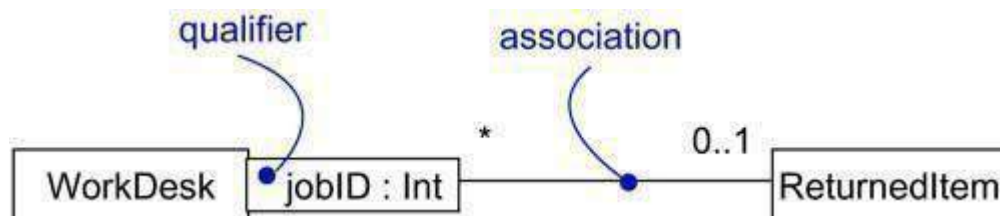
Figure Visibility



Qualification

In the context of an association, one of the most common modeling idioms you'll encounter is the problem of lookup. Given an object at one end of an association, how do you identify an object or set of objects at the other end? For example, consider the problem of modeling a work desk at a manufacturing site at which returned items are processed to be fixed. As Figure 10-5 shows, you'd model an association between two classes, **WorkDesk** and **ReturnedItem**. In the context of the **WorkDesk**, you'd have a **jobId** that would identify a particular **ReturnedItem**. In that sense, **jobId** is an attribute of the association. It's not a feature of **ReturnedItem** because items really have no knowledge of things like repairs or jobs. Then, given an object of **WorkDesk** and given a particular value for **jobId**, you can navigate to zero or one objects of **ReturnedItem**. In the UML, you'd model this idiom using a qualifier, which is an association attribute whose values partition the set of objects related to an object across an association. You render a qualifier as a small rectangle attached to the end of an association, placing the attributes in the rectangle, as the figure shows. The source object, together with the values of the qualifier's attributes, yield a target object (if the target multiplicity is at most one) or a set of objects (if the target multiplicity is many).

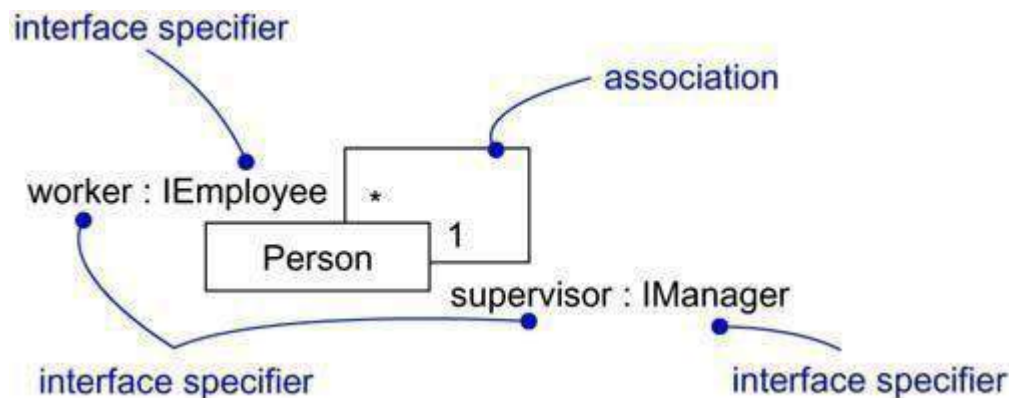
Figure Qualification



Interface Specifier

An interface is a collection of operations that are used to specify a service of a class or a component; every class may realize many interfaces. Collectively, the interfaces realized by a class represent a complete specification of the behavior of that class. However, in the context of an association with another target class, a source class may choose to present only part of its face to the world. For example, in the vocabulary of a human resources system, a **Person** class may realize many interfaces: **IManager**, **IEmployee**, **IOfficer**, and so on. As Figure shows, you can model the relationship between a supervisor and her workers with a one-to-many association, explicitly labeling the roles of this association as **supervisor** and **worker**. In the context of this association, a **Person** in the role of **supervisor** presents only the **IManager** face to the **worker**; a **Person** in the role of **worker** presents only the **IEmployee** face to the **supervisor**. As the figure shows, you can explicitly show the type of role using the syntax **rolename : iname**, where **iname** is some interface of the other classifier.

Figure Interface Specifiers



Composition

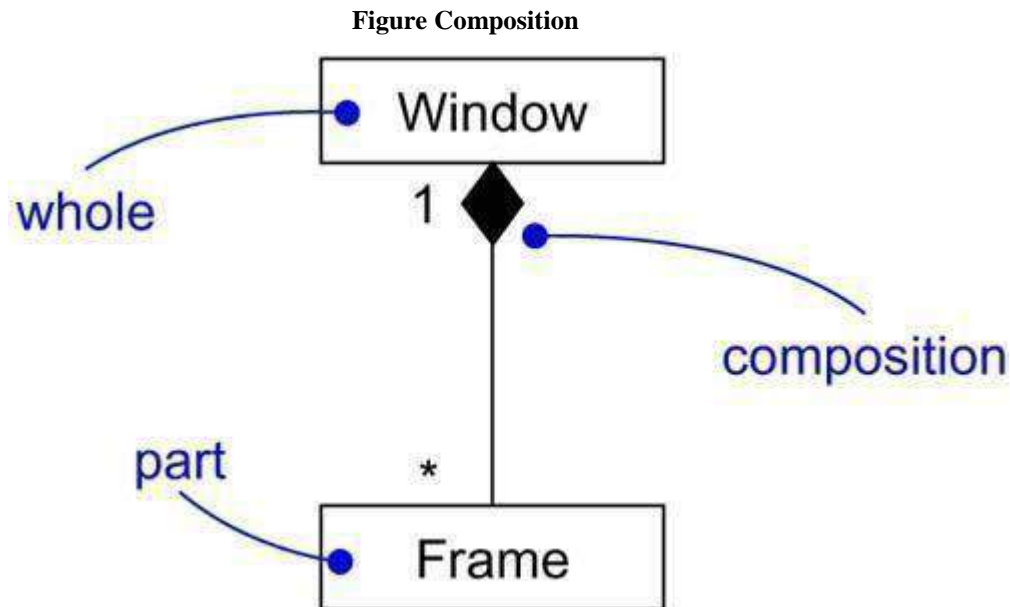
Aggregation turns out to be a simple concept with some fairly deep semantics. Simple aggregation is entirely conceptual and does nothing more than distinguish a "whole" from a "part." Simple aggregation does not change the meaning of navigation across the association between the whole and its parts, nor does it link the lifetimes of the whole and its parts.

However, there is a variation of simple aggregation• composition• that does add some important semantics. Composition is a form of aggregation, with strong ownership and coincident lifetime as part of the whole. Parts with non-fixed multiplicity may be created after the composite itself, but once created they live and die with it. Such parts can also be explicitly removed before the death of the composite.

This means that, in a composite aggregation, an object may be a part of only one composite at a time. For example, in a windowing system, a **Frame** belongs to exactly one **Window**. This is in contrast to simple aggregation, in which a part may be shared by several wholes. For example, in the model of a house, a **Wall** may be a part of one or more **Room** objects.

In addition, in a composite aggregation, the whole is responsible for the disposition of its parts, which means that the composite must manage the creation and destruction of its parts. For example, when you create a **Frame** in a windowing system, you must attach it to an enclosing **Window**. Similarly, when you destroy the **Window**, the **Window** object must in turn destroy its **Frame** parts.

As Figure shows, composition is really just a special kind of association and is specified by adorning a plain association with a filled diamond at the whole end.

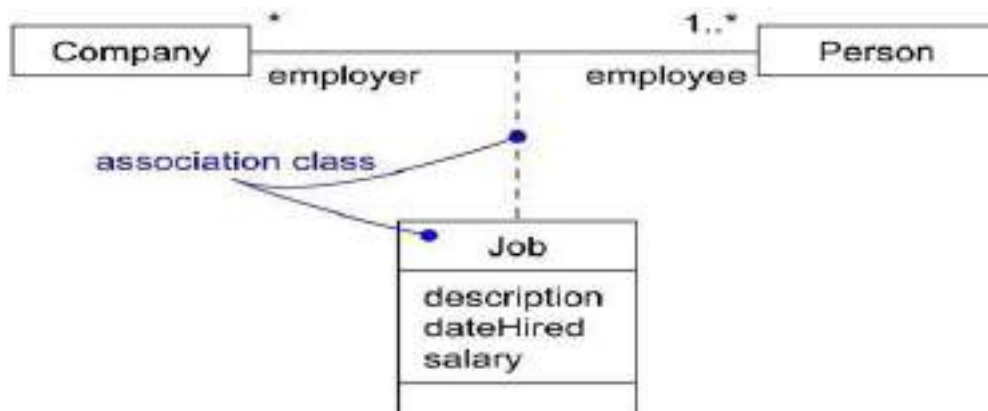


Association Classes

In an association between two classes, the association itself might have properties. For example, in an employer/employee relationship between a **Company** and a **Person**, there is a **Job** that represents the properties of that relationship that apply to exactly one pairing of the **Person** and **Company**. It wouldn't be appropriate to model this situation with a **Company** to **Job** association together with a **Job** to **Person** association. That wouldn't tie a specific instance of the **Job** to the specific pairing of **Company** and **Person**.

In the UML, you'd model this as an association class, which is a modeling element that has both association and class properties. An association class can be seen as an association that also has class properties, or as a class that also has association properties. You render an association class as a class symbol attached by a dashed line to an association as in Figure .

Figure Association Classes



Constraints

These simple and advanced properties of associations are sufficient for most of the structural relationships you'll encounter. However, if you want to specify a shade of meaning, the UML defines five constraints that may be applied to association relationships.

First, you can distinguish if the association is real or conceptual.

1. implicit	Specifies that the relationship is not manifest but, rather, is only conceptual
--------------------	---

For example, if you have an association between two base classes, you can specify that same association between two children of those base classes (because they inherit the relationships of the parent classes). You'd mark it as **implicit**, because it's not manifest separately but, rather, is implicit from the relationship that exists between the parent classes.

Second, you can specify that the objects at one end of an association (with a multiplicity greater than one) are ordered or unordered.

2. ordered	Specifies that the set of objects at one end of an association are in an explicit order
-------------------	---

For example, in a **User/Password** association, the **Passwords** associated with the **User** might be kept in a least-recently used order, and would be marked as **ordered**.

Next, there are three properties, defined using constraint notation, that relate to the changeability of the instances of an association.

Finally, there are three defined constraints that relate to the changeability of the instances of an association.

3. changeable	Links between objects may be added, removed, and changed freely
4. addOnly	New links may be added from an object on the opposite end of the association
5. frozen	A link, once added from an object on the opposite end of the association, may not be modified or deleted

Finally, there is one constraint for managing related sets of associations:

6.	Specifies that, over a set of associations, exactly one is manifest for each associated xor object
----	---

Realization

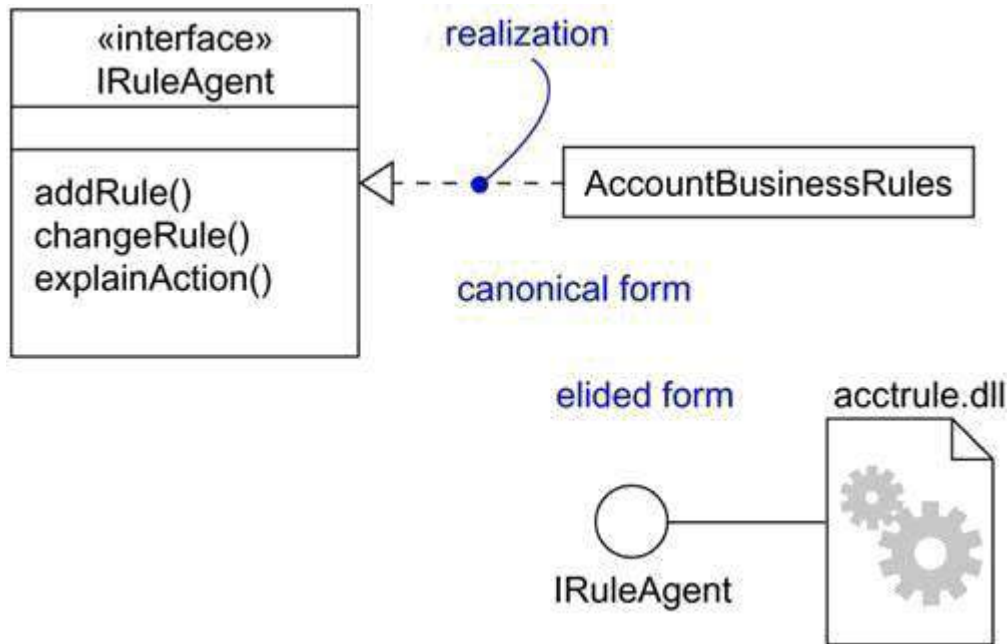
A *realization* is a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out. Graphically, a realization is rendered as a dashed directed line with a large open arrowhead pointing to the classifier that specifies the contract.

Realization is sufficiently different from dependency, generalization, and association relationships that it is treated as a separate kind of relationship. Semantically, realization is somewhat of a cross between dependency and generalization, and its notation is a combination of the notation for dependency and generalization. You'll use realization in two circumstances: in the context of interfaces and in the context of collaborations.

Most of the time, you'll use realization to specify the relationship between an interface and the class or component that provides an operation or service for it. An interface is a collection of operations that are used to specify a service of a class or a component. Therefore, an interface specifies a contract that a class or component must carry out. An interface may be realized by many such classes or components, and a class or component may realize many interfaces. Perhaps the most interesting thing about

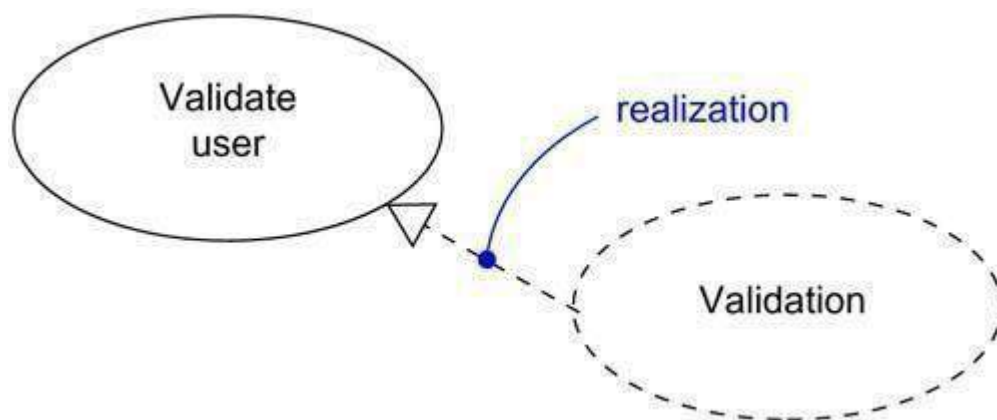
interfaces is that they let you separate the specification of a contract (the interface itself) from its implementation (by a class or a component). Furthermore, interfaces span the logical and physical parts of a system's architecture. For example, as Figure 10-9 shows, a class (such as **AccountBusinessRules** in an order entry system) in a system's design view might realize a given interface (such as **IRuleAgent**). That same interface (**IRuleAgent**) might also be realized by a component (such as **acctrule.dll**) in the system's implementation view. Note that you can represent realization in two ways: in the canonical form (using the **interface** stereotype and the dashed directed line with a large open arrowhead) and in an elided form (using the interface lollipop notation).

Figure Realization of an Interface



You'll also use realization to specify the relationship between a use case and the collaboration that realizes that use case, as Figure shows. In this circumstance, you'll almost always use the canonical form of realization.

Figure Realization of a Use Case



Modeling Webs of Relationships

When you model these webs of relationships,

- Don't begin in isolation. Apply use cases and scenarios to drive your discovery of the relationships among a set of abstractions.
- In general, start by modeling the structural relationships that are present. These reflect the static view of the system and are therefore fairly tangible.
- Next, identify opportunities for generalization/specialization relationships; use multiple inheritance sparingly.
- Only after completing the preceding steps should you look for dependencies; they generally represent more-subtle forms of semantic connection.
- For each kind of relationship, start with its basic form and apply advanced features only as absolutely necessary to express your intent.
- Remember that it is both undesirable and unnecessary to model all relationships among a set of abstractions in a single diagram or view. Rather, build up your system's relationships by considering different views on the system. Highlight interesting sets of relationships in individual diagrams.

Interfaces, Types, and Roles

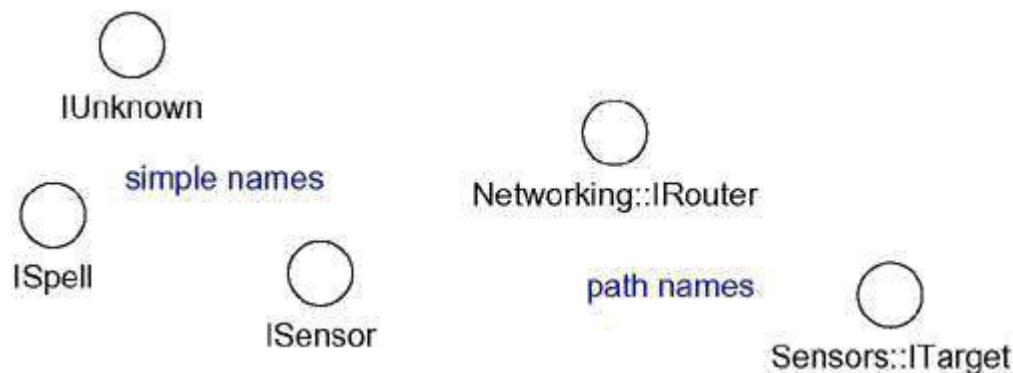
Terms and Concepts

An *interface* is a collection of operations that are used to specify a service of a class or a component. A *type* is a stereotype of a class used to specify a domain of objects, together with the operations (but not the methods) applicable to the object. A *role* is the behavior of an entity participating in a particular context. Graphically, an interface is rendered as a circle; in its expanded form, an interface may be rendered as a stereotyped class in order to expose its operations and other properties.

Names

Every interface must have a name that distinguishes it from other interfaces. A *name* is a textual string. That name alone is known as a *simple name*; a *path name* is the interface name prefixed by the name of the package in which that interface lives. An interface may be drawn showing only its name, as in [Figure](#)

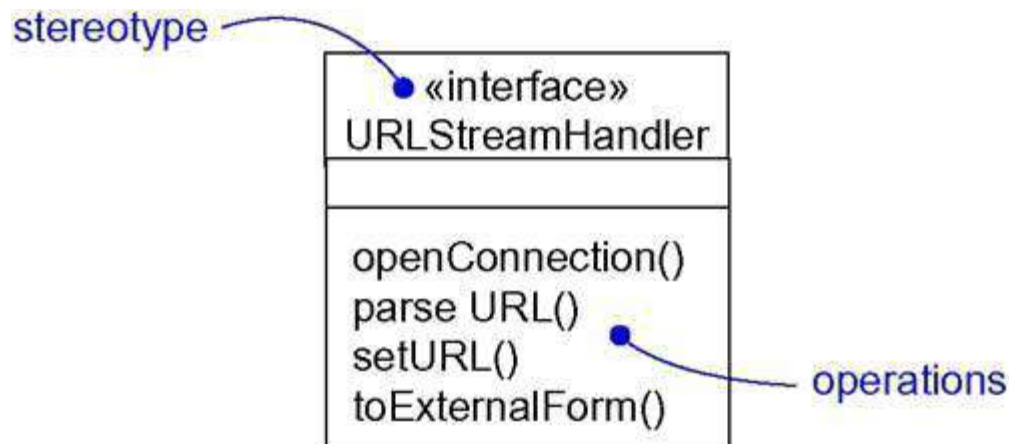
Figure Simple and Path Names



Operations

An interface is a named collection of operations used to specify a service of a class or of a component. Unlike classes or types, interfaces do not specify any structure (so they may not include any attributes), nor do they specify any implementation (so they may not include any methods, which provide the implementation of an operation). Like a class, an interface may have any number of operations. These operations may be adorned with visibility properties, concurrency properties, stereotypes, tagged values, and constraints.

Figure Operations



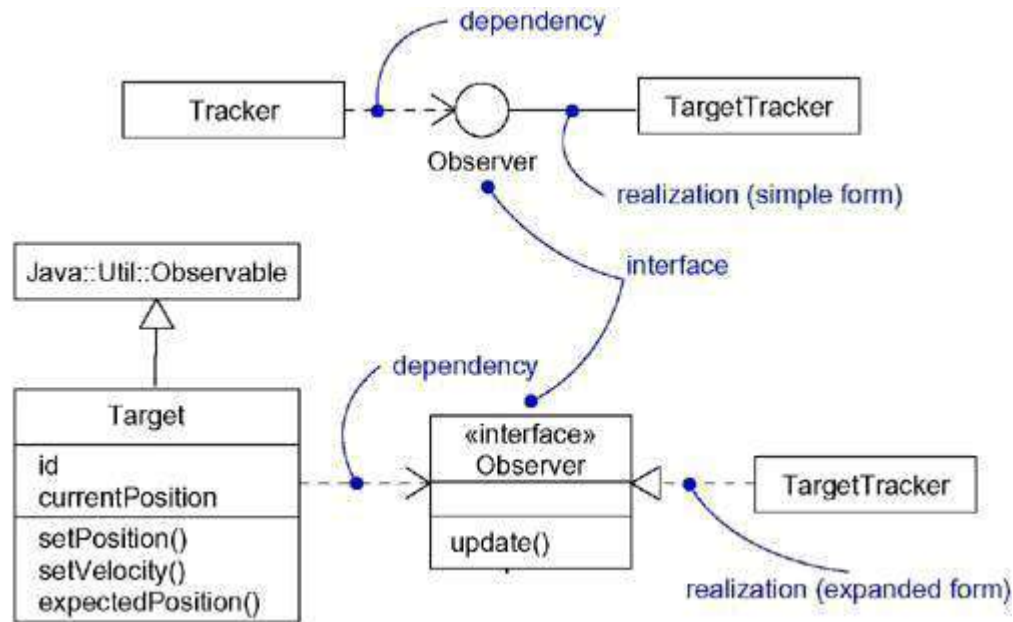
Relationships

Like a class, an interface may participate in generalization, association, and dependency relationships. In addition, an interface may participate in realization relationships. Realization is a semantic relationship between two classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.

An interface specifies a contract for a class or a component without dictating its implementation. A class or component may realize many interfaces. In so doing, it commits to carry out all these contracts faithfully, which means that it provides a set of methods that properly implement the operations defined in the interface. Similarly, a class or a component may depend on many interfaces. In so doing, it expects that these contracts will be honored by some set of components that realize them. This is why we say that an interface represents a seam in a system. An interface specifies a contract, and the client and the supplier on each side of the contract may change independently, as long as each fulfills its obligations to the contract.

As Figure illustrates, you can show that an element realizes an interface in two ways. First, you can use the simple form in which the interface and its realization relationship are rendered as a lollipop sticking off to one side of a class or component. This form is useful when you simply want to expose the seams in your system. However, the limitation of this style is that you can't directly visualize the operations or signals provided by the interface. Second, you can use the expanded form in which you render an interface as a stereotyped class, which allows you to visualize its operations and other properties, and then draw a realization relationship from the classifier or component to the interface. In the UML, a realization relationship is rendered as a dashed directed line with a large open arrowhead pointing to the interface. This notation is a cross between generalization and dependency.

Figure Realizations



In both cases, you attach the class or component that builds on an interface with a dependency relationship from the element to the interface.

Understanding an Interface

When you are handed an interface, the first thing you'll see is a set of operations that specify a service of a class or a component. Look a little deeper and you'll see the full signature of those operations, along with any of their special properties, such as visibility, scope, and concurrency semantics.

These properties are important, but for complex interfaces, they aren't enough to help you understand the semantics of the service they represent, much less know how to use those operations properly. In the absence of any other information, you'd have to dive into some abstraction that realizes the interface to figure out what each operation does and how those operations are meant to work together. However, that defeats the purpose of an interface, which is to provide a clear separation of concerns in a system.

In the UML, you can supply much more information to an interface in order to make it understandable and approachable. First, you may attach pre- and postconditions to each operation and invariants to the class or component as a whole. By doing this, a client who needs to use an interface will be able to understand what the interface does and how to use it, without having to dive into an implementation. If you need to be rigorous, you can use the UML's OCL to formally specify the semantics. Second, you can attach a state machine to the interface. You can use this state machine to specify the legal partial ordering of an interface's operations. Third, you can attach collaborations to the interface. You can use collaborations to specify the expected behavior of the interface through a series of interaction diagrams.

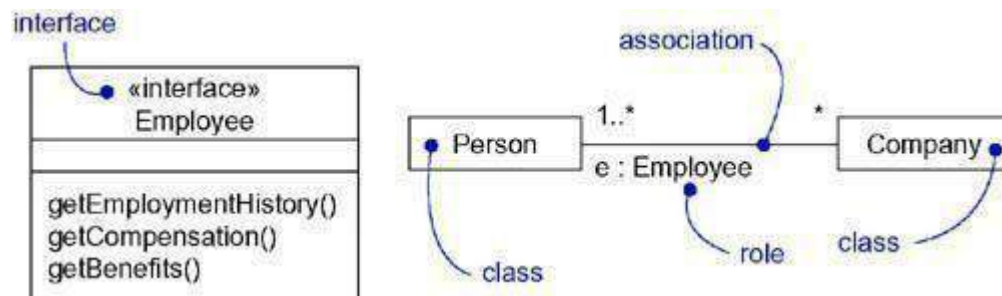
Types and Roles

A class may realize many interfaces. An instance of that class must therefore support all those interfaces, because an interface defines a contract, and any abstraction that conforms to that interface must, by definition, faithfully carry out that contract. Nonetheless, in a given context, an instance may present only one or more of its interfaces as relevant. In that case, each interface represents a role that the object plays. A role names a behavior of an entity participating in a particular context. Stated another way, a role is the face that an abstraction presents to the world.

For example, consider an instance of the class **Person**. Depending on the context, that **Person** instance may play the role of **Mother**, **Comforter**, **PayerOfBills**, **Employee**, **Customer**, **Manager**, **Pilot**, **Singer**, and so on. When an object plays a particular role, it presents a face to the world, and clients that interact with it expect a certain behavior depending on the role that it plays at the time. For example, an instance of **Person** in the role of **Manager** would present a different set of properties than if the instance were playing the role of **Mother**.

In the UML, you can specify a role an abstraction presents to another abstraction by adorning the name of an association end with a specific interface. For example, Figure shows the interface **Employee**, whose definition includes three operations. There exists an association between the classes **Person** and **Company** in which context **Person** plays the role **e**, whose type is **Employee**. In a different association, the **Person** might present an entirely different face to the world. With this explicit type, the role the **Person** plays is more than just a name meaningful to the human reader of this diagram. In the UML, this means that the **Person** presents the role of **Employee** to the **Company**, and in that context, only the properties specified by **Employee** are visible and relevant to the **Company**.

Figure Roles



A class diagram like this one is useful for modeling the static binding of an abstraction to its interface. You can model the dynamic binding of an abstraction to its interface by using the **become** stereotype in an interaction diagram, showing an object changing from one role to another.

If you want to formally model the semantics of an abstraction and its conformance to a specific interface, you'll want to use the defined stereotype **type**. **Type** is a stereotype of class, and you use it to specify a domain of objects, together with the operations (but not the methods) applicable to the objects of that type. The concept of type is closely related to that of interface, except that a type's definition may include attributes while an interface may not. If you want to show that an abstraction is statically typed, you'll want to use **implementationClass**, a stereotype of class that specifies a class whose instances are statically typed (unlike **Person** in the example above) and that defines the physical data structure and methods of an object as implemented in traditional programming languages.

Common Modeling Techniques

Modeling the Seams in a System

The most common purpose for which you'll use interfaces is to model the seams in a system composed of software components, such as COM+ or Java Beans. You'll reuse some components from other systems or buy off the shelf; you will create others from scratch. In any case, you'll need to write glue code that weaves these components together. That requires you to understand the interfaces provided and relied on by each component.

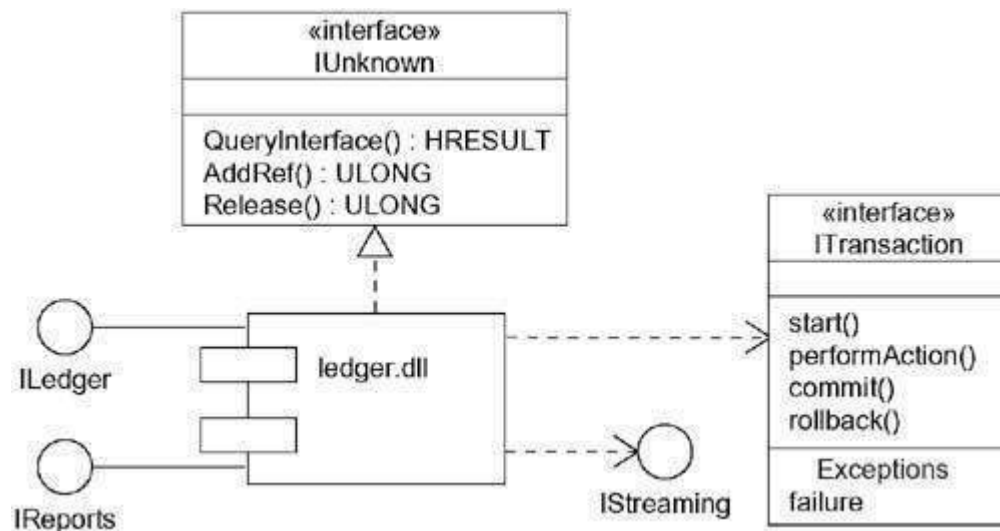
Identifying the seams in a system involves identifying clear lines of demarcation in your architecture. On either side of those lines, you'll find components that may change independently, without affecting the components on the other side, as long as the components on both sides conform to the contract specified by that interface.

To model the seams in a system,

- Within the collection of classes and components in your system, draw a line around those that tend to be tightly coupled relative to other sets of classes and components.
- Refine your grouping by considering the impact of change. Classes or components that tend to change together should be grouped together as collaborations.
- Consider the operations and the signals that cross these boundaries, from instances of one set of classes or components to instances of other sets of classes and components.
- Package logically related sets of these operations and signals as interfaces.
- For each such collaboration in your system, identify the interfaces it relies on (imports) and those it provides to others (exports). You model the importing of interfaces by dependency relationships, and you model the exporting of interfaces by realization relationships.
- For each such interface in your system, document its dynamics by using pre- and postconditions for each operation, and use cases and state machines for the interface as a whole.

For example, Figure 11-6 shows the seams surrounding a component (the library **ledger.dll**) drawn from a financial system. This component realizes three interfaces: **IUnknown**, **ILedger**, and **IReports**. In this diagram, **IUnknown** is shown in its expanded form; the other two are shown in their simple form, as lollipops. These three interfaces are realized by **ledger.dll** and are exported to other components for them to build on.

Figure Modeling the Seams in a System



As this diagram also shows, **ledger.dll** imports two interfaces, **IStreaming** and **ITransaction**, the latter of which is shown in its expanded form. These two interfaces are required by the **ledger.dll** component for its proper operation. Therefore, in a running system, you must supply components that realize these two interfaces. By identifying interfaces such as **ITransaction**, you've effectively decoupled the components on either side of the interface, permitting you to employ any component that conforms to that interface.

Interfaces such as **ITransaction** are more than just a pile of operations. This particular interface makes some assumptions about the order in which its operations should be called. Although not shown here, you could attach use cases to this interface and enumerate the common ways you'd use it.

Modeling Static and Dynamic Types

Most object-oriented programming languages are statically typed, which means that the type of an object is bound at the time the object is created. Even so, that object will likely play different roles over time. This means that clients that use that object interact with the object through different sets of interfaces, representing interesting, possibly overlapping, sets of operations.

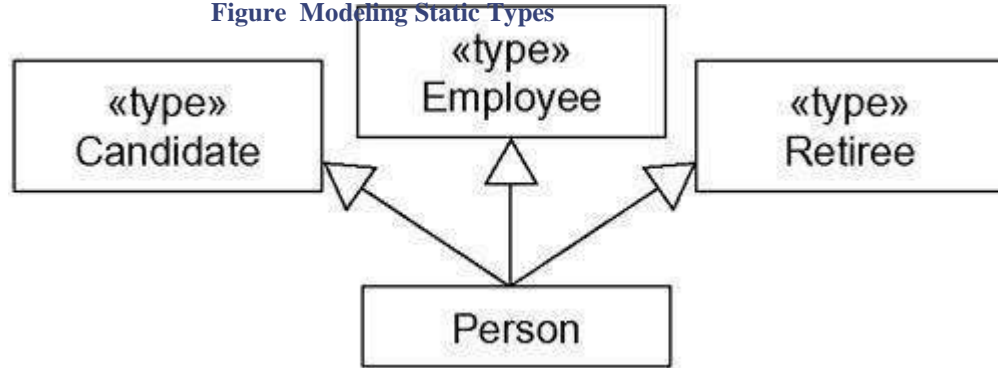
Modeling the static nature of an object can be visualized in a class diagram. However, when you are modeling things like business objects, which naturally change their roles throughout a workflow, it's sometimes useful to explicitly model the dynamic nature of that object's type. In these circumstances, an object can gain and lose types during its life.

To model a dynamic type,

- Specify the different possible types of that object by rendering each type as a class stereotyped as **type** (if the abstraction requires structure and behavior) or as **interface** (if the abstraction requires only behavior).
- Model all the roles the class of the object may take on at any point in time. You can do so in two ways:
 1. First, in a class diagram, explicitly type each role that the class plays in its association with other classes. Doing this specifies the face instances of that class put on in the context of the associated object.
 2. Second, also in a class diagram, specify the class-to-type relationships using generalization.
- In an interaction diagram, properly render each instance of the dynamically typed class. Display the role of the instance in brackets below the object's name.
- To show the change in role of an object, render the object once for each role it plays in the interaction, and connect these objects with a message stereotyped as **become**.

For example, Figure shows the roles that instances of the class **Person** might play in the context of a human resources system.

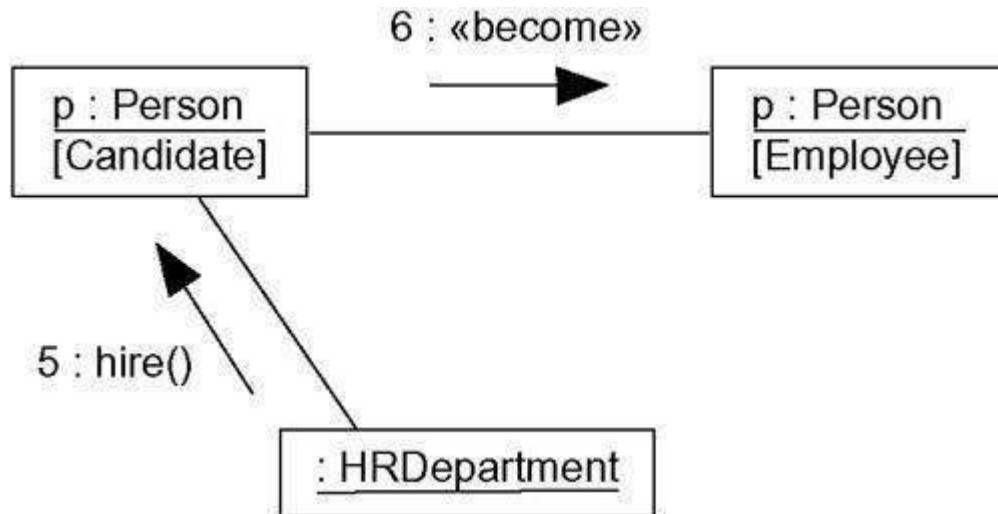
Figure Modeling Static Types



This diagram specifies that instances of the **Person** class may be any of the three types• namely, **Candidate**, **Employee**, or **Retiree**.

Figure shows the dynamic nature of a person's type. In this fragment of an interaction diagram, **p** (the **Person** object) changes its role from **Candidate** to **Employee**.

Figure Modeling Dynamic Types



Packages

Terms and Concepts

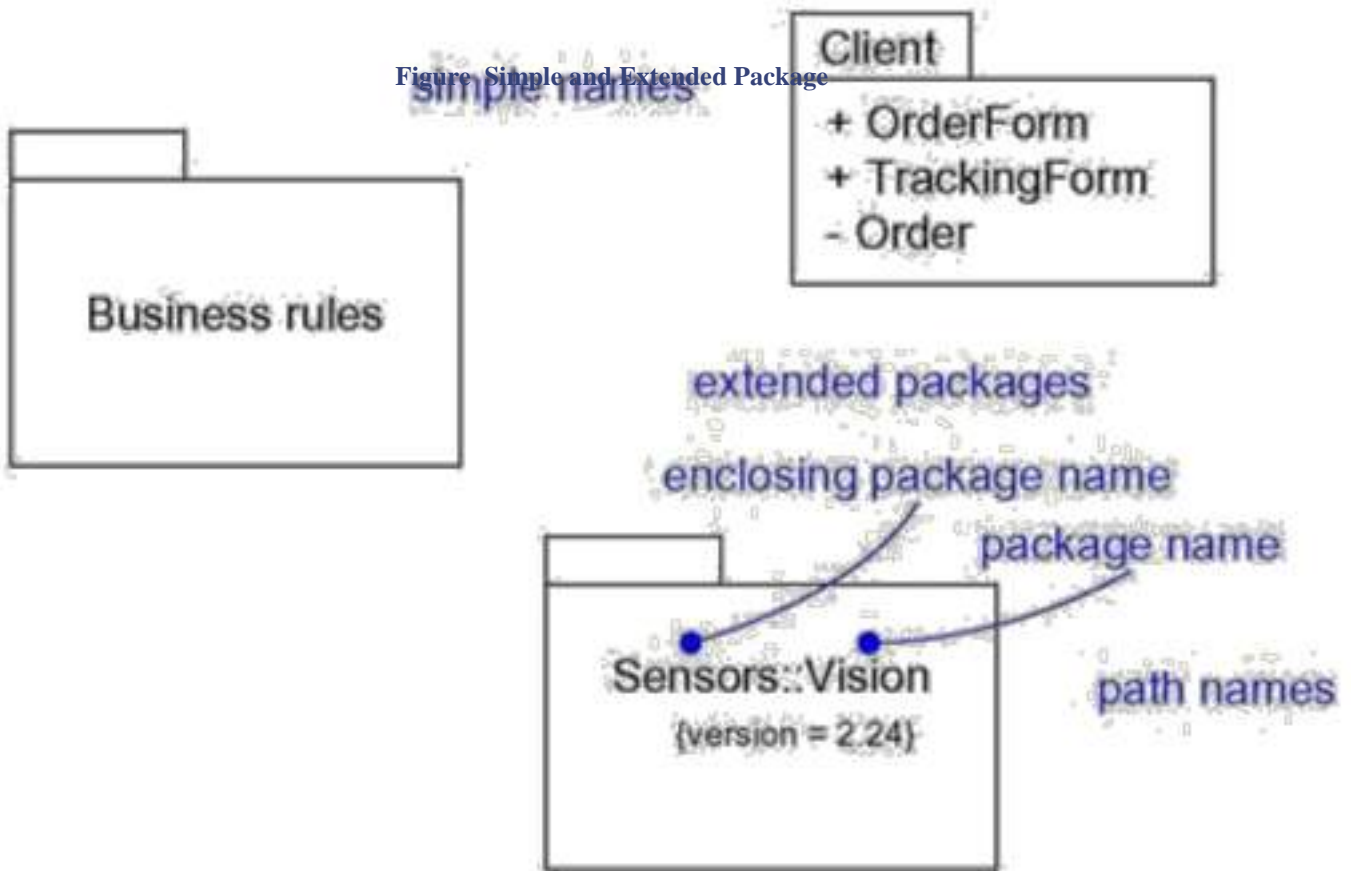
A *package* is a general- purpose mechanism for organizing elements into groups. Graphically, a package is rendered as a tabbed folder.

Names

A package name must be unique within its enclosing package.

Every package must have a name that distinguishes it from other packages. A *name* is a textual string. That name alone is known as a *simple name*; a *path name* is the package name prefixed by the name of the package in which that package lives, if any. A package is typically drawn showing only its name, as in Figure. Just as with classes, you may draw packages adorned with tagged values or with additional compartments to expose their details.

Figure: Simple and Extended Package



Owned Elements

A package may own other elements, including classes, interfaces, components, nodes, collaborations, use cases, diagrams, and even other packages. Owning is a composite relationship, which means that the element is declared in the package. If the package is destroyed, the element is destroyed. Every element is uniquely owned by exactly one package.

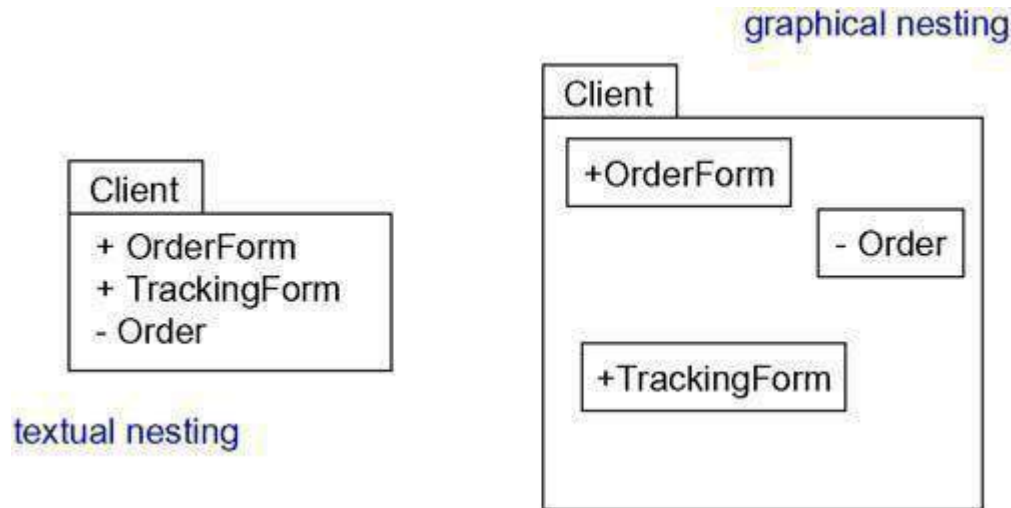
A package forms a namespace, which means that elements of the same kind must be named uniquely within the context of its enclosing package. For example, you can't have two classes named **Queue** owned by the same package, but you can have a class named **Queue** in package **P1** and another (and different) class named **Queue** in package **P2**. The classes **P1::Queue** and **P2::Queue** are, in fact, different classes and can be distinguished by their path names. Different kinds of elements may have the same name.

Elements of different kinds may have the same name within a package. Thus, you can have a class named **Timer**, as well as a component named **Timer**, within the same package. In practice, however, to avoid confusion, it's best to name elements uniquely for all kinds within a package.

Packages may own other packages. This means that it's possible to decompose your models hierarchically. For example, you might have a class named **Camera** that lives in the package **Vision** that in turn lives in the package **Sensors**. The full name of this class is **Sensors::Vision::Camera**. In practice, it's best to avoid deeply nested packages. Two to three levels of nesting is about the limit that's manageable. More than nesting, you'll use importing to organize your packages.

As Figure shows, you can explicitly show the contents of a package either textually or graphically. Note that when you show these owned elements, you place the name of the package in the tab. In practice, you typically won't want to show the contents of packages this way. Instead, you'll use tools to zoom into the contents of a package.

Figure Owned Elements



Visibility

You can control the visibility of the elements owned by a package just as you can control the visibility of the attributes and operations owned by a class. Typically, an element owned by a package is public, which means that it is visible to the contents of any package that imports the element's enclosing package. Conversely, protected elements can only be seen by children, and private elements cannot be seen outside the package in which they are declared. In Figure, **OrderForm** is a public part of the package **Client**, and **Order** is a private part. A package that imports **Client** can see **OrderForm**, but it cannot see **Order**. As viewed from the outside, the fully qualified name of **OrderForm** would be **Client::OrderForm**.

You specify the visibility of an element owned by a package by prefixing the element's name with an appropriate visibility symbol. Public elements are rendered by prefixing their name with a + symbol, as for **OrderForm** in Figure. Collectively, the public parts of a package constitute the package's interface.

Just as with classes, you can designate an element as protected or private, rendered by prefixing the element's name with a # symbol and a - symbol, respectively. Protected elements are visible only to packages that inherit from another package; private elements are not visible outside the package at all.

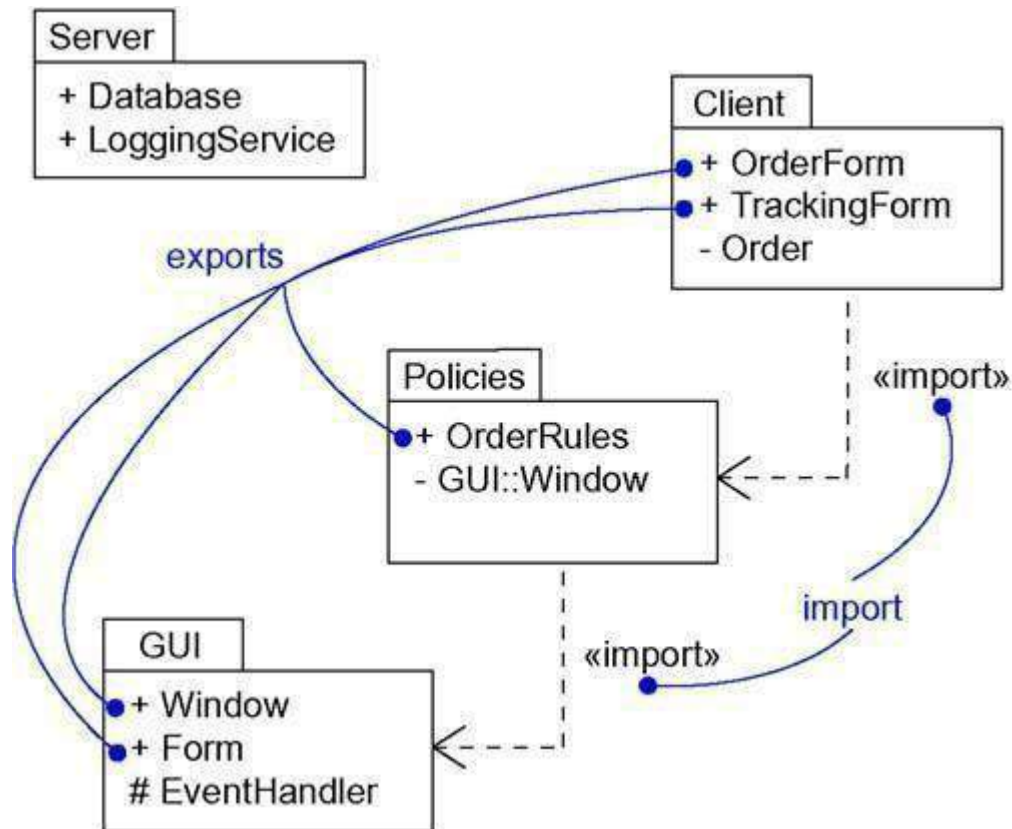
Importing and Exporting

Suppose you have two classes named **A** and **B** sitting side by side. Because they are peers, **A** can see **B** and **B** can see **A**, so both can depend on the other. Just two classes makes for a trivial system, so you really don't need any kind of packaging.

So suppose that instead you put **A** in one package and **B** in another package, both packages sitting side by side. Suppose also that **A** and **B** are both declared as public parts of their respective packages. This is a very different situation. Although **A** and **B** are both public, neither can access the other because their enclosing packages form an opaque wall. However, if **A**'s package imports **B**'s package, **A** can now see **B**, although **B** cannot see **A**. Importing grants a one-way permission for the elements in one package to access the elements in another package. In the UML, you model an import relationship as a dependency adorned with the stereotype **import**. By packaging your abstractions into meaningful chunks and then controlling their access by importing, you can control the complexity of large numbers of abstractions.

The public parts of a package are called its exports. For example, in Figure, the package **GUI** exports two classes, **Window** and **Form**. **EventHandler** is not exported by **GUI**; **EventHandler** is a protected part of the package.

Figure Importing and Exporting



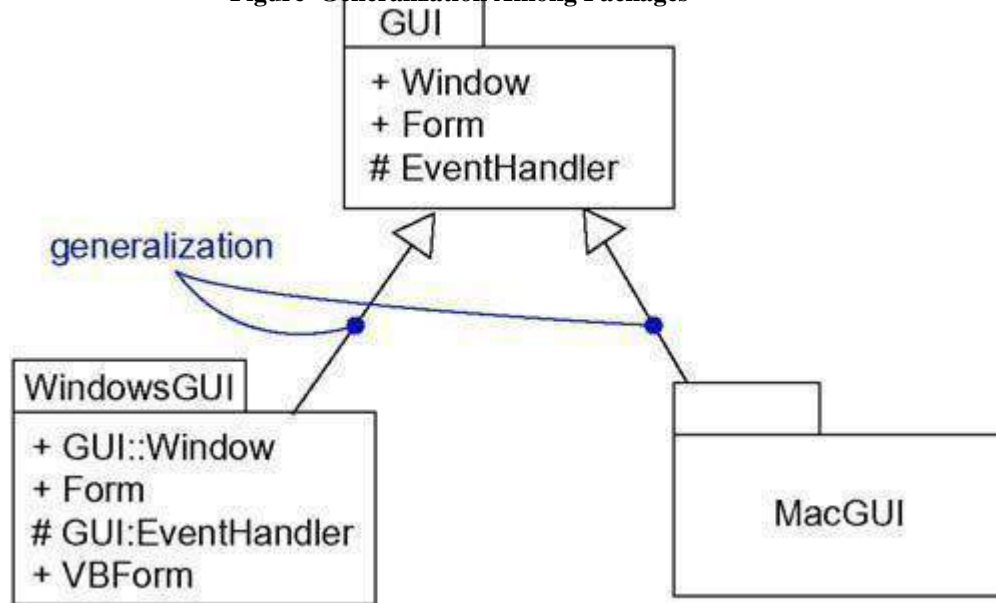
The parts that one package exports are visible only to the contents of those packages that explicitly import the package. In this example, **Policies** explicitly imports the package **GUI**. **GUI::Window** and **GUI::Form** are therefore made visible to the contents of the package **Policies**. However, **GUI::EventHandler** is not visible because it is protected. Because the package **Server** doesn't import **GUI**, the contents of **Server** don't have permission to access any of the contents of **GUI**. Similarly, the contents of **GUI** don't have permission to access any of the contents of **Server**. Import and access dependencies are not transitive. In this example, **Client** imports **Policies** and **Policies** imports **GUI**, but **Client** does not by implication import **GUI**. Therefore, the contents of **Client** have access to the exports of **Policies**, but they do not have access to the exports of **GUI**. To gain access, **Client** would have to import **GUI** explicitly.

Generalization

There are two kinds of relationships you can have between packages: import and access dependencies, used to import into one package elements exported from another, and generalizations, used to specify families of packages.

Generalization among packages is very much like generalization among classes. For example, in Figure, the package **GUI** is shown to export two classes (**Window** and **Form**) and one protected class (**EventHandler**). Two packages specialize the more general package **GUI**: **WindowsGUI** and **MacGUI**. These specialized packages inherit the public and protected elements of the more general package. But, just as in class inheritance, packages can replace more general elements and add new ones. For example, the package **WindowsGUI** inherits from **GUI**, so it includes the classes **GUI::Window** and **GUI::EventHandler**. In addition, **WindowsGUI** overrides one class (**Form**) and adds a new one (**VBForm**).

Figure Generalization Among Packages



Packages involved in generalization relationships follow the same principle of substitutability as do classes. A specialized package (such as **WindowsGUI**) can be used anywhere a more general package (such as **GUI**) is used.

Standard Elements

All of the UML's extensibility mechanisms apply to packages. Most often, you'll use tagged values to add new package properties (such as specifying the author of a package) and stereotypes to specify new kinds of packages (such as packages that encapsulate operating system services).

The UML defines five standard stereotypes that apply to packages.

1. facade	Specifies a package that is only a view on some other package
2. framework	Specifies a package consisting mainly of patterns
3. stub	Specifies a package that serves as a proxy for the public contents of another

	package
4. subsystem	Specifies a package representing an independent part of the entire system being modeled
5. system	Specifies a package representing the entire system being modeled

The UML does not specify icons for any of these stereotypes. In addition to these five package stereotypes, you'll also use dependencies designated using the standard stereotype import.

Most of these standard elements are discussed elsewhere, except for facade and stub. These two stereotypes help you to manage very large models. You use facades to provide elided views on otherwise complex packages. For example, your system might contain the package **BusinessModel**. You might want to expose a subset of its elements to one set of users (to show only those elements associated with customers), and another subset to a different set of users (to show only those elements associated with products). To do so, you would define facades, which import (and never own) only a subset of the elements in another package. You use stubs when you have tools that split apart a system into packages that you manipulate at different times and potentially in different places. For example, if you have a development team working in two locations, the team at one site would provide a stub for the packages the other team required. This strategy lets the teams work independently without disturbing each other's work, with the stub packages capturing the seams in the system that must be managed carefully.

Common Modeling Techniques

Modeling Groups of Elements

The most common purpose for which you'll use packages is to organize modeling elements into groups that you can name and manipulate as a set. If you are developing a trivial application, you won't need packages at all. All your abstractions will fit nicely into one package. For every other system, however, you'll find that many of your system's classes, interfaces, components, nodes, and even diagrams tend to naturally fall into groups. You model these groups as packages.

There is one important distinction between classes and packages: Classes are abstractions of things found in your problem or solution; packages are mechanisms you use to organize the things in your model. Packages have no identity (meaning that you can't have instances of packages, so they are invisible in the running system); classes do have identity (classes have instances, which are elements of a running system).

Most of the time, you'll use packages to group the same basic kind of elements. For example, you might separate all the classes and their corresponding relationships from your system's design view into a series of packages, using the UML's import dependencies to control access among these packages. You might organize all the components in your system's implementation view in a similar fashion.

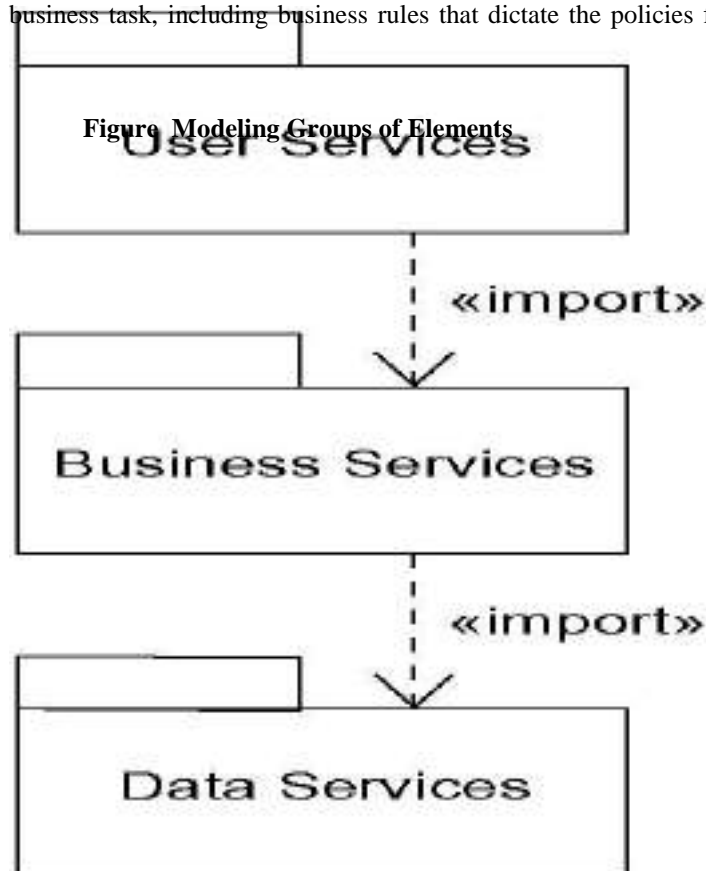
You can also use packages to group different kinds of elements. For example, for a system being developed by a geographically distributed team, you might use packages as your unit of configuration management, putting in them all the classes and diagrams that each team can check in and check out separately. In fact, it's common to use packages to group modeling elements and their associated diagrams.

To model groups of elements,

- Scan the modeling elements in a particular architectural view and look for clumps defined by elements that are conceptually or semantically close to one another.
- Surround each of these clumps in a package.

- For each package, distinguish which elements should be accessible outside the package. Mark them public, and all others protected or private. When in doubt, hide the element.
- Explicitly connect packages that build on others via import dependencies.
- In the case of families of packages, connect specialized packages to their more general part via generalizations.

For example, Figure shows a set of packages that organize the classes in an information system's design view into a classic three-tier architecture. The elements in the package **UserServices** provide the visual interface for presenting information and gathering data. The elements in the package **Data Services** maintain, access, and update data. The elements in the package **Business Services** bridge the elements in the other two packages and encompass all the classes and other elements that manage requests from the user to execute a business task, including business rules that dictate the policies for manipulating data.



In a trivial system, you could lump all your abstractions into one package. However, by organizing your classes and other elements of the system's design view into three packages, you not only make your model more understandable, but you can control access to the elements of your model by hiding some and exporting others.

Modeling Architectural Views

Using packages to group related elements is important; you can't develop complex models without doing so. This approach works well for organizing related elements, such as classes, interfaces, components, nodes, and diagrams. As you consider the different views of a software system's architecture, you need even larger chunks. You can use packages to model the views of an architecture.

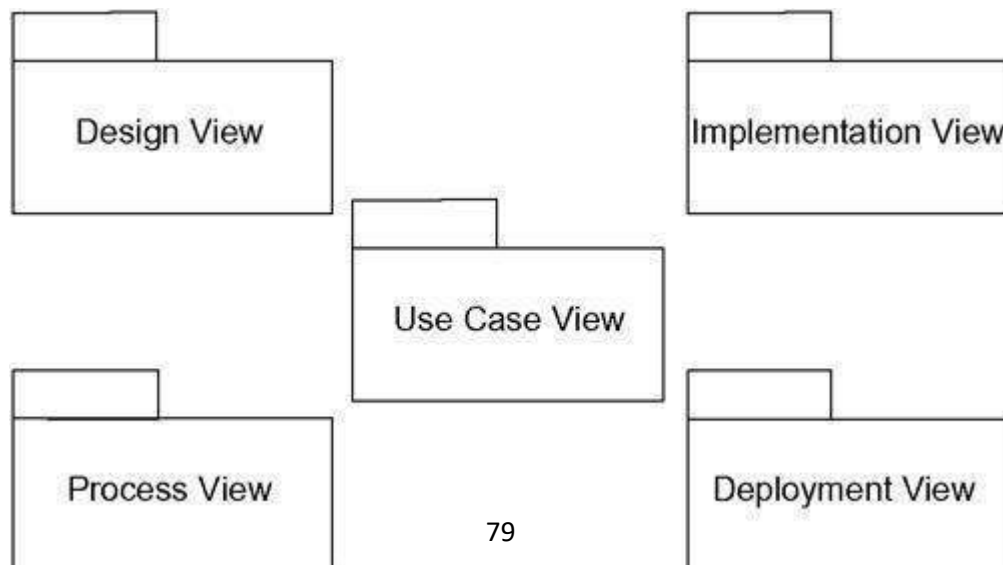
Remember that a view is a projection into the organization and structure of a system, focused on a particular aspect of that system. This definition has two implications. First, you can decompose a system into almost orthogonal packages, each of which addresses a set of architecturally significant decisions. For example, you might have a design view, a process view, an implementation view, a deployment view, and a use case view. Second, these packages own all the abstractions germane to that view. For example, all the components in your model would belong to the package that represents the implementation view.

To model architectural views,

- Identify the set of architectural views that are significant in the context of your problem. In practice, this typically includes a design view, a process view, an implementation view, a deployment view, and a use case view.
- Place the elements (and diagrams) that are necessary and sufficient to visualize, specify, construct, and document the semantics of each view into the appropriate package.
- As necessary, further group these elements into their own packages.
- There will typically be dependencies across the elements in different views. So, in general, let each view at the top of a system be open to all others at that level.

For example, Figure illustrates a canonical top-level decomposition that's appropriate foreven the most complex system you might encounter.

Figure Modeling Architectural Views



Class Diagrams

Terms and Concepts

A *class diagram* is a diagram that shows a set of classes, interfaces, and collaborations and their relationships. Graphically, a class diagram is a collection of vertices and arcs.

Common Properties

A class diagram is just a special kind of diagram and shares the same common properties as do all other **diagrams**• a name and graphical content that are a projection into a model. What distinguishes a class diagram from all other kinds of diagrams is its particular content.

Contents

Class diagrams commonly contain the following things:

- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships

Like all other diagrams, class diagrams may contain notes and constraints.

Class diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place instances in your class diagrams, as well, especially when you want to visualize the (possibly dynamic) type of an instance.

Common Uses

You use class diagrams to model the static design view of a system. This view primarily supports the **functional requirements of a system**• the services the system should provide to its end users.

When you model the static design view of a system, you'll typically use class diagrams in one of three ways.

1. To model the vocabulary of a system

Modeling the vocabulary of a system involves making a decision about which abstractions are a part of the system under consideration and which fall outside its boundaries. You use class diagrams to specify these abstractions and their responsibilities.

2. To model simple collaborations

A collaboration is a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. For example, when you're modeling the semantics of a transaction in a distributed system, you can't just stare at a single class to understand what's going on. Rather, these semantics are carried out by a set of classes that work together. You use class diagrams to visualize and specify this set of classes and their relationships.

3. To model a logical database schema

Think of a schema as the blueprint for the conceptual design of a database. In many domains, you'll want to store persistent information in a relational database or in an object-oriented database. You can model schemas for these databases using class diagrams.

Common Modeling Techniques

Modeling Simple Collaborations

No class stands alone. Rather, each works in collaboration with others to carry out some semantics greater than each individual. Therefore, in addition to capturing the vocabulary of your system, you'll also need to turn your attention to visualizing, specifying, constructing, and documenting the various ways these things in your vocabulary work together. You use class diagrams to represent such collaborations.

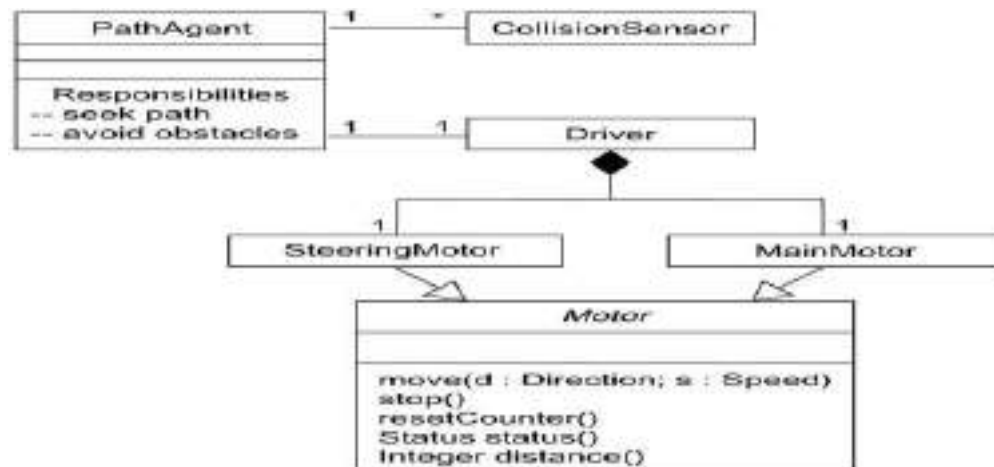
When you create a class diagram, you just model a part of the things and relationships that make up your system's design view. For this reason, each class diagram should focus on one collaboration at a time.

To model a collaboration,

- Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.
- For each mechanism, identify the classes, interfaces, and other collaborations that participate in this collaboration. Identify the relationships among these things, as well.
- Use scenarios to walk through these things. Along the way, you'll discover parts of your model that were missing and parts that were just plain semantically wrong.
- Be sure to populate these elements with their contents. For classes, start with getting a good balance of responsibilities. Then, over time, turn these into concrete attributes and operations.

For example, Figure shows a set of classes drawn from the implementation of an autonomous robot. The figure focuses on the classes involved in the mechanism for moving the robot along a path. You'll find one abstract class (**Motor**) with two concrete children, **SteeringMotor** and **MainMotor**. Both of these classes inherit the five operations of their parent, **Motor**. The two classes are, in turn, shown as parts of another class, **Driver**. The class **PathAgent** has a one-to-one association to **Driver** and a one-to-many association to **CollisionSensor**. No attributes or operations are shown for **PathAgent**, although its responsibilities are given.

Figure Modeling Simple Collaborations



There are many more classes involved in this system, but this diagram focuses only on those abstractions that are directly involved in moving the robot. You'll see some of these same classes in other diagrams. For example, although not shown here, the class **PathAgent** collaborates with at least two other classes (**Environment** and **GoalAgent**) in a higher-level mechanism for managing the conflicting goals the robot might have at a given moment. Similarly, also not shown here, the classes **CollisionSensor** and **Driver** (and its parts) collaborate with another class (**FaultAgent**) in a mechanism responsible for continuously checking the robot's hardware for errors. By focusing on each of these collaborations in different diagrams, you provide an understandable view of the system from several angles.

Modeling a Logical Database Schema

Many of the systems you'll model will have persistent objects, which means that they can be stored in a database for later retrieval. Most often, you'll use a relational database, an object-oriented database, or a hybrid object/relational database for persistent storage. The UML is well-suited to modeling logical database schemas, as well as physical databases themselves.

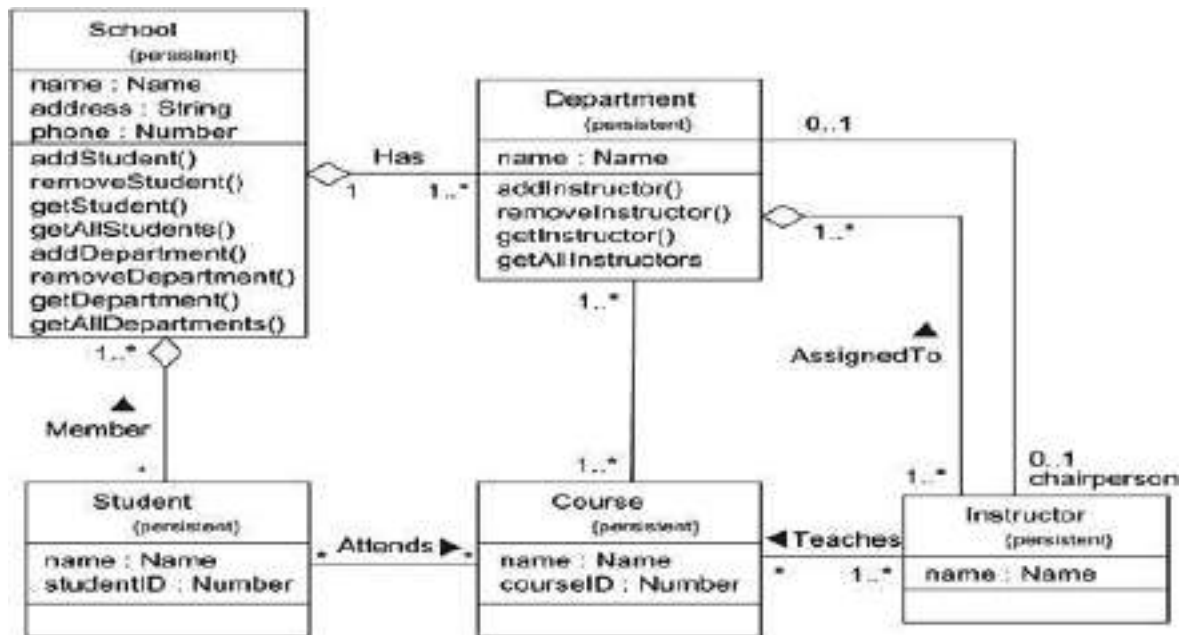
The UML's class diagrams are a superset of entity-relationship (E-R) diagrams, a common modeling tool for logical database design. Whereas classical E-R diagrams focus only on data, class diagrams go a step further by permitting the modeling of behavior, as well. In the physical database, these logical operations are generally turned into triggers or stored procedures.

To model a schema,

- Identify those classes in your model whose state must transcend the lifetime of their applications.
- Create a class diagram that contains these classes and mark them as persistent (a standard tagged value). You can define your own set of tagged values to address database-specific details.
- Expand the structural details of these classes. In general, this means specifying the details of their attributes and focusing on the associations and their cardinalities that structure these classes.
- Watch for common patterns that complicate physical database design, such as cyclic associations, one-to-one associations, and n-ary associations. Where necessary, create intermediate abstractions to simplify your logical structure.
- Consider also the behavior of these classes by expanding operations that are important for data access and data integrity. In general, to provide a better separation of concerns, business rules concerned with the manipulation of sets of these objects should be encapsulated in a layer above these persistent classes.
- Where possible, use tools to help you transform your logical design into a physical design.

Figure shows a set of classes drawn from an information system for a school. This figure expands upon an earlier class diagram, and you'll see the details of these classes revealed to a level sufficient to construct a physical database. Starting at the bottom-left of this diagram, you will find the classes named **Student**, **Course**, and **Instructor**. There's an association between **Student** and **Course**, specifying that students attend courses. Furthermore, every student may attend any number of courses and every course may have any number of students.

Figure Modeling a Schema



All six of these classes are marked as persistent, indicating that their instances are intended to live in a database or some other form of persistent store. This diagram also exposes the attributes of all six of these classes. Notice that all the attributes are primitive types. When you are modeling a schema, you'll generally want to model the relationship to any nonprimitive types using an explicit aggregation rather than an attribute.

Two of these classes (**School** and **Department**) expose several operations for manipulating their parts. These operations are included because they are important to maintain data integrity (adding or removing a **Department**, for example, will have some rippling effects). There are many other operations that you might consider for these and the other classes, such as querying the prerequisites of a course before assigning a student. These are more business rules than they are operations for database integrity and so are best placed at a higher level of abstraction than this schema.

Forward and Reverse Engineering

Modeling is important, but you have to remember that the primary product of a development team is software, not diagrams. Of course, the reason you create models is to predictably deliver at the right time the right software that satisfies the evolving goals of its users and the business. For this reason, it's important that the models you create and the implementations you deploy map to one another and do so in a way that minimizes or even eliminates the cost of keeping your models and your implementation in sync with one another.

For some uses of the UML, the models you create will never map to code. For example, if you are modeling a business process using activity diagrams, many of the activities you model will involve people, not computers. In other cases, you'll want to model systems whose parts are, from your level of abstraction, just a piece of hardware (although at another level of abstraction, it's a good bet that this hardware contains an embedded computer and software).

In most cases, though, the models you create will map to code. The UML does not specify a particular mapping to any object-oriented programming language, but the UML was designed with such mappings

in mind. This is especially true for class diagrams, whose contents have a clear mapping to all the industrial-strength object-oriented languages, such as Java, C++, Smalltalk, Eiffel, Ada, ObjectPascal, and Forte. The UML was also designed to map to a variety of commercial object-based languages, such as Visual Basic.

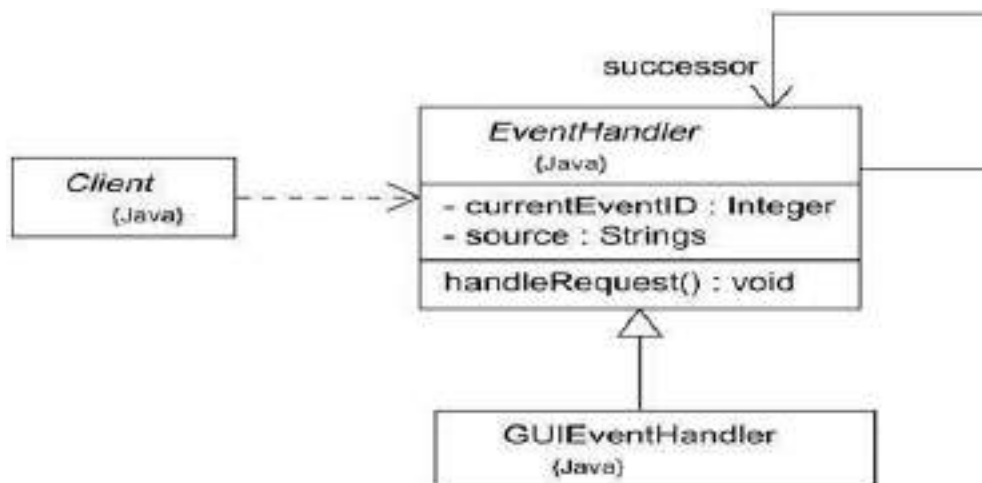
Forward engineering is the process of transforming a model into code through a mapping to an implementation language. Forward engineering results in a loss of information, because models written in the UML are semantically richer than any current object-oriented programming language. In fact, this is a major reason why you need models in addition to code. Structural features, such as collaborations, and behavioral features, such as interactions, can be visualized clearly in the UML, but not so clearly from raw code.

To forward engineer a class diagram,

- Identify the rules for mapping to your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Depending on the semantics of the languages you choose, you may have to constrain your use of certain UML features. For example, the UML permits you to model multiple inheritance, but Smalltalk permits only single inheritance. You can either choose to prohibit developers from modeling with multiple inheritance (which makes your models language-dependent) or develop idioms that transform these richer features into the implementation language (which makes the mapping more complex).
- Use tagged values to specify your target language. You can do this at the level of individual classes if you need precise control. You can also do so at a higher level, such as with collaborations or packages.
- Use tools to forward engineer your models.

Figure illustrates a simple class diagram specifying an instantiation of the chain of responsibility pattern. This particular instantiation involves three classes: **Client**, **EventHandler**, and **GUIEventHandler**. **Client** and **EventHandler** are shown as abstract classes, whereas **GUIEventHandler** is concrete. **EventHandler** has the usual operation expected of this pattern (**handleRequest**), although two private attributes have been added for this instantiation.

Figure Forward Engineering



All of these classes specify a mapping to Java, as noted in their tagged value. Forward engineering the classes in this diagram to Java is straightforward, using a tool. Forward engineering the class **EventHandler** yields the following code.

```
public abstract class EventHandler {  
  
    EventHandler successor;  
    private Integer currentEventID; private  
    String source;  
  
    EventHandler() {}  
    public void handleRequest() {}  
  
}
```

Reverse engineering is the process of transforming code into a model through a mapping from a specific implementation language. Reverse engineering results in a flood of information, some of which is at a lower level of detail than you'll need to build useful models. At the same time, reverse engineering is incomplete. There is a loss of information when forward engineering models into code, and so you can't completely recreate a model from code unless your tools encode information in the source comments that goes beyond the semantics of the implementation language.

To reverse engineer a class diagram,

- Identify the rules for mapping from your implementation language or languages of choice. This is something you'll want to do for your project or your organization as a whole.
- Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or modify an existing one that was previously forward engineered.

Instances

Terms and Concepts

An *instance* is a concrete manifestation of an abstraction to which a set of operations can be applied and which has a state that stores the effects of the operations. *Instance* and *object* are largely synonymous. Graphically, an instance is rendered by underlining its name.

Abstractions and Instances

Instances don't stand alone; they are almost always tied to an abstraction. Most instances you'll model with the UML will be instances of classes (and these things are called objects), although you can have instances of other things, such as components, nodes, use cases, and associations. In the UML, an instance is easily distinguishable from an abstraction. To indicate an instance, you underline its name.

In a general sense, an object is something that takes up space in the real or conceptual world, and you can do things to it. For example, an instance of a node is typically a computer that physically sits in a room; an instance of a component takes up some space on the file system; an instance of a customer record consumes some amount of physical memory. Similarly, an instance of a flight envelope for an aircraft is something you can manipulate mathematically.

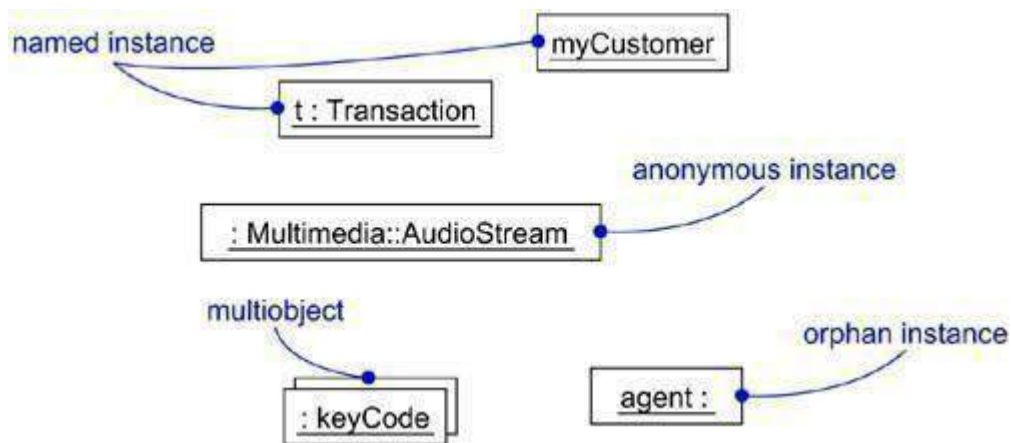
You can use the UML to model these physical instances, but you can also model things that are not so concrete. For example, an abstract class, by definition, may not have any direct instances.

However, you can model indirect instances of abstract classes in order to show the use of a prototypical instance of that abstract class. Literally, no such object might exist. But pragmatically, this instance lets you name one of any potential instances of concrete children of that abstract class. This same touch applies to interfaces. By their very definition, interfaces may not have any direct instances, but you can model a prototypical instance of an interface, representing one of any potential instances of concrete classes that realize that interface.

When you model instances, you'll place them in object diagrams (if you want to visualize their structural details) or in interaction and activity diagrams (if you want to visualize their participation in dynamic situations). Although typically not necessary, you can place objects in class diagrams if you want to explicitly show the relationship of an object to its abstraction.

The classifier of an instance is usually static. For example, once you create an instance of a class, its class won't change during the lifetime of that object. In some modeling situations and in some programming languages, however, it is possible to change the abstraction of an instance. For example, a **Caterpillar** object might become a **Butterfly** object. It's the same object, but of a different abstraction.

Figure Named, Anonymous, Multiple, and Orphan Instances



Names

Every instance must have a name that distinguishes it from other instances within its context. Typically, an object lives within the context of an operation, a component, or a node. A *name* is a textual string, such as `t` and `myCustomer` in Figure. That name alone is known as a *simple name*. The abstraction of the instance may be a simple name (such as `Transaction`) or it may be a *path name* (such as `Multimedia::AudioStream`) which is the abstraction's name prefixed by the name of the package in which that abstraction lives.

When you explicitly name an object, you are really giving it a name (such as `t`) that's usable by a human. You can also simply name an object (such as `aCustomer`) and elide its abstraction if it's obvious in the

given context. In many cases, however, the real name of an object is known only to the computer on which that object lives. In such cases, you can render an anonymous object (such as **:Multimedia::AudioStream**). Each occurrence of an anonymous object is considered distinct from all other occurrences. If you don't even know the object's associated abstraction, you must at least give it an explicit name (such as **agent** :).

Especially when you are modeling large collections of objects, it's clumsy to render the collection itself plus its individual instances. Instead, you can model multiobjects (such as **:keyCode**) as in Figure, representing a collection of anonymous objects.

Operations

Not only is an object something that usually takes up space in the real world, it is also something you can do things to. The operations you can perform on an object are declared in the object's abstraction. For example, if the class **Transaction** defines the operation **commit**, then given the instance **t : Transaction**, you can write expressions such as **t.commit()**. The execution of this expression means that **t** (the object) is operated on by **commit** (the operation). Depending on the inheritance lattice associated with **Transaction**, this operation may or may not be invoked polymorphically.

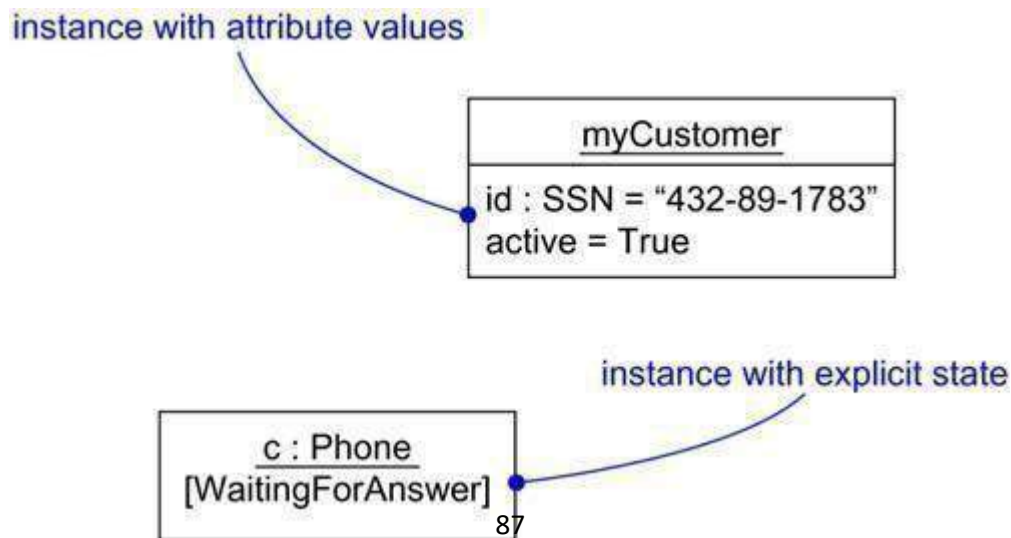
State

An object also has state, which in this sense encompasses all the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties. These properties include the attributes of the object, as well as all its aggregate parts. An object's state is therefore dynamic. So when you visualize its state, you are really specifying the value of its state at a given moment in time and space. It's possible to show the changing state of an object by showing it multiple times in the same interaction diagram, but with each occurrence representing a different state.

When you operate on an object, you typically change its state; when you query an object, you don't change its state. For example, when you make an airline reservation (represented by the object **r : Reservation**), you might set the value of one of its attributes (for example, **price = 395.75**). If you change your reservation, perhaps by adding a new leg to your itinerary, then its state might change (for example, **price = 1024.86**).

As Figure shows, you can use the UML to render the value of an object's attributes. For example, **myCustomer** is shown with the attribute **id** having the value "432-89-1783." In this case, **id**'s type (**SSN**) is shown explicitly, although it can be elided (as for **active = True**), because its type can be found in the declaration of **id** in **myCustomer**'s associated class.

Figure Object State

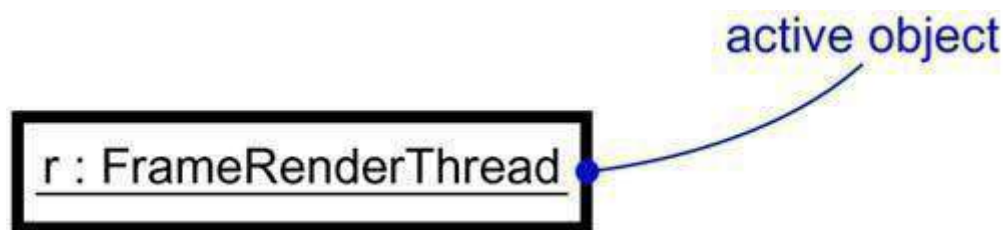


You can associate a state machine with a class, which is especially useful when modeling event-driven systems or when modeling the lifetime of a class. In these cases, you can also show the state of this machine for a given object at a given time. For example, as [Figure](#) shows, the object **c** (an instance of the class **Phone**) is indicated in the state **WaitingForAnswer**, a named state defined in the state machine for **Phone**.

Other Features

Processes and threads are an important element of a system's process view, so the UML provides a visual cue to distinguish elements that are active (those that are part of a process or thread and represent a root of a flow of control) from those that are passive. You can declare active classes that reify a process or thread, and in turn you can distinguish an instance of an active class, as in [Figure](#).

Figure 13-4 Active Objects

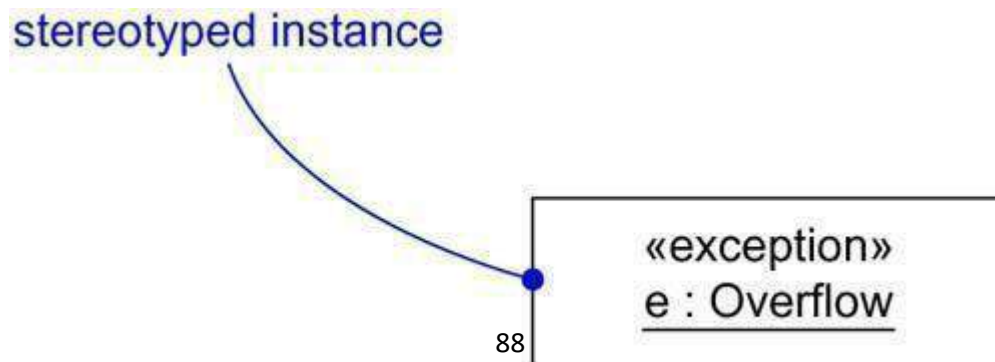


There are two other elements in the UML that may have instances. The first is a link. A link is a semantic connection among objects. An instance of an association is therefore a link. A link is rendered as a line, just like an association, but it can be distinguished from an association because links only connect objects. The second is a class-scoped attribute and operation. A class-scoped feature is in effect an object in the class that is shared by all instances of the class.

Standard Elements

All of the UML's extensibility mechanisms apply to objects. Usually, however, you don't stereotype an instance directly, nor do you give it its own tagged values. Instead, an object's stereotype and tagged values derive from the stereotype and tagged values of its associated abstraction. For example, as [Figure](#) shows, you can explicitly indicate an object's stereotype, as well as its abstraction.

Figure Stereotyped Objects



The UML defines two standard stereotypes that apply to the dependency relationships among objects and among classes:

1. instanceOf	Specifies that the client object is an instance of the supplier classifier
2. instantiate	Specifies that the client class creates instances of the supplier class

There are also two stereotypes related to objects that apply to messages and transitions:

1. become	Specifies that the client is the same object as the supplier, but at a later time and with possibly different values, state, or roles
2. copy	Specifies that the client object is an exact but independent copy of the supplier

The UML defines a standard constraint that applies to objects:

transient	Specifies that an instance of the role is created during execution of the enclosing interaction but is destroyed before completion of execution
------------------	---

Common Modeling Techniques

Modeling Concrete Instances

When you model concrete instances, you are in effect visualizing things that live in the real world. You can't exactly see an instance of a **Customer** class, for example, unless that customer is standing beside you; in a debugger, you might be able to see a representation of that object, however. One of the things for which you'll use objects is to model concrete instances that exist in the real world. For example, if you want to model the topology of your organization's network, you'll use deployment diagrams containing instances of nodes. Similarly, if you want to model the components that live on the physical nodes in this network, you'll use component diagrams containing instances of the components. Finally, suppose you have a debugger connected to your running system; it can present the structural relationships among instances by rendering an object diagram.

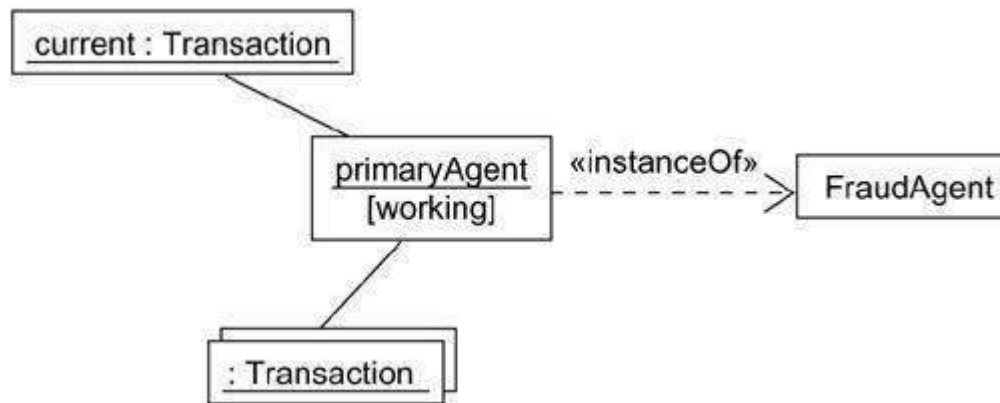
To model concrete instances,

- Identify those instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.
- Render these objects in the UML as instances. Where possible, give each object a name. If there is no meaningful name for the object, render it as an anonymous object.
- Expose the stereotype, tagged values, and attributes (with their values) of each instance necessary and sufficient to model your problem.
- Render these instances and their relationships in an object diagram or other diagram appropriate to the kind of the instance.

For example, Figure 13-6 shows an object diagram drawn from the execution of a credit card validation system, perhaps as seen by a debugger that's probing the running system. There is one multiobject, containing anonymous instances of the class **Transaction**. There are also two **explicitly named objects**•

primaryAgent and **current** both of which expose their class, although in different ways. The diagram also explicitly shows the current state of the object **primaryAgent**.

Figure Modeling Concrete Instances



Note also the use of the dependency stereotyped as **instanceOf**, indicating the class of **primaryAgent**. Typically, you'll want to explicitly show these class/object relationships only if you also intend to show relationships with other classes.

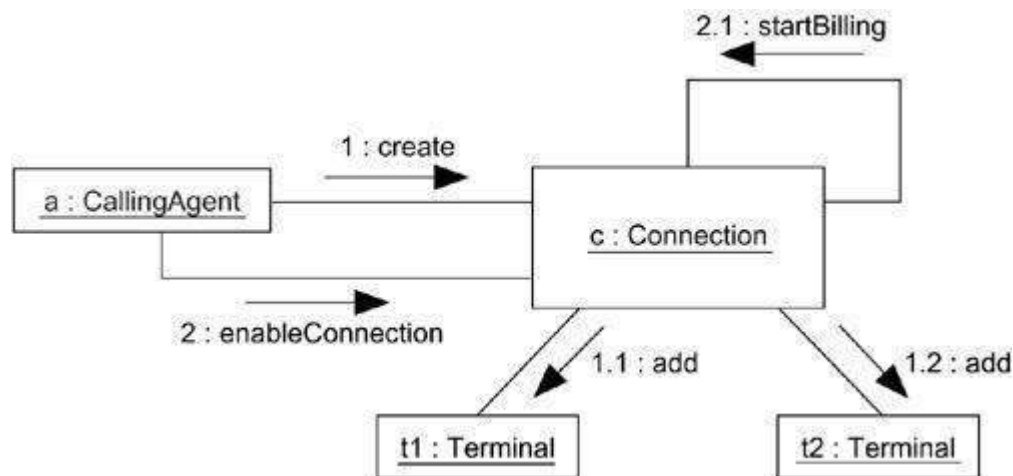
Modeling Prototypical Instances

Perhaps the most important thing for which you'll use instances is to model the dynamic interactions among objects. When you model such interactions, you are generally not modeling concrete instances that exist in the real world. Instead, you are modeling conceptual objects that are essentially proxies or stand-ins for objects that will eventually act that way in the real world. These are prototypical objects and, therefore, are roles to which concrete instances conform. For example, if you want to model the ways objects in a windowing application react to a mouse event, you'd draw an interaction diagram containing prototypical instances of windows, events, and handlers. To model prototypical instances,

- Identify those prototypical instances necessary and sufficient to visualize, specify, construct, or document the problem you are modeling.
- Render these objects in the UML as instances. Where possible, give each object a name. If there is no meaningful name for the object, render it as an anonymous object.
- Expose the properties of each instance necessary and sufficient to model your problem.
- Render these instances and their relationships in an interaction diagram or an activity diagram.

Figure shows an interaction diagram illustrating a partial scenario for initiating a phone call in the context of a switch. There are four prototypical objects: **a** (a **CallingAgent**), **c** (a **Connection**), and **t1** and **t2** (both instances of **Terminal**). All four of these objects are prototypical; all represent conceptual proxies for concrete objects that may exist in the real world.

Figure Modeling Prototypical Instances



Object Diagrams

Terms and Concepts

An *object diagram* is a diagram that shows a set of objects and their relationships at a point in time. Graphically, an object diagram is a collection of vertices and arcs.

Common Properties

An object diagram is a special kind of diagram and shares the same common properties as all other **diagrams**• **that is, a name and graphical contents that are a projection into a model. What distinguishes an** object diagram from all other kinds of diagrams is its particular content.

Contents

Object diagrams commonly contain

- Objects
- Links

Like all other diagrams, object diagrams may contain notes and constraints.

Object diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place classes in your object diagrams, as well, especially when you want to visualize the classes behind each instance.

Common Uses

You use object diagrams to model the static design view or static process view of a system just as you do with class diagrams, but from the perspective of real or prototypical instances. This view primarily **supports the functional requirements of a system**• **that is, the services the system should provide to its end** users. Object diagrams let you model static data structures.

When you model the static design view or static process view of a system, you typically use object

diagrams in one way:

- To model object structures

Modeling object structures involves taking a snapshot of the objects in a system at a given moment in time. An object diagram represents one static frame in the dynamic storyboard represented by an interaction diagram. You use object diagrams to visualize, specify, construct, and document the existence of certain instances in your system, together with their relationships to one another.

Common Modeling Techniques

Modeling Object Structures

When you construct a class diagram, a component diagram, or a deployment diagram, what you are really doing is capturing a set of abstractions that are interesting to you as a group and, in that context, exposing their semantics and their relationships to other abstractions in the group. These diagrams show only potentiality. If class **A** has a one-to-many association to class **B**, then for one instance of **A** there might be five instances of **B**; for another instance of **A** there might be only one instance of **B**. Furthermore, at a given moment in time, that instance of **A**, along with the related instances of **B**, will each have certain values for their attributes and state machines.

If you freeze a running system or just imagine a moment of time in a modeled system, you'll find a set of objects, each in a specific state, and each in a particular relationship to other objects. You can use object diagrams to visualize, specify, construct, and document the structure of these objects. Object diagrams are especially useful for modeling complex data structures.

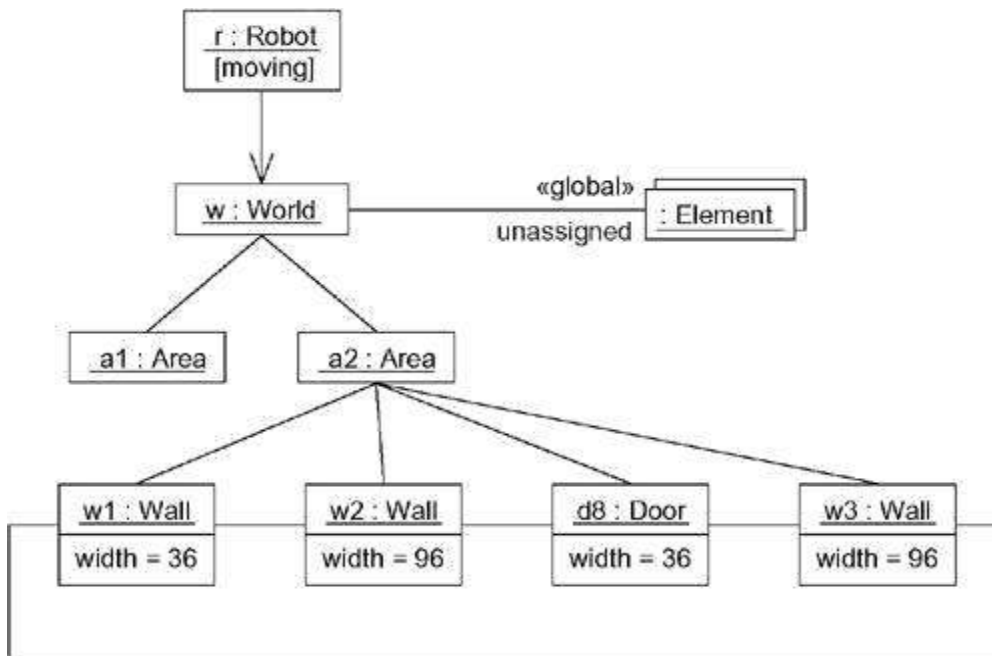
When you model your system's design view, a set of class diagrams can be used to completely specify the semantics of your abstractions and their relationships. With object diagrams, however, you cannot completely specify the object structure of your system. For an individual class, there may be a multitude of possible instances, and for a set of classes in relationship to one another, there may be many times more possible configurations of these objects. Therefore, when you use object diagrams, you can only meaningfully expose interesting sets of concrete or prototypical objects. This is what it means to model an **object structure**• **an object diagram shows one set of objects in relation to one another at one moment in time.**

To model an object structure,

- Identify the mechanism you'd like to model. A mechanism represents some function or behavior of the part of the system you are modeling that results from the interaction of a society of classes, interfaces, and other things.
- For each mechanism, identify the classes, interfaces, and other elements that participate in this collaboration; identify the relationships among these things, as well.
- Consider one scenario that walks through this mechanism. Freeze that scenario at a moment in time, and render each object that participates in the mechanism.
- Expose the state and attribute values of each such object, as necessary, to understand the scenario.
- Similarly, expose the links among these objects, representing instances of associations among them.

For example, Figure shows a set of objects drawn from the implementation of an autonomous robot. This figure focuses on some of the objects involved in the mechanism used by the robot to calculate a model of the world in which it moves. There are many more objects involved in a running system, but this diagram focuses on only those abstractions that are directly involved in creating this world view.

Figure Modeling Object Structures



As this figure indicates, one object represents the robot itself (**r**, an instance of **Robot**), and is currently in the state marked **moving**. This object has a link to **w**, an instance of **World**, which represents an abstraction of the robot's world model. This object has a link to a multiobject that consists of instances of **Element**, which represent entities that the robot has identified but not yet assigned in its world view. These elements are marked as part of the robot's global state.

At this moment in time, **w** is linked to two instances of **Area**. One of them (**a2**) is shown with its own links to three **Wall** and one **Door** object. Each of these walls is marked with its current width, and each is shown linked to its neighboring walls. As this object diagram suggests, the robot has recognized this enclosed area, which has walls on three sides and a door on the fourth.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) an object diagram is theoretically possible but pragmatically of limited value. In an object-oriented system, instances are things that are created and destroyed by the application during run time. Therefore, you can't exactly instantiate these objects from the outside.

Although this is true of most typical object diagrams (which contain instances of classes), it's not true of object diagrams containing instances of components and of nodes. Both of these are special cases of component diagrams and deployment diagrams, respectively, and are discussed elsewhere. In these cases, component instances and node instances are things that live outside the running system and are amenable to some degree of forward engineering.

Reverse engineering (the creation of a model from code) an object diagram is a very different thing. In fact, while you are debugging your system, this is something that you or your tools will do all the time. For example, if you are chasing down a dangling link, you'll want to literally or mentally draw an object diagram of the affected objects to see where, at a given moment in time, an object's state or its relationship to other objects is broken.

To reverse engineer an object diagram,

-
- Chose the target you want to reverse engineer. Typically, you'll set your context inside an operation or relative to an instance of one particular class.
- Using a tool or simply walking through a scenario, stop execution at a certain moment in time.
- Identify the set of interesting objects that collaborate in that context and render them in an object diagram.
- As necessary to understand their semantics, expose these object's states.
- As necessary to understand their semantics, identify the links that exist among these objects.
- If your diagram ends up overly complicated, prune it by eliminating objects that are not germane to the questions about the scenario you need answered. If your diagram is too simplistic, expand the neighbors of certain interesting objects and expose each object's state more deeply.

Unit-3

Basic Behavioral Modeling-I

Syllabus : Interactions, Interaction diagrams, Use cases, Use case Diagrams, Activity diagrams

Interactions

Terms and Concepts

An *interaction* is a behavior that comprises a set of messages exchanged among a set of objects within a context to accomplish a purpose. A *message* is a specification of a communication between objects that conveys information with the expectation that activity will ensue.

Context

You may find an interaction wherever objects are linked to one another. You'll find interactions in the collaboration of objects that exist in the context of your system or subsystem. You will also find interactions in the context of an operation. Finally, you'll find interactions in the context of a class.

Most often, you'll find interactions in the collaboration of objects that exist in the context of your system or subsystem as a whole. For example, in a system for Web commerce, you'll find objects on the client (such as instances of the classes **BookOrder** and **OrderForm**) interacting with one another. You'll also find objects on the client (again, such as instances of **BookOrder**) interacting with objects on the server (such as instances of **BackOrderManager**). These interactions therefore not only involve localized collaborations of objects (such as the interactions surrounding **OrderForm**), but they may also cut across many conceptual levels of your system (such as the interactions surrounding **BackOrderManager**).

You'll also find interactions among objects in the implementation of an operation. The parameters of an operation, any variables local to the operation, and any objects global to the operation (but still visible to the operation) may interact with one another to carry out the algorithm of that operation's implementation. For example, invoking the operation **moveToPosition(p :Position)** defined for a class in a mobile robot will involve the interaction of a parameter (**p**), an object global to the operation (such as the object **currentPosition**), and possibly several local objects (such as local variables used by the operation to calculate intermediate points in a path to the new position).

Finally, you will find interactions in the context of a class. You can use interactions to visualize, specify, construct, and document the semantics of a class. For example, to understand the meaning of a class **RayTraceAgent**, you might create interactions that show how the attributes of that class collaborate with one another (and with objects global to instances of the class and with parameters defined in the class's operations).

Objects and Roles

The objects that participate in an interaction are either concrete things or prototypical things. As a concrete thing, an object represents something in the real world. For example, **p**, an instance of the class **Person**, might denote a particular human. Alternately, as a prototypical thing, **p** might represent any instance of **Person**.

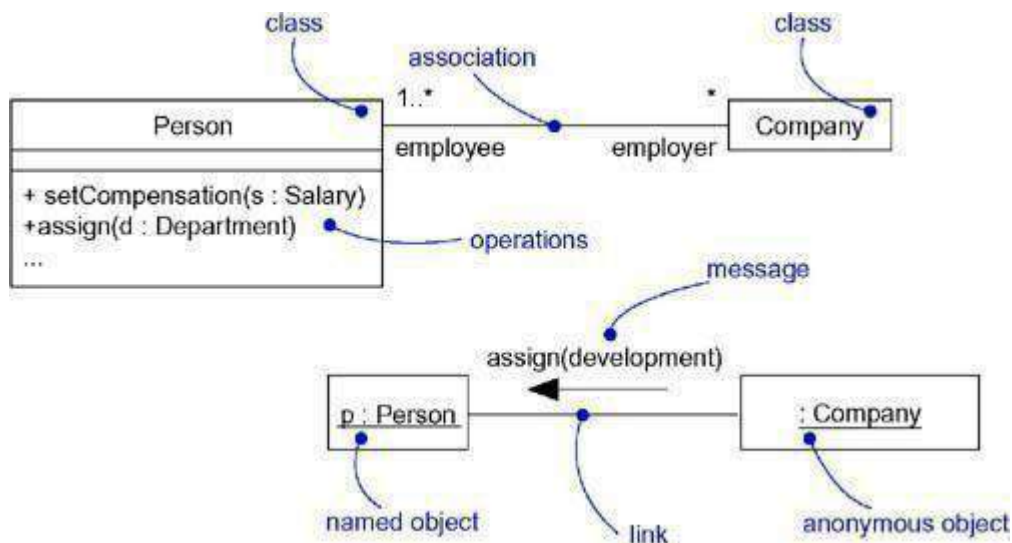
In the context of an interaction, you may find instances of classes, components, nodes, and use cases. Although abstract classes and interfaces, by definition, may not have any direct instances, you may find instances of these things in an interaction. Such instances do not represent direct instances of the abstract class or of the interface, but may represent, respectively, indirect (or prototypical) instances of any concrete children of the abstract class or of some concrete class that realizes that interface.

You can think of an object diagram as a representation of the static aspect of an interaction, setting the stage for the interaction by specifying all the objects that work together. An interaction goes further by introducing a dynamic sequence of messages that may pass along the links that connect these objects

Links

A link is a semantic connection among objects. In general, a link is an instance of an association. As Figure shows, wherever a class has an association to another class, there may be a link between the instances of the two classes; wherever there is a link between two objects, one object can send a message to the other object.

Figure Links and Associations



A link specifies a path along which one object can dispatch a message to another (or the same) object. Most of the time, it is sufficient to specify that such a path exists. If you need to be more precise about how that path exists, you can adorn the appropriate end of the link with any of the following standard stereotypes.

	Specifies that the corresponding object is visible by association
association	
self	Specifies that the corresponding object is visible because it is the dispatcher of the operation
global	Specifies that the corresponding object is visible because it is in an enclosing scope
local	Specifies that the corresponding object is visible because it is in a local scope
parameter	Specifies that the corresponding object is visible because it is a parameter

Messages

Call	Invokes an operation on an object; an object may send a message to itself, resulting in the local invocation of an operation
Return	Returns a value to the caller
Send	Sends a signal to an object
Create	Creates an object
Destroy	Destroys an object; an object may commit suicide by destroying itself

Suppose you have a set of objects and a set of links that connect those objects. If that's all you have, then you have a completely static model that can be represented by an object diagram. Object diagrams model the state of a society of objects at a given moment in time and are useful when you want to visualize, specify, construct, or document a static object structure.

Suppose you want to model the changing state of a society of objects over a period of time. Think of it as taking a motion picture of a set of objects, each frame representing a successive moment in time. If these objects are not totally idle, you'll see objects passing messages to other objects, sending events, and invoking operations. In addition, at each frame, you can explicitly visualize the current state and role of individual instances.

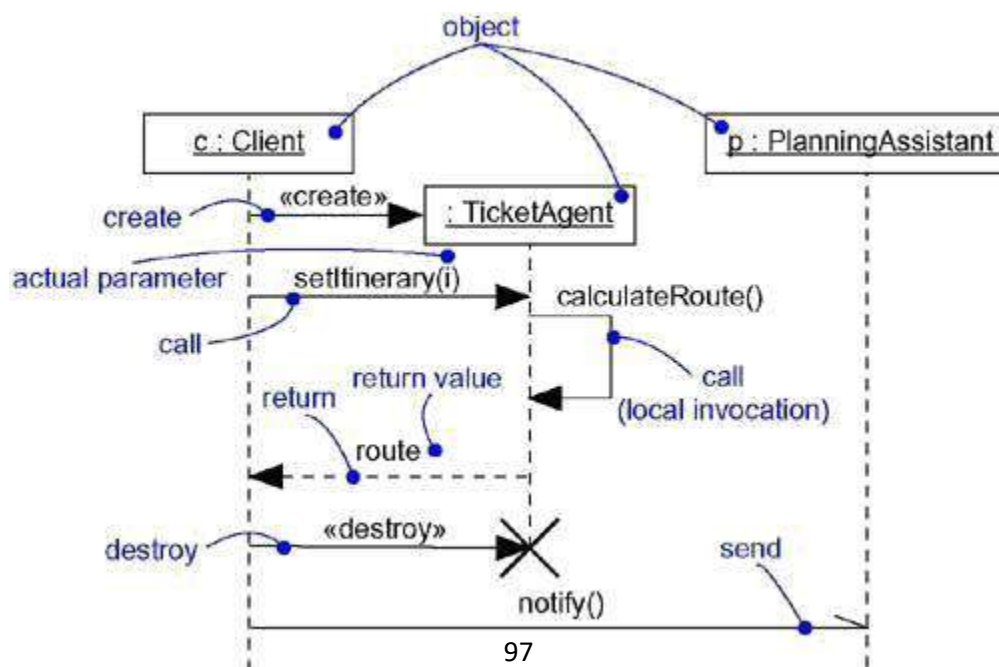
A message is the specification of a communication among objects that conveys information with the expectation that activity will ensue. The receipt of a message instance may be considered an instance of an event.

When you pass a message, the action that results is an executable statement that forms an abstraction of a computational procedure. An action may result in a change in state.

In the UML, you can model several kinds of actions.

The UML provides a visual distinction among these kinds of messages, as [Figure](#) shows.

Figure Messages



The most common kind of message you'll model is the call, in which one object invokes an operation of another (or the same) object. An object can't just call any random operation. If an object, such as **c** in the example above, calls the operation **setItinerary** on an instance of the class **TicketAgent**, the operation **setItinerary** must not only be defined for the class **TicketAgent** (that is, it must be declared in the class **TicketAgent** or one of its parents), it must also be visible to the caller **c**.

When an object calls an operation or sends a signal to another object, you can provide actual parameters to the message. Similarly, when an object returns control to another object, you can model the return value, as well.

Sequencing

When an object passes a message to another object (in effect, delegating some action to the receiver), the receiving object might in turn send a message to another object, which might send a message to yet a different object, and so on. This stream of messages forms a sequence. Any sequence must have a beginning; the start of every sequence is rooted in some process or thread. Furthermore, any sequence will continue as long as the process or thread that owns it lives. A nonstop system, such as you might find in real time device control, will continue to execute as long as the node it runs on is up.

Each process and thread within a system defines a distinct flow of control, and within each flow, messages are ordered in sequence by time. To better visualize the sequence of a message, you can explicitly model the order of the message relative to the start of the sequence by prefixing the message with a sequence number set apart by a colon separator.

Most commonly, you can specify a procedural or nested flow of control, rendered using a filled solid arrowhead, as Figure shows. In this case, the message **findAt** is specified as the first message nested in the second message of the sequence (**2.1**).

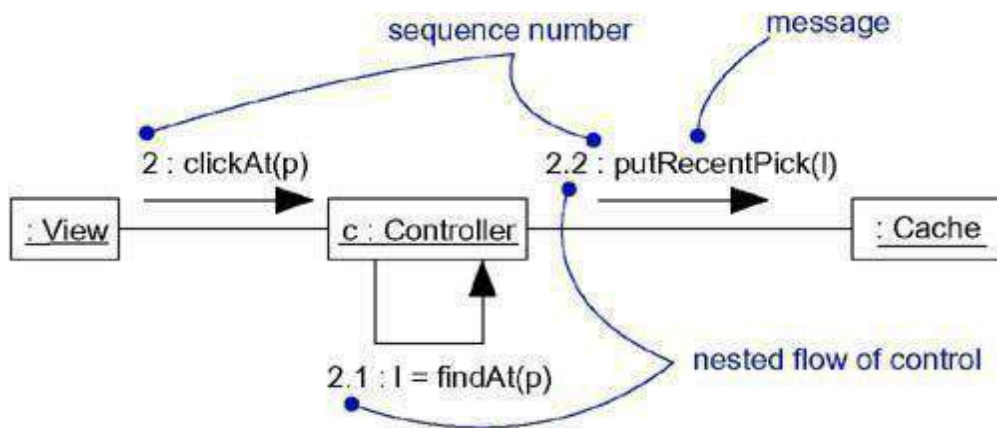


Figure Procedural Sequence

Less common but also possible, as Figure shows, you can specify a flat flow of control, rendered using a stick arrowhead, to model the nonprocedural progression of control from step to step. In this case, the message **assertCall** is specified as the second message in the sequence.

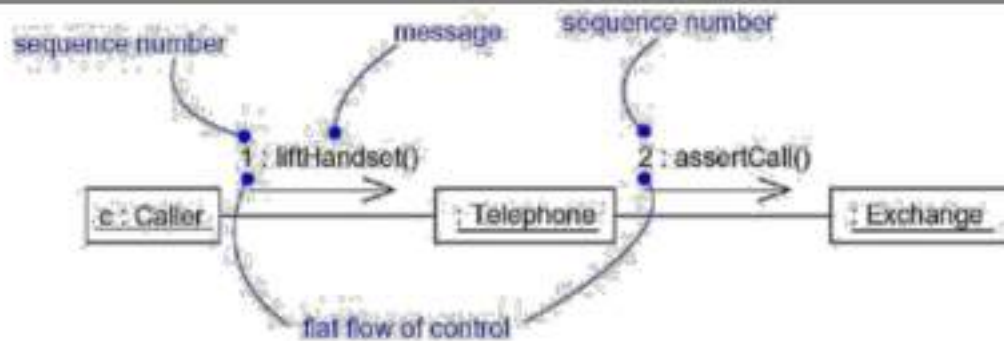


Figure Flat Sequence

When you are modeling interactions that involve multiple flows of control, it's especially important to identify the process or thread that sent a particular message. In the UML, you can distinguish one flow of control from another by prefixing a message's sequence number with the name of the process or thread that sits at the root of the sequence. For example, the expression

D5 :ejectHatch(3)

specifies that the operation **ejectHatch** is dispatched (with the actual argument **3**) as the fifth message in the sequence rooted by the process or thread named **D**.

Not only can you show the actual arguments sent along with an operation or a signal in the context of an interaction, you can show the return values of a function as well. As the following expression shows, the value **p** is returned from the operation **find**, dispatched with the actual parameter "**Rachelle**". This is a nested sequence, dispatched as the second message nested in the third message nested in the first message of the sequence. In the same diagram, **p** can then be used as an actual parameter in other messages.

Creation, Modification, and Destruction

Most of the time, the objects you show participating in an interaction exist for the entire duration of the interaction. However, in some interactions, objects may be created (specified by a **create** message) and destroyed (specified by a **destroy** message). The same is true of links: the relationships among objects may come and go. To specify if an object or link enters and/or leaves during an interaction, you can attach one of the following constraints to the element:

new	Specifies that the instance or link is created during execution of the enclosing interaction
destroyed the en	Specifies that the instance or link is destroyed prior to completion of execution of losing interaction
transient interact	Specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution

During an interaction, an object typically changes the values of its attributes, its state, or its roles. You can represent the modification of an object by replicating the object in the interaction (with possibly different attribute values, state, or roles). On a sequence diagram, you'd place each variant of the object on the same lifeline. In an interaction diagram, you'd connect each variant with a **become** message.

Representation

When you model an interaction, you typically include both objects (each one playing a specific role) and messages (each one representing the communication between objects, with some resulting action).

You can visualize those objects and messages involved in an interaction in two ways: by emphasizing the time ordering of its messages, and by emphasizing the structural organization of the objects that send and

receive messages. In the UML, the first kind of representation is called a sequence diagram; the second kind of representation is called a collaboration diagram. Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams.

Sequence diagrams and collaboration diagrams are largely isomorphic, meaning that you can take one and transform it into the other without loss of information. There are some visual differences, however. First, sequence diagrams permit you to model the lifeline of an object. An object's lifeline represents the existence of the object at a particular time, possibly covering the object's creation and destruction. Second, collaboration diagrams permit you to model the structural links that may exist among the objects in an interaction.

Common Modeling Techniques

Modeling a Flow of Control

When you model an interaction, you essentially build a storyboard of the actions that take place among a set of objects. Techniques such as CRC cards are particularly useful in helping you to discover and think about such interactions.

To model a flow of control,

- Set the context for the interaction, whether it is the system as a whole, a class, or an individual operation.
- Set the stage for the interaction by identifying which objects play a role; set their initial properties, including their attribute values, state, and role.
- If your model emphasizes the structural organization of these objects, identify the links that connect them, relevant to the paths of communication that take place in this interaction. Specify the nature of the links using the UML's standard stereotypes and constraints, as necessary.
- In time order, specify the messages that pass from object to object. As necessary, distinguish the different kinds of messages; include parameters and return values to convey the necessary detail of this interaction.
- Also to convey the necessary detail of this interaction, adorn each object at every moment in time with its state and role.

For example, Figure shows a set of objects that interact in the context of a publish and subscribe mechanism (an instance of the observer design pattern). This figure includes three objects: **p** (a **StockQuotePublisher**), **s1**, and **s2** (both instances of **StockQuoteSubscriber**). This figure is an example of a sequence diagram, which emphasizes the time order of messages.

Figure Flow of Control by Time

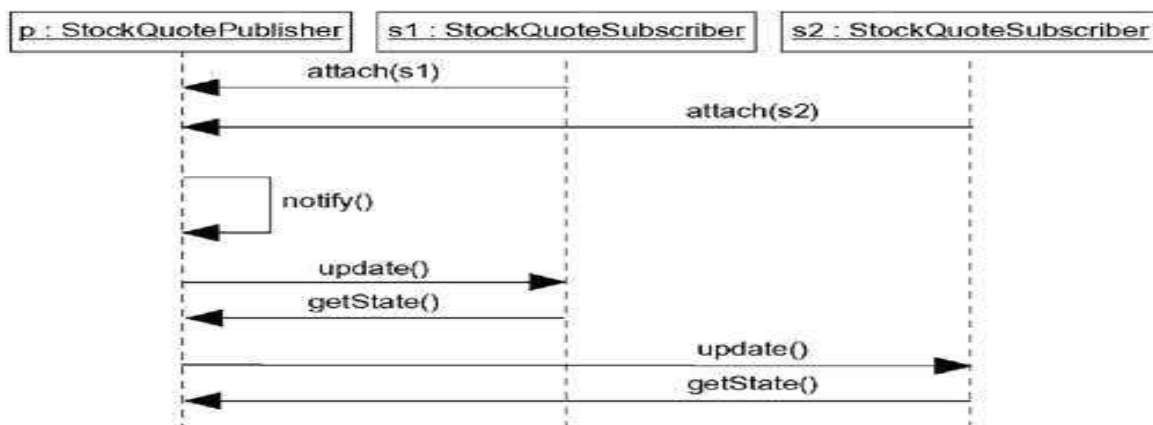
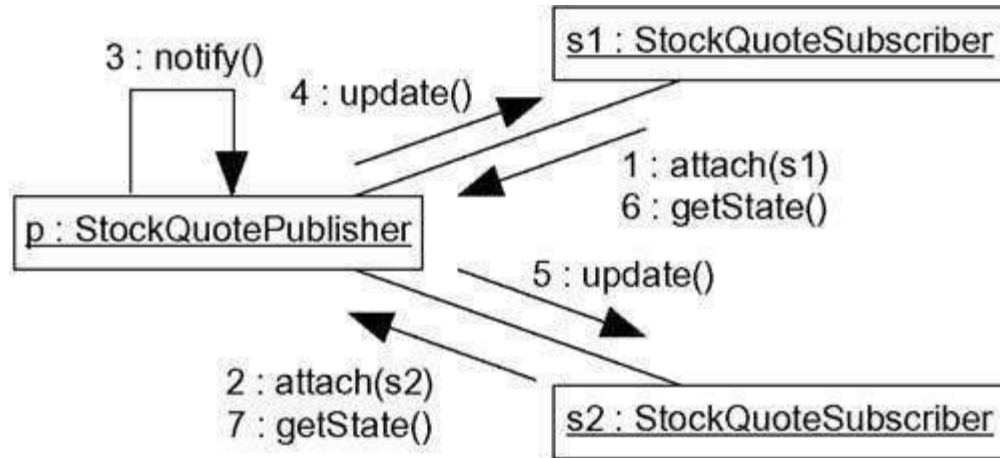


Figure is semantically equivalent to the previous one, but it is drawn as a collaboration diagram, which emphasizes the structural organization of the objects. This figure shows the same flow of control, but it also provides a visualization of the links among these objects.

Figure Flow of Control by Organization



Interaction Diagrams

Terms and Concepts

An *interaction diagram* shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis. A *collaboration diagram* is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Graphically, a collaboration diagram is a collection of vertices and arcs.

Common Properties

An interaction diagram is just a special kind of diagram and shares the same common properties as do all **other diagrams**• a name and graphical contents that are a projection into a model. What distinguishes an interaction diagram from all other kinds of diagrams is its particular content.

Contents

Interaction diagrams commonly contain

- Objects
- Links
- Messages

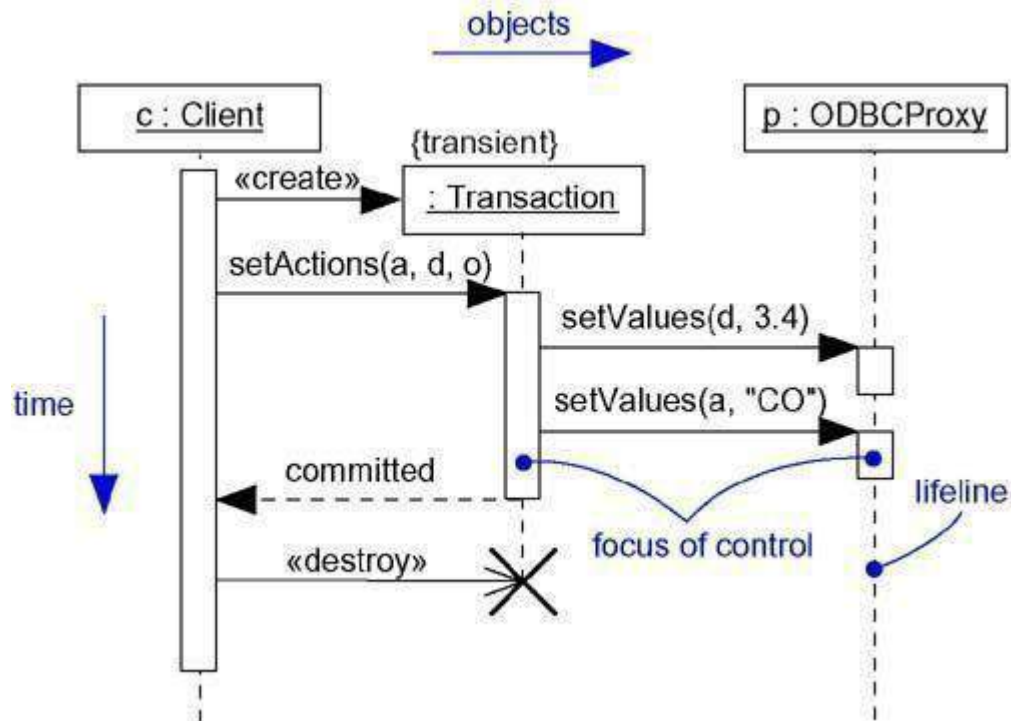
Like all other diagrams, interaction diagrams may contain notes and constraints.

Sequence Diagrams

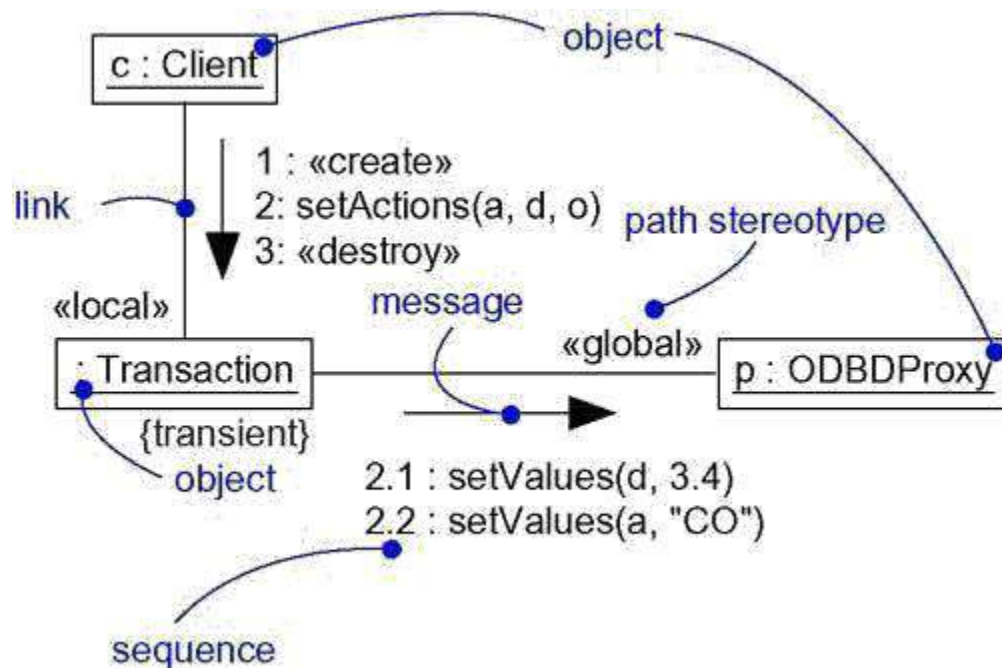
A sequence diagram emphasizes the time ordering of messages. As Figure shows, you form a sequence diagram by first placing the objects that participate in the interaction at the top of your diagram, across the

X axis. Typically, you place the object that initiates the interaction at the left, and increasingly more subordinate objects to the right. Next, you place the messages that these objects send and receive along the Y axis, in order of increasing time from top to bottom. This gives the reader a clear visual cue to the flow of control over time.

Figure Sequence Diagram



Sequence diagrams have two features that distinguish them from collaboration diagrams.



First, there is the object lifeline. An object lifeline is the vertical dashed line that represents the existence of an object over a period of time. Most objects that appear in an interaction diagram will be in existence for the duration of the interaction, so these objects are all aligned at the top of the diagram, with their lifelines drawn from the top of the diagram to the bottom. Objects may be created during the interaction. Their lifelines start with the receipt of the message stereotyped as **create**. Objects may be destroyed during the interaction. Their lifelines end with the receipt of the message stereotyped as **destroy** (and are given the visual cue of a large **X**, marking the end of their lives).

Second, there is the focus of control. The focus of control is a tall, thin rectangle that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure. The top of the rectangle is aligned with the start of the action; the bottom is aligned with its completion (and can be marked by a return message). You can show the nesting of a focus of control (caused by recursion, a call to a self- operation, or by a callback from another object) by stacking another focus of control slightly to the right of its parent (and can do so to an arbitrary depth). If you want to be especially precise about where the focus of control lies, you can also shade the region of the rectangle during which the object's method is actually computing (and control has not passed to another object).

Collaboration Diagrams

A collaboration diagram emphasizes the organization of the objects that participate in an interaction. As Figure shows, you form a collaboration diagram by first placing the objects that participate in the interaction as the vertices in a graph. Next, you render the links that connect these objects as the arcs of this graph. Finally, you adorn these links with the messages that objects send and receive. This gives the reader a clear visual cue to the flow of control in the context of the structural organization of objects that collaborate.

Figure Collaboration Diagram

Collaboration diagrams have two features that distinguish them from sequence diagrams.

First, there is the path. To indicate how one object is linked to another, you can attach a path stereotype to the far end of a link (such as **»local**, indicating that the designated object is local to the sender). Typically, you will only need to render the path of the link explicitly for **local**, **parameter**, **global**, and **self** (but not **association**) paths.

Second, there is the sequence number. To indicate the time order of a message, you prefix the message with a number (starting with the message numbered **1**), increasing monotonically for each new message in the flow of control (**2**, **3**, and so on). To show nesting, you use Dewey decimal numbering (**1** is the first message; **1.1** is the first message nested in message **1**; **1.2** is the second message nested in message **1**; and so on). You can show nesting to an arbitrary depth. Note also that, along the same link, you can show many messages (possibly being sent from different directions), and each will have a unique sequence number.

Most of the time, you'll model straight, sequential flows of control. However, you can also model more-complex flows, involving iteration and branching. An iteration represents a repeated sequence of messages. To model an iteration, you prefix the sequence number of a message with an iteration expression such as ***[i := 1..n]** (or just ***** if you want to indicate iteration but don't want to specify its details). An iteration indicates that the message (and any nested messages) will be repeated in accordance with the given expression. Similarly, a condition represents a message whose execution is contingent on the evaluation of a Boolean condition. To model a condition, you prefix the sequence number of a message with a condition clause, such as **[x > 0]**. The alternate paths of a branch will have the same sequence number, but each path must be uniquely distinguishable by a nonoverlapping condition.

For both iteration and branching, the UML does not prescribe the format of the expression inside the brackets; you can use pseudocode or the syntax of a specific programming language.

Semantic Equivalence

Because they both derive from the same information in the UML's metamodel, sequence diagrams and collaboration diagrams are semantically equivalent. As a result, you can take a diagram in one form and convert it to the other without any loss of information, as you can see in the previous two figures, which are semantically equivalent. However, this does not mean that both diagrams will explicitly visualize the same information. For example, in the previous two figures, the collaboration diagram shows how the objects are linked (note the **»local** and **»global** stereotypes), whereas the corresponding sequence diagram does not. Similarly, the sequence diagram shows message return (note the return value **committed**), but the corresponding collaboration diagram does not. In both cases, the two diagrams share the same underlying model, but each may render some things the other does not.

Common Uses

You use interaction diagrams to model the dynamic aspects of a system. These dynamic aspects may involve the interaction of any kind of instance in any view of a system's architecture, including instances of classes (including active classes), interfaces, components, and nodes.

When you use an interaction diagram to model some dynamic aspect of a system, you do so in the context of the system as a whole, a subsystem, an operation, or a class. You can also attach interaction diagrams to use cases (to model a scenario) and to collaborations (to model the dynamic aspects of a society of objects).

When you model the dynamic aspects of a system, you typically use interaction diagrams in two ways.

1. To model flows of control by time ordering

Here you'll use sequence diagrams. Modeling a flow of control by time ordering emphasizes the passing of messages as they unfold over time, which is a particularly useful way to visualize dynamic behavior in the context of a use case scenario. Sequence diagrams do a better job of visualizing simple iteration and branching than do collaboration diagrams.

2. To model flows of control by organization

Here you'll use collaboration diagrams. Modeling a flow of control by organization emphasizes the structural relationships among the instances in the interaction, along which messages may be

passed. Collaboration diagrams do a better job of visualizing complex iteration and branching and of visualizing multiple concurrent flows of control than do sequence diagrams.

Common Modeling Techniques

Modeling Flows of Control by Time Ordering

Consider the objects that live in the context of a system, subsystem, operation or class. Consider also the objects and roles that participate in a use case or collaboration. To model a flow of control that winds through these objects and roles, you use an interaction diagram; to emphasize the passing of messages as they unfold over time, you use a sequence diagram, a kind of interaction diagram.

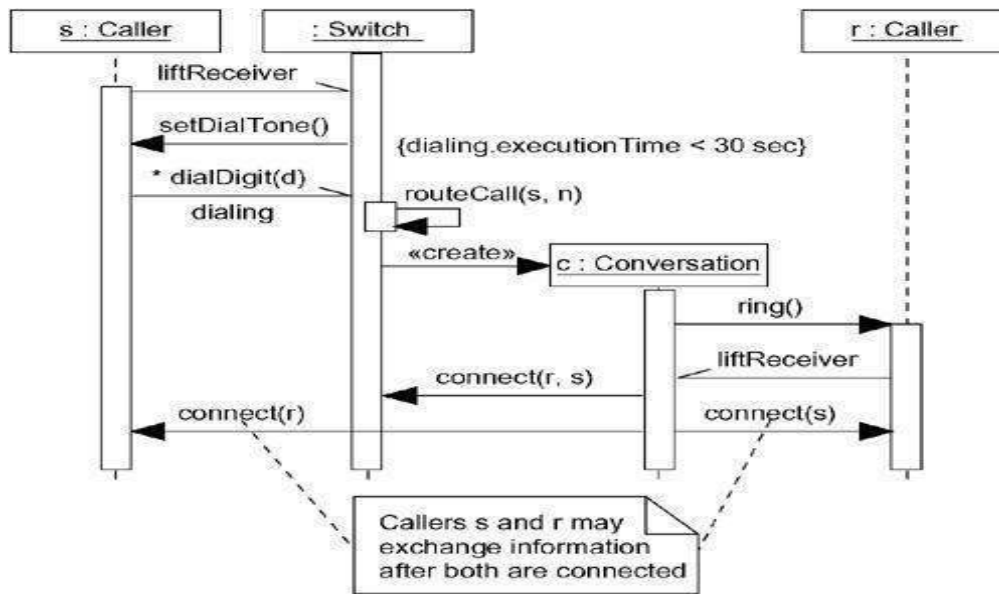
To model a flow of control by time ordering,

- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the sequence diagram from left to right, placing the more important objects to the left and their neighboring objects to the right.
- Set the lifeline for each object. In most cases, objects will persist through the entire interaction. For those objects that are created and destroyed during the interaction, set their lifelines, as appropriate, and explicitly indicate their birth and death with appropriately stereotyped messages.
- Starting with the message that initiates this interaction, lay out each subsequent message from top to bottom between the lifelines, showing each message's properties (such as its parameters), as necessary to explain the semantics of the interaction.
- If you need to visualize the nesting of messages or the points in time when actual computation is taking place, adorn each object's lifeline with its focus of control.
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

A single sequence diagram can show only one flow of control (although you can show simple variations by using the UML's notation for iteration and branching). Typically, you'll have a number of interaction diagrams, some of which are primary and others that show alternative paths or exceptional conditions. You can use packages to organize these collections of sequence diagrams, giving each diagram a suitable name to distinguish it from its siblings.

For example, Figure shows a sequence diagram that specifies the flow of control involved in initiating a simple, two-party phone call. At this level of abstraction, there are four objects involved: two **Callers** (**s** and **r**), an unnamed telephone **Switch**, and **c**, the reification of the **Conversation** between the two parties. The sequence begins with one **Caller** (**s**) dispatching a signal (**liftReceiver**) to the **Switch** object. In turn, the **Switch** calls **setDialTone** on the **Caller**, and the **Caller** iterates on the message **dialDigit**. Note that this message has a timing mark (**dialing**) that is used in a timing constraint (its **executionTime** must be less than 30 seconds). This diagram does not indicate what happens if this time constraint is violated. For that you could include a branch or a completely separate sequence diagram. The **Switch** object then calls itself with the message **routeCall**. It then creates a **Conversation** object (**c**), to which it delegates the rest of the work. Although not shown in this interaction, **c** would have the additional responsibility of being a party in the switch's billing mechanism (which would be expressed in another interaction diagram). The **Conversation** object (**c**) rings the **Caller** (**r**), who asynchronously sends the message **liftReceiver**. The **Conversation** object then tells the **Switch** to **connect** the call, then tells both **Caller** objects to **connect**, after which they may exchange information, as indicated by the attached note.

Figure Modeling Flows of Control by Time Ordering



An interaction diagram can begin or end at any point of a sequence. A complete trace of the flow of control would be incredibly complex, so it's reasonable to break up parts of a larger flow into separate diagrams.

Modeling Flows of Control by Organization

Consider the objects that live in the context of a system, subsystem, operation, or class. Consider also the objects and roles that participate in a use case or collaboration. To model a flow of control that winds through these objects and roles, you use an interaction diagram; to show the passing of messages in the context of that structure, you use a collaboration diagram, a kind of interaction diagram.

To model a flow of control by organization,

- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the collaboration diagram as vertices in a graph, placing the more important objects in the center of the diagram and their neighboring objects to the outside.
- Set the initial properties of each of these objects. If the attribute values, tagged values, state, or role of any object changes in significant ways over the duration of the interaction, place a duplicate object on the diagram, update it with these new values, and connect them by a message stereotyped as **become** or **copy** (with a suitable sequence number).
- Specify the links among these objects, along which messages may pass.
 1. Lay out the association links first; these are the most important ones, because they represent structural connections.
 2. Lay out other links next, and adorn them with suitable path stereotypes (such as **global** and **local**) to explicitly specify how these objects are related to one another.
- Starting with the message that initiates this interaction, attach each subsequent message to the appropriate link, setting its sequence number, as appropriate. Show nesting by using Dewey

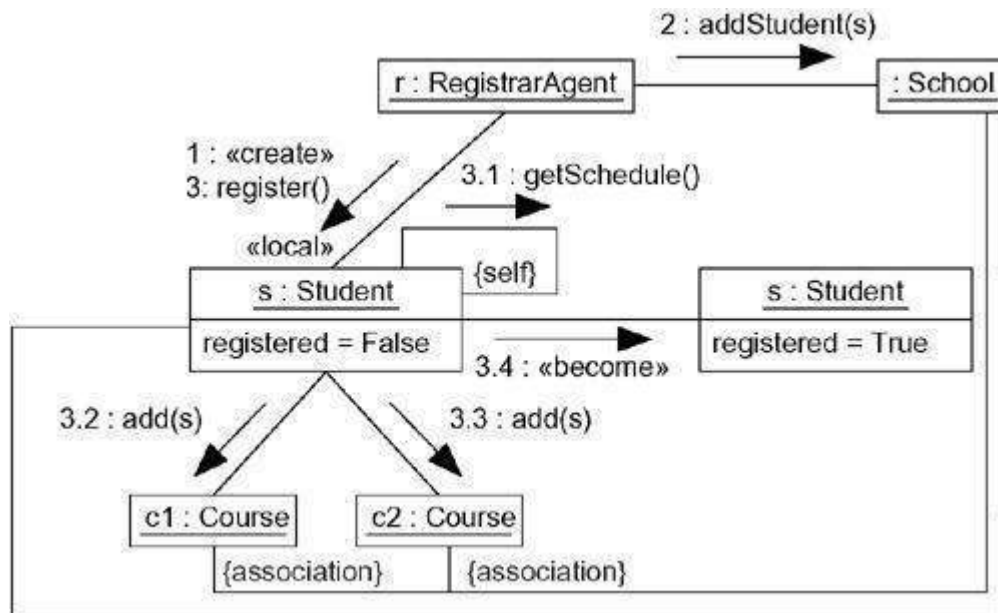
decimal numbering.

- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

As with sequence diagrams, a single collaboration diagram can show only one flow of control (although you can show simple variations by using the UML's notation for interaction and branching). Typically, you'll have a number of such interaction diagrams, some of which are primary and others that show alternative paths or exceptional conditions. You can use packages to organize these collections of collaboration diagrams, giving each diagram a suitable name to distinguish it from its siblings.

For example, Figure—shows a collaboration diagram that specifies the flow of control involved in registering a new student at a school, with an emphasis on the structural relationships among these objects. You see five objects: a **RegistrarAgent** (r), a **Student** (s), two **Course** objects (c1 and c2), and an unnamed **School** object. The flow of control is numbered explicitly. Action begins with the **RegistrarAgent** creating a **Student** object, adding the student to the school (the message **addStudent**), then telling the **Student** object to register itself. The **Student** object then invokes **getSchedule** on itself, presumably obtaining the **Course** objects for which it must register. The **Student** object then adds itself to each **Course** object. The flow ends with s rendered again, showing that it has an updated value for its **registered** attribute.

Figure Modeling Flows of Control by Organization



Note that this diagram shows a link between the **School** object and the two **Course** objects, plus another link between the **School** object and the **Student** object, although no messages are shown along these paths. These links help explain how the **Student** object can see the two **Course** objects to which it adds itself. **s**, **c1**, and **c2** are linked to the **School** via association, so **s** can find **c1** and **c2** during its call to **getSchedule** (which might return a collection of **Course** objects), indirectly through the **School** object.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for both sequence and collaboration diagrams, especially if the context of the diagram is an operation. For example, using the previous collaboration diagram, a reasonably clever forward engineering tool could generate the following Java code for the operation **register**, attached to the **Student** class.

```
public void register() { CourseCollection c =
    getSchedule(); for (inti = 0; i<c.size(); i++)
        c.item(i).add(this);
    this.registered = true;
}
```

"Reasonably clever" means the tool would have to realize that **getSchedule** returns a **CourseCollection** object, which it could determine by looking at the operation's signature. Bywalking across the contents of this object using a standard iteration idiom (which the tool could know about implicitly), the code could then generalize to any number of course offerings.

Reverse engineering (the creation of a model from code) is also possible for both sequence and collaboration diagrams, especially if the context of the code is the body of an operation. Segments of the previous diagram could have been produced by a tool from a prototypical execution of the **register** operation.

When you are modeling interactions that involve multiple flows of control, it's especially important to identify the process or thread that sent a particular message. In the UML, you can distinguish one flow of control from another by prefixing a message's sequence number with the name of the process or thread that sits at the root of the sequence. For example, the expression

D5 :ejectHatch(3)

specifies that the operation **ejectHatch** is dispatched (with the actual argument **3**) as the fifth message in the sequence rooted by the process or thread named **D**.

Not only can you show the actual arguments sent along with an operation or a signal in the context of an interaction, you can show the return values of a function as well. As the following expression shows, the value **p** is returned from the operation **find**, dispatched with the actual parameter "**Rachelle**". This is a nested sequence, dispatched as the second message nested in the third message nested in the first message of the sequence. In the same diagram, **p** can then be used as an actual parameter in other messages.

Creation, Modification, and Destruction

Most of the time, the objects you show participating in an interaction exist for the entire duration of the interaction. However, in some interactions, objects may be created (specified by a **create** message) and destroyed (specified by a **destroy** message). The same is true of links: the relationships among objects may come and go. To specify if an object or link enters and/or leaves during an interaction, you can attach one of the following constraints to the element:

new	Specifies that the instance or link is created during execution of the enclosing interaction
destroyed	Specifies that the instance or link is destroyed prior to completion of execution of the enclosing interaction
transient	Specifies that the instance or link is created during execution of the enclosing interaction but is destroyed before completion of execution

During an interaction, an object typically changes the values of its attributes, its state, or its roles. You can represent the modification of an object by replicating the object in the interaction (with possibly different attribute values, state, or roles). On a sequence diagram, you'd place each variant of the object on the same lifeline. In an interaction diagram, you'd connect each variant with a **become** message.

Representation

When you model an interaction, you typically include both objects (each one playing a specific role) and messages (each one representing the communication between objects, with some resulting action).

You can visualize those objects and messages involved in an interaction in two ways: by emphasizing the time ordering of its messages, and by emphasizing the structural organization of the objects that send and receive messages. In the UML, the first kind of representation is called a sequence diagram; the second kind of representation is called a collaboration diagram. Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams.

Sequence diagrams and collaboration diagrams are largely isomorphic, meaning that you can take one and transform it into the other without loss of information. There are some visual differences, however. First, sequence diagrams permit you to model the lifeline of an object. An object's lifeline represents the existence of the object at a particular time, possibly covering the object's creation and destruction. Second, collaboration diagrams permit you to model the structural links that may exist among the objects in an interaction.

Common Modeling Techniques

Modeling a Flow of Control

The most common purpose for which you'll use interactions is to model the flow of control that characterizes the behavior of a system as a whole, including use cases, patterns, mechanisms, and frameworks, or the behavior of a class or an individual operation. Whereas classes, interfaces, components, nodes, and their relationships model the static aspects of your system, interactions model its dynamic aspects.

When you model an interaction, you essentially build a storyboard of the actions that take place among a set of objects. Techniques such as CRC cards are particularly useful in helping you to discover and think about such interactions.

To model a flow of control,

- Set the context for the interaction, whether it is the system as a whole, a class, or an individual operation.
- Set the stage for the interaction by identifying which objects play a role; set their initial properties, including their attribute values, state, and role.
- If your model emphasizes the structural organization of these objects, identify the links that connect them, relevant to the paths of communication that take place in this interaction. Specify the nature of the links using the UML's standard stereotypes and constraints, as necessary.
- In time order, specify the messages that pass from object to object. As necessary, distinguish the different kinds of messages; include parameters and return values to convey the necessary detail of this interaction.
- Also to convey the necessary detail of this interaction, adorn each object at every

moment in time with its state and role.

For example, Figure shows a set of objects that interact in the context of a publish and subscribe mechanism (an instance of the observer design pattern). This figure includes three objects: **p** (a **StockQuotePublisher**), **s1**, and **s2** (both instances of **StockQuoteSubscriber**). This figure is an example of a sequence diagram, which emphasizes the time order of messages.

Figure Flow of Control by Time

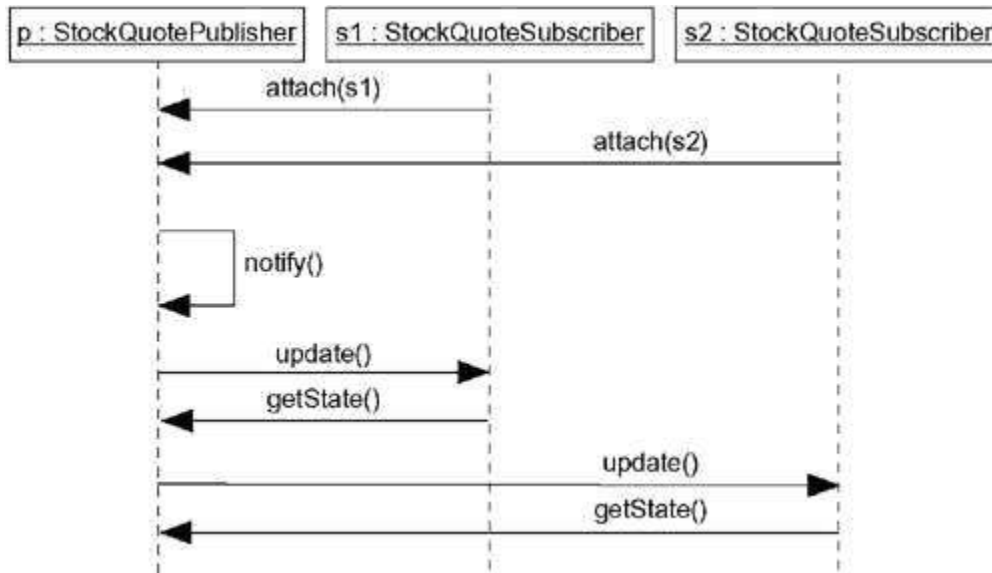
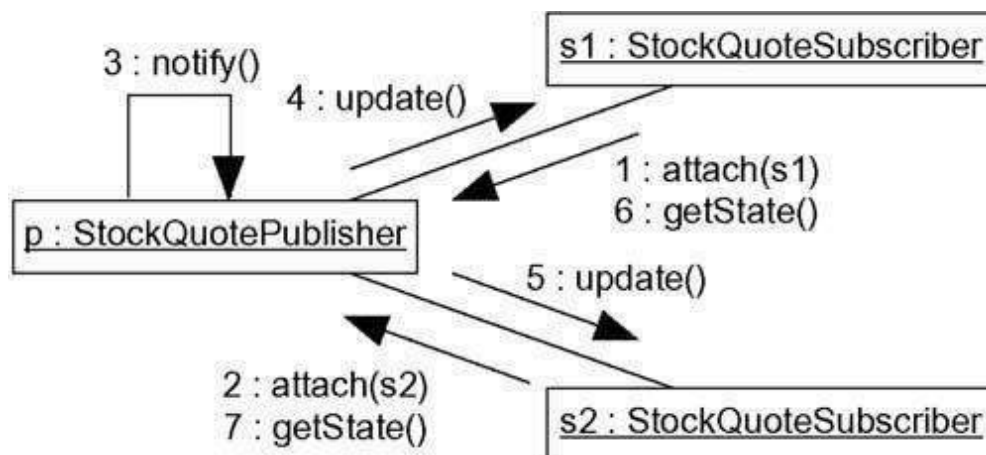


Figure is semantically equivalent to the previous one, but it is drawn as a collaboration diagram, which emphasizes the structural organization of the objects. This figure shows the same flow of control, but it also provides a visualization of the links among these objects.

Figure Flow of Control by Organization



Interaction Diagrams

Terms and Concepts

An *interaction diagram* shows an interaction, consisting of a set of objects and their relationships, including the messages that may be dispatched among them. A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis. A *collaboration diagram* is an interaction diagram that emphasizes the structural organization of the objects that send and receive messages. Graphically, a collaboration diagram is a collection of vertices and arcs.

Common Properties

An interaction diagram is just a special kind of diagram and shares the same common properties as do all **other diagrams**• a name and graphical contents that are a projection into a model. What distinguishes an interaction diagram from all other kinds of diagrams is its particular content.

Contents

Interaction diagrams commonly contain

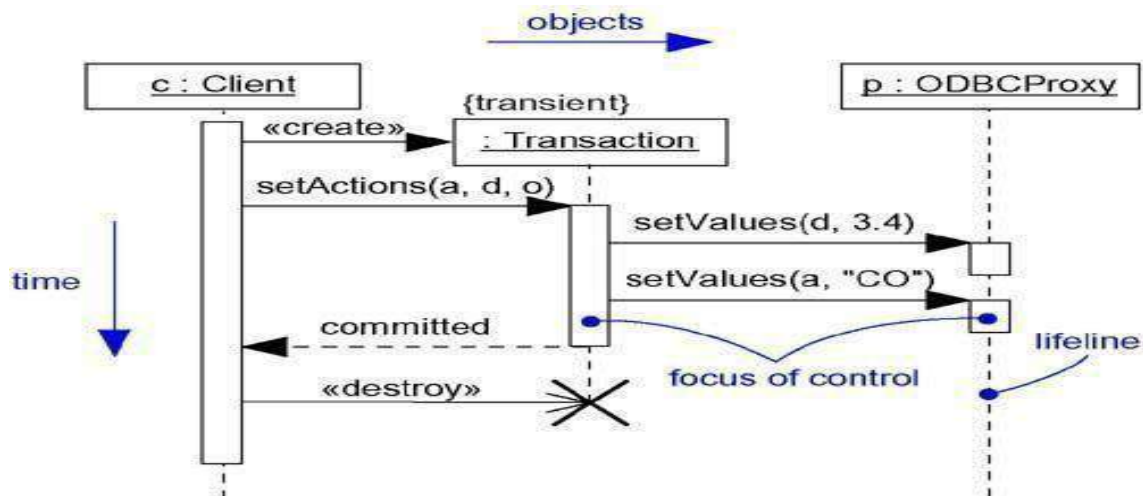
- Objects
- Links
- Messages

Like all other diagrams, interaction diagrams may contain notes and constraints.

Sequence Diagrams

A sequence diagram emphasizes the time ordering of messages. As Figure shows, you form a sequence diagram by first placing the objects that participate in the interaction at the top of your diagram, across the X axis. Typically, you place the object that initiates the interaction at the left, and increasingly more subordinate objects to the right. Next, you place the messages that these objects send and receive along the Y axis, in order of increasing time from top to bottom. This gives the reader a clear visual cue to the flow of control over time.

Figure Sequence Diagram



sequence diagrams have two features that distinguish them from collaboration diagrams.

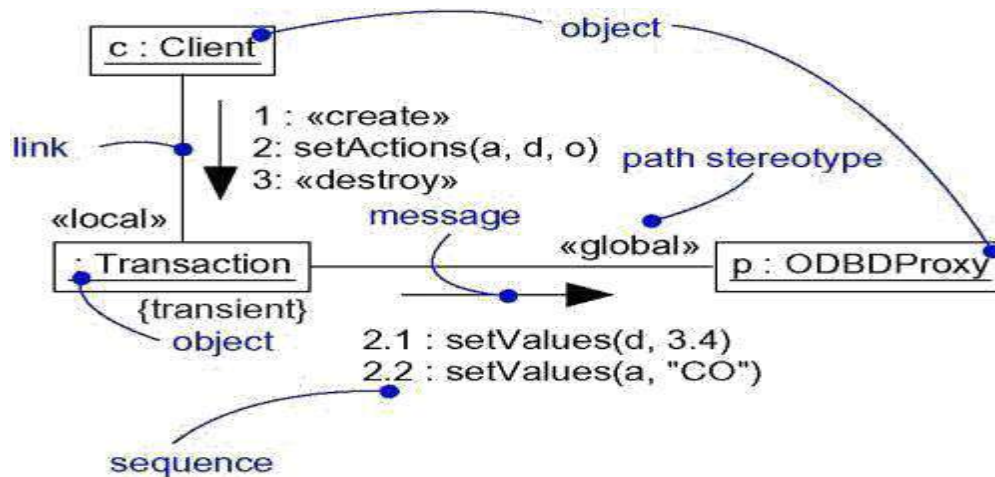
First, there is the object lifeline. An object lifeline is the vertical dashed line that represents the existence of an object over a period of time. Most objects that appear in an interaction diagram will be in existence for the duration of the interaction, so these objects are all aligned at the top of the diagram, with their lifelines drawn from the top of the diagram to the bottom. Objects may be created during the interaction. Their lifelines start with the receipt of the message stereotyped as **create**. Objects may be destroyed during the interaction. Their lifelines end with the receipt of the message stereotyped as **destroy** (and are given the visual cue of a large **X**, marking the end of their lives).

Second, there is the focus of control. The focus of control is a tall, thin rectangle that shows the period of time during which an object is performing an action, either directly or through a subordinate procedure. The top of the rectangle is aligned with the start of the action; the bottom is aligned with its completion (and can be marked by a return message). You can show the nesting of a focus of control (caused by recursion, a call to a self-operation, or by a callback from another object) by stacking another focus of control slightly to the right of its parent (and can do so to an arbitrary depth). If you want to be especially precise about where the focus of control lies, you can also shade the region of the rectangle during which the object's method is actually computing (and control has not passed to another object).

Collaboration Diagrams

A collaboration diagram emphasizes the organization of the objects that participate in an interaction. As Figure shows, you form a collaboration diagram by first placing the objects that participate in the interaction as the vertices in a graph. Next, you render the links that connect these objects as the arcs of this graph. Finally, you adorn these links with the messages that objects send and receive. This gives the reader a clear visual cue to the flow of control in the context of the structural organization of objects that collaborate.

Figure Collaboration Diagram



collaboration diagrams have two features that distinguish them from sequence diagrams.

First, there is the path. To indicate how one object is linked to another, you can attach a path stereotype to the far end of a link (such as **»local**, indicating that the designated object is local to the sender). Typically, you will only need to render the path of the link explicitly for **local**, **parameter**, **global**, and **self** (but not **association**) paths.

Second, there is the sequence number. To indicate the time order of a message, you prefix the message with a number (starting with the message numbered **1**), increasing monotonically for each new message in

the flow of control (2, 3, and so on). To show nesting, you use Dewey decimal numbering (1 is the first message; 1.1 is the first message nested in message 1; 1.2 is the second message nested in message 1; and so on). You can show nesting to an arbitrary depth. Note also that, along the same link, you can show many messages (possibly being sent from different directions), and each will have a unique sequence number.

Most of the time, you'll model straight, sequential flows of control. However, you can also model more-complex flows, involving iteration and branching. An iteration represents a repeated sequence of messages. To model an iteration, you prefix the sequence number of a message with an iteration expression such as `*[i := 1..n]` (or just `*` if you want to indicate iteration but don't want to specify its details). An iteration indicates that the message (and any nested messages) will be repeated in accordance with the given expression. Similarly, a condition represents a message whose execution is contingent on the evaluation of a Boolean condition. To model a condition, you prefix the sequence number of a message with a condition clause, such as `[x > 0]`. The alternate paths of a branch will have the same sequence number, but each path must be uniquely distinguishable by a nonoverlapping condition.

For both iteration and branching, the UML does not prescribe the format of the expression inside the brackets; you can use pseudocode or the syntax of a specific programming language.

Semantic Equivalence

Because they both derive from the same information in the UML's metamodel, sequence diagrams and collaboration diagrams are semantically equivalent. As a result, you can take a diagram in one form and convert it to the other without any loss of information, as you can see in the previous two figures, which are semantically equivalent. However, this does not mean that both diagrams will explicitly visualize the same information. For example, in the previous two figures, the collaboration diagram shows how the objects are linked (note the **»local** and **»global** stereotypes), whereas the corresponding sequence diagram does not. Similarly, the sequence diagram shows message return (note the return value **committed**), but the corresponding collaboration diagram does not. In both cases, the two diagrams share the same underlying model, but each may render some things the other does not.

Common Uses

You use interaction diagrams to model the dynamic aspects of a system. These dynamic aspects may involve the interaction of any kind of instance in any view of a system's architecture, including instances of classes (including active classes), interfaces, components, and nodes.

When you use an interaction diagram to model some dynamic aspect of a system, you do so in the context of the system as a whole, a subsystem, an operation, or a class. You can also attach interaction diagrams to use cases (to model a scenario) and to collaborations (to model the dynamic aspects of a society of objects).

When you model the dynamic aspects of a system, you typically use interaction diagrams in two ways.

1. To model flows of control by time ordering

Here you'll use sequence diagrams. Modeling a flow of control by time ordering emphasizes the passing of messages as they unfold over time, which is a particularly useful way to visualize dynamic behavior in the context of a use case scenario. Sequence diagrams do a better job of visualizing simple iteration and branching than do collaboration diagrams.

2. To model flows of control by organization

Here you'll use collaboration diagrams. Modeling a flow of control by organization emphasizes the structural relationships among the instances in the interaction, along which messages may be

passed. Collaboration diagrams do a better job of visualizing complex iteration and branching and of visualizing multiple concurrent flows of control than do sequence diagrams.

Common Modeling Techniques

Modeling Flows of Control by Time Ordering

Consider the objects that live in the context of a system, subsystem, operation or class. Consider also the objects and roles that participate in a use case or collaboration. To model a flow of control that winds through these objects and roles, you use an interaction diagram; to emphasize the passing of messages as they unfold over time, you use a sequence diagram, a kind of interaction diagram.

To model a flow of control by time ordering,

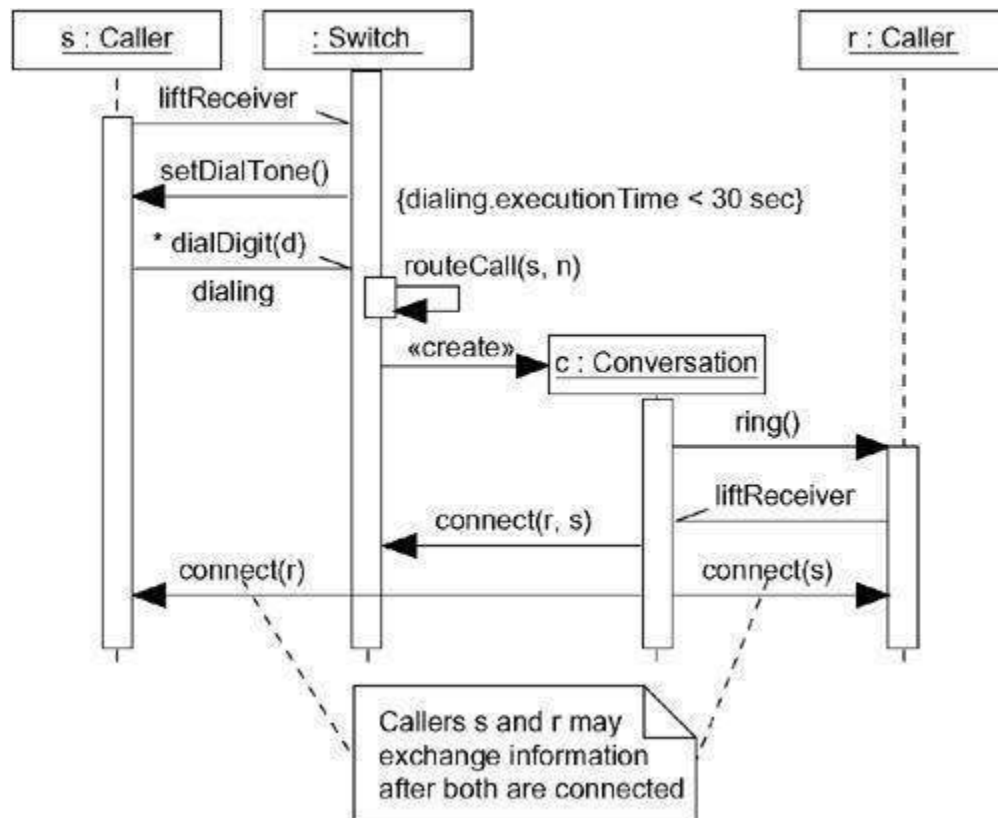
- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the sequence diagram from left to right, placing the more important objects to the left and their neighboring objects to the right.
- Set the lifeline for each object. In most cases, objects will persist through the entire interaction. For those objects that are created and destroyed during the interaction, set their lifelines, as appropriate, and explicitly indicate their birth and death with appropriately stereotyped messages.
- Starting with the message that initiates this interaction, lay out each subsequent message from top to bottom between the lifelines, showing each message's properties (such as its parameters), as necessary to explain the semantics of the interaction.
- If you need to visualize the nesting of messages or the points in time when actual computation is taking place, adorn each object's lifeline with its focus of control.
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

A single sequence diagram can show only one flow of control (although you can show simple variations by using the UML's notation for iteration and branching). Typically, you'll have a number of interaction diagrams, some of which are primary and others that show alternative paths or exceptional conditions. You can use packages to organize these collections of sequence diagrams, giving each diagram a suitable name to distinguish it from its siblings.

For example, Figure shows a sequence diagram that specifies the flow of control involved in initiating a simple, two-party phone call. At this level of abstraction, there are four objects involved: two **Callers** (**s** and **r**), an unnamed telephone **Switch**, and **c**, the reification of the **Conversation** between the two parties. The sequence begins with one **Caller** (**s**) dispatching a signal (**liftReceiver**) to the **Switch** object. In turn, the **Switch** calls **setDialTone** on the **Caller**, and the **Caller** iterates on the message **dialDigit**. Note that this message has a timing mark (**dialing**) that is used in a timing constraint (its **executionTime** must be less than 30 seconds). This diagram does not indicate what happens if this time constraint is violated. For that you could include a branch or a completely separate sequence diagram. The **Switch** object then calls itself with the message **routeCall**. It then creates a **Conversation** object (**c**), to which it delegates the rest of the work. Although not shown in this interaction, **c** would have the additional responsibility of being a party in the switch's billing mechanism (which would be expressed in another interaction diagram). The **Conversation** object (**c**) rings the **Caller** (**r**), who asynchronously sends the message **liftReceiver**. The **Conversation** object then tells the **Switch** to **connect** the call, then tells both **Caller** objects to **connect**,

after which they may exchange information, as indicated by the attached note.

Figure Modeling Flows of Control by Time Ordering



An interaction diagram can begin or end at any point of a sequence. A complete trace of the flow of control would be incredibly complex, so it's reasonable to break up parts of a larger flow into separate diagrams.

Modeling Flows of Control by Organization

Consider the objects that live in the context of a system, subsystem, operation, or class. Consider also the objects and roles that participate in a use case or collaboration. To model a flow of control that winds through these objects and roles, you use an interaction diagram; to show the passing of messages in the context of that structure, you use a collaboration diagram, a kind of interaction diagram.

To model a flow of control by organization,

- Set the context for the interaction, whether it is a system, subsystem, operation, or class, or one scenario of a use case or collaboration.
- Set the stage for the interaction by identifying which objects play a role in the interaction. Lay them out on the collaboration diagram as vertices in a graph, placing the more important objects in the center of the diagram and their neighboring objects to the outside.
- Set the initial properties of each of these objects. If the attribute values, tagged values, state, or role of any object changes in significant ways over the duration of the interaction, place a duplicate object on the diagram, update it with these new values, and connect them by a message

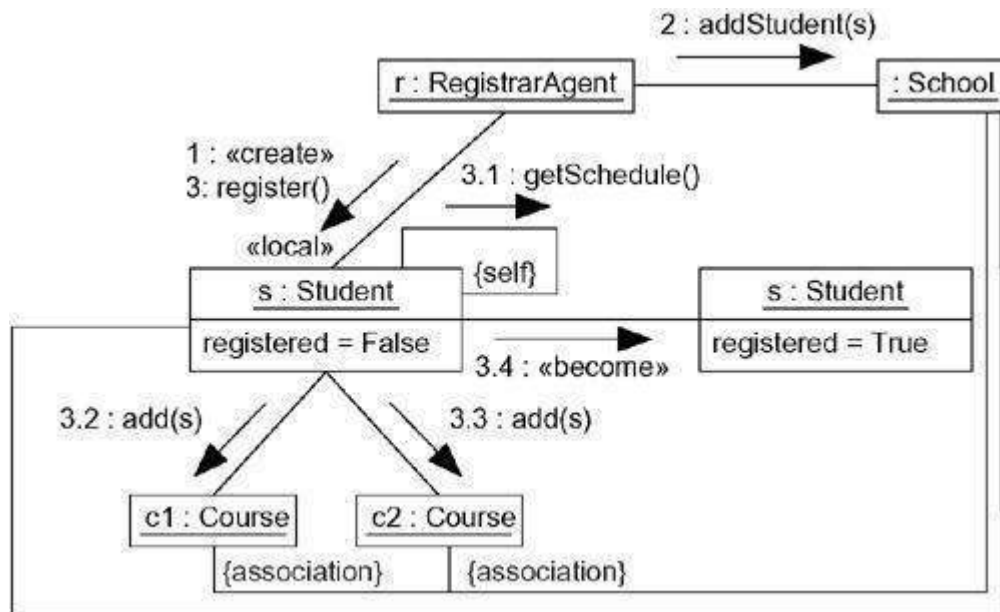
stereotyped as **become** or **copy** (with a suitable sequence number).

- Specify the links among these objects, along which messages may pass.
 1. Lay out the association links first; these are the most important ones, because they represent structural connections.
 2. Lay out other links next, and adorn them with suitable path stereotypes (such as **global** and **local**) to explicitly specify how these objects are related to one another.
- Starting with the message that initiates this interaction, attach each subsequent message to the appropriate link, setting its sequence number, as appropriate. Show nesting by using Dewey decimal numbering.
- If you need to specify time or space constraints, adorn each message with a timing mark and attach suitable time or space constraints.
- If you need to specify this flow of control more formally, attach pre- and postconditions to each message.

As with sequence diagrams, a single collaboration diagram can show only one flow of control (although you can show simple variations by using the UML's notation for interaction and branching). Typically, you'll have a number of such interaction diagrams, some of which are primary and others that show alternative paths or exceptional conditions. You can use packages to organize these collections of collaboration diagrams, giving each diagram a suitable name to distinguish it from its siblings.

For example, Figure shows a collaboration diagram that specifies the flow of control involved in registering a new student at a school, with an emphasis on the structural relationships among these objects. You see five objects: a **RegistrarAgent** (r), a **Student** (s), two **Course** objects (c1 and c2), and an unnamed **School** object. The flow of control is numbered explicitly. Action begins with the **RegistrarAgent** creating a **Student** object, adding the student to the school (the message **addStudent**), then telling the **Student** object to register itself. The **Student** object then invokes **getSchedule** on itself, presumably obtaining the **Course** objects for which it must register. The **Student** object then adds itself to each **Course** object. The flow ends with s rendered again, showing that it has an updated value for its **registered** attribute.

Figure Modeling Flows of Control by Organization



Note that this diagram shows a link between the **School** object and the two **Course** objects, plus another link between the **School** object and the **Student** object, although no messages are shown along these paths. These links help explain how the **Student** object can see the two **Course** objects to which it adds itself. **s**, **c1**, and **c2** are linked to the **School** via association, so **s** can find **c1** and **c2** during its call to **getSchedule**(which might return a collection of **Course** objects), indirectly through the **School** object.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for both sequence and collaboration diagrams, especially if the context of the diagram is an operation. For example, using the previous collaboration diagram, a reasonably clever forward engineering tool could generate the following Java code for the operation **register**, attached to the **Student** class.

```
public void register() { CourseCollection c =  
    getSchedule(); for (inti = 0; i<c.size(); i++)  
        c.item(i).add(this);  
    this.registered = true;  
}
```

"Reasonably clever" means the tool would have to realize that **getSchedule** returns a **CourseCollection** object, which it could determine by looking at the operation's signature. Bywalking across the contents of this object using a standard iteration idiom (which the tool could know about implicitly), the code could then generalize to any number of course offerings.

Reverse engineering (the creation of a model from code) is also possible for both sequence and collaboration diagrams, especially if the context of the code is the body of an operation. Segments of the previous diagram could have been produced by a tool from a prototypical execution of the **register** operation.

Use Cases

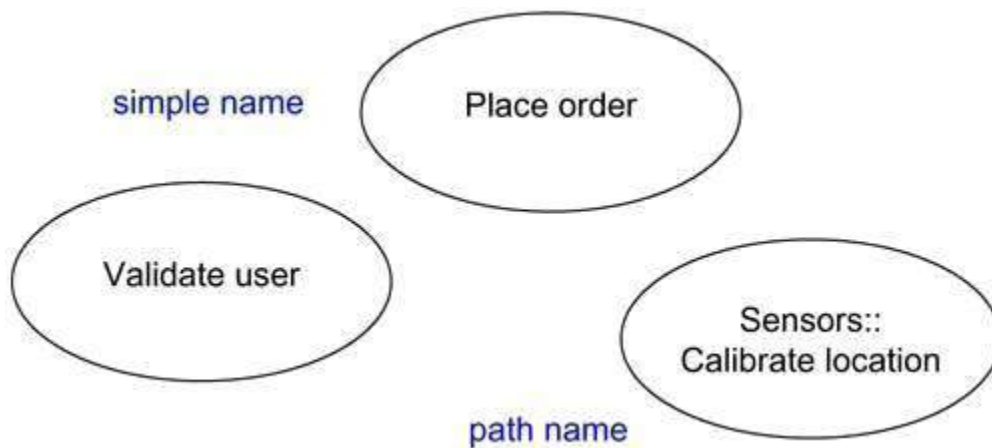
Terms and Concepts

A *use case* is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor. Graphically, a use case is rendered as an ellipse.

Names

Every use case must have a name that distinguishes it from other use cases. A *name* is a textual string. That name alone is known as a *simple name*; a *path name* is the use case name prefixed by the name of the package in which that use case lives. A use case is typically drawn showing only its name, as in [Figure](#).

Figure Simple and Path Names



Note

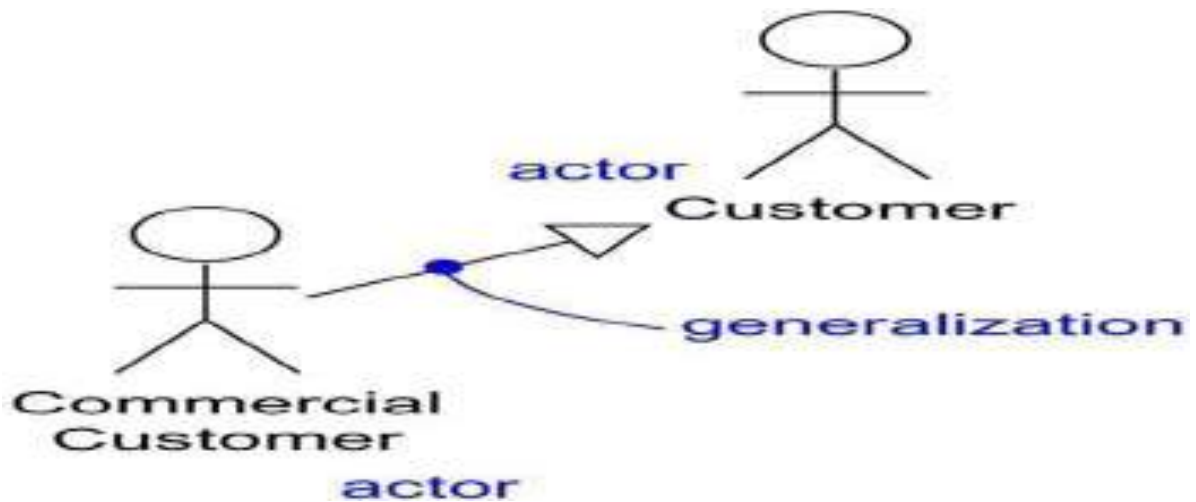
A use case name may be text consisting of any number of letters, numbers, and most punctuation marks (except for marks such as the colon, which is used to separate a class name and the name of its enclosing package) and may continue over several lines. In practice, use case names are short active verb phrases naming some behavior found in the vocabulary of the system you are modeling.

Use Cases and Actors

An actor represents a coherent set of roles that users of use cases play when interacting with these use cases. Typically, an actor represents a role that a human, a hardware device, or even another system plays with a system. For example, if you work for a bank, you might be a **LoanOfficer**. If you do your personal banking there, as well, you'll also play the role of **Customer**. An instance of an actor, therefore, represents an individual interacting with the system in a specific way. Although you'll use actors in your models, actors are not actually part of the system. They live outside the system.

As [Figure](#) indicates, actors are rendered as stick figures. You can define general kinds of actors (such as **Customer**) and specialize them (such as **CommercialCustomer**) using generalization relationships.

Figure Actors



Actors may be connected to use cases only by association. An association between an actor and a use case indicates that the actor and the use case communicate with one another, each one possibly sending and receiving messages.

Use Cases and Flow of Events

A use case describes *what* a system (or a subsystem, class, or interface) does but it does not specify *how* it does it. When you model, it's important that you keep clear the separation of concerns between this outside and inside view.

You can specify the behavior of a use case by describing a flow of events in text clearly enough for an outsider to understand it easily. When you write this flow of events, you should include how and when the use case starts and ends, when the use case interacts with the actors and what objects are exchanged, and the basic flow and alternative flows of the behavior.

For example, in the context of an ATM system, you might describe the use case **ValidateUser** in the following way:

Main flow of events:

The use case starts when the system prompts the *Customer* for a PIN number. The *Customer* can now enter a PIN number via the keypad. The *Customer* commits the entry by pressing the Enter button. The system then checks this PIN number to see if it is valid. If the PIN number is valid, the system acknowledges the entry, thus ending the use case.

Exceptional flow of events:

The *Customer* can cancel a transaction at any time by pressing the Cancel button, thus restarting the use case. No changes are made to the *Customer's* account.

Exceptional flow of events:

The *Customer* can clear a PIN number anytime before committing it and reenter a new PIN number.

Exceptional flow of events:

If the *Customer* enters an invalid PIN number, the use case restarts. If this happens three times in a row, the system cancels the entire transaction, preventing the *Customer* from interacting with the ATM for 60 seconds.

Use Cases and Scenarios

Typically, you'll first describe the flow of events for a use case in text. As you refine your understanding of your system's requirements, however, you'll want to also use interaction diagrams to specify these flows graphically. Typically, you'll use one sequence diagram to specify a use case's main flow, and variations of that diagram to specify a use case's exceptional flows.

It is desirable to separate main versus alternative flows because a use case describes a set of sequences, not just a single sequence, and it would be impossible to express all the details of an interesting use case in just one sequence. For example, in a human resources system, you might find the use case **Hire employee**. This general business function might have many possible variations. You might hire a person from another company (the most common scenario); you might transfer a person from one division to another (common in international companies); or you might hire a foreign national (which involves its own special rules). Each of these variants can be expressed in a different sequence.

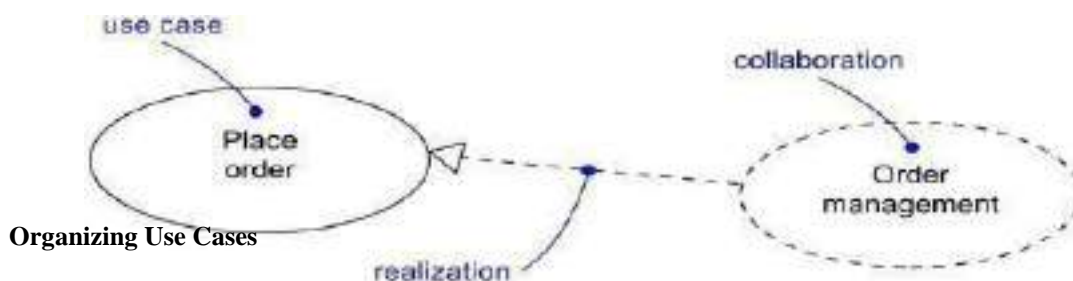
This one use case (**Hire employee**) actually describes a set of sequences in which each sequence in the set represents one possible flow through all these variations. Each sequence is called a scenario. A scenario is a specific sequence of actions that illustrates behavior. Scenarios are to use cases as instances are to classes, meaning that a scenario is basically one instance of a use case.

Use Cases and Collaborations

A use case captures the intended behavior of the system (or subsystem, class, or interface) you are developing, without having to specify how that behavior is implemented. That's an important separation because the analysis of a system (which specifies behavior) should, as much as possible, not be influenced by implementation issues (which specify how that behavior is to be carried out). Ultimately, however, you have to implement your use cases, and you do so by creating a society of classes and other elements that work together to implement the behavior of this use case. This society of elements, including both its static and dynamic structure, is modeled in the UML as a collaboration.

As Figure shows, you can explicitly specify the realization of a use case by a collaboration. Most of the time, though, a given use case is realized by exactly one collaboration, so you will not need to model this relationship explicitly.

Figure Use Cases and Collaborations

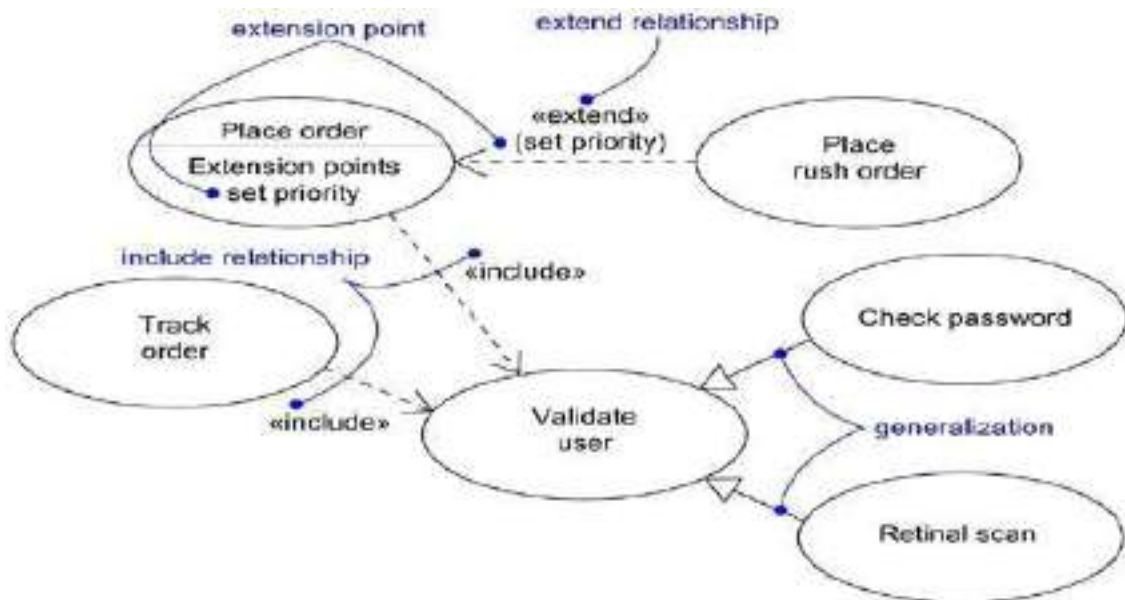


You can organize use cases by grouping them in packages in the same manner in which you can organize classes.

You can also organize use cases by specifying generalization, include, and extend relationships among them. You apply these relationships in order to factor common behavior (by pulling such behavior from other use cases that it includes) and in order to factor variants (by pushing such behavior into other use cases that extend it).

Generalization among use cases is just like generalization among classes. Here it means that the child use case inherits the behavior and meaning of the parent use case; the child may add to or override the behavior of its parent; and the child may be substituted any place the parent appears (both the parent and the child may have concrete instances). For example, in a banking system, you might have the use case **Validate User**, which is responsible for verifying the identify of the user. You might then have two specialized children of this use case (**Check password** and **Retinal scan**), both of which behave just like **Validate User** and may be applied anywhere **Validate User** appears, yet both of which add their own behavior (the former by checking atextual password, the latter by checking the unique retina patterns of the user). As shown in Figure , generalization among use cases is rendered as a solid directed line with a largeopen arrowhead, just like generalization among classes.

Figure Generalization, Include, and Extend



In include relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base. The included use case never stands alone, but is only instantiated as part of some larger base that includes it. You can think of include as the base use case pulling behavior from the supplier use case.

You use an include relationship to avoid describing the same flow of events several times, by putting the common behavior in a use case of its own (the use case that is included by a base use case). The include **relationship is essentially an example of delegation• you take a set of responsibilities of the system and capture it in one place (the included use case), then let all other parts of the system (other use cases) include the new aggregation of responsibilities whenever they need to use that functionality.**

You render an include relationship as a dependency, stereotyped as **include**. To specify the location in a flow of events in which the base use case includes the behavior of another, you simply write **include** followed by the name of the use case you want to include, as in the following flow for **Track order**.

Main flow of events:

Obtain and verify the order number. **include (Validate user)**. For each part in the order, query its status, then report back to the user.

An extend relationship between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case. The base use case may stand alone, but under certain conditions, its behavior may be extended by the behavior of another use case. This base use case may be extended only at certain points called, not surprisingly, its extension points. You can think of extend as the extension use case pushing behavior to the base use case. You use an extend relationship to model the part of a use case the user may see as optional system behavior. In this way, you separate optional behavior from mandatory behavior. You may also use an extend relationship to model a separate subflow that is executed only under given conditions. Finally, you may use an extend relationship to model several flows that may be inserted at a certain point, governed by explicit interaction with an actor.

You render an extend relationship as a dependency, stereotyped as **extend**. You may list the extension points of the base use case in an extra compartment. These extension points are just labels that may appear in the flow of the base use case. For example, the flow for **Place order** might read as follows:

Main flow of events:

include(Validate user). Collect the user's order items. **(set priority)**.
Submit the order for processing.

In this example, **set priority** is an extension point. A use case may have more than one extension point (which may appear more than once), and these are always matched by name. Under normal circumstances, this base use case will execute without regard for the priority of the order. If, on the other hand, this is an instance of a priority order, the flow for this base case will carry out as above. But at the extension point **(set priority)**, the behavior of the extending use case (**Place rush order**) will be performed, then the flow will resume. If there are multiple extension points, the extending use case will simply fold in its flows in order.

Other Features

Use cases are classifiers, so they may have attributes and operations that you may render just as for classes. You can think of these attributes as the objects inside the use case that you need to describe its outside behavior. Similarly, you can think of these operations as the actions of the system you need to describe a flow of events. These objects and operations may be used in your interaction diagrams to specify the behavior of the use case. As classifiers, you can also attach state machines to use cases. You can use state machines as yet another way to describe the behavior represented by a use case.

Common Modeling Techniques

Modeling the Behavior of an Element

The most common thing for which you'll apply use cases is to model the behavior of an element, whether it is the system as a whole, a subsystem, or a class. When you model the behavior of these things, it's important that you focus on what that element does, not how it does it.

Applying use cases to elements in this way is important for three reasons. First, by modeling the behavior

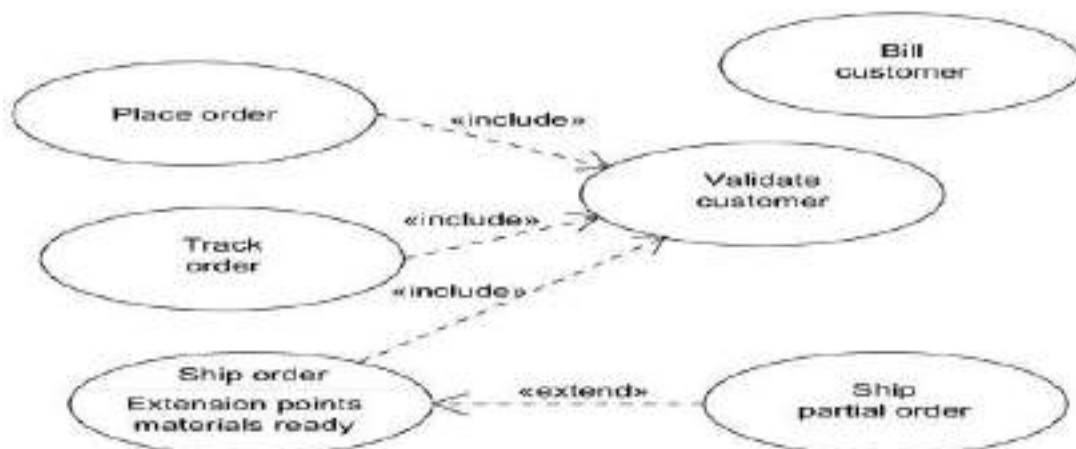
of an element with use cases, you provide a way for domain experts to specify its outside view to a degree sufficient for developers to construct its inside view. Use cases provide a forum for your domain experts, end users, and developers to communicate to one another. Second, use cases provide a way for developers to approach an element and understand it. A system, subsystem, or class may be complex and full of operations and other parts. By specifying an element's use cases, you help users of these elements to approach them in a direct way, according to how they are likely to use them. In the absence of such use cases, users have to discover on their own how to use those elements. Use cases let the author of an element communicate his or her intent about how that element should be used. Third, use cases serve as the basis for testing each element as it evolves during development. By continuously testing each element against its use cases, you continuously validate its implementation. Not only do these use cases provide a source of regression tests, but every time you throw a new use case at an element, you are forced to reconsider your implementation to ensure that this element is resilient to change. If it is not, you must fix your architecture appropriately.

To model the behavior of an element,

- Identify the actors that interact with the element. Candidate actors include groups that require certain behavior to perform their tasks or that are needed directly or indirectly to perform the element's functions.
- Organize actors by identifying general and more specialized roles.
- For each actor, consider the primary ways in which that actor interacts with the element. Consider also interactions that change the state of the element or its environment or that involve a response to some event.
- Consider also the exceptional ways in which each actor interacts with the element.
- Organize these behaviors as use cases, applying include and extend relationships to factor common behavior and distinguish exceptional behavior.

For example, a retail system will interact with customers who place and track orders. In turn, the system will ship orders and bill the customer. As Figure shows, you can model the behavior of such a system by declaring these behaviors as use cases (**Place order**, **Track order**, **Ship order**, and **Bill customer**). Common behavior can be factored out (**Validate customer**) and variants (**Ship partial order**) can be distinguished, as well. For each of these use cases, you would include a specification of the behavior, either by text, state machine, or interactions.

Figure Modeling the Behavior of an Element



Use Case Diagrams

Terms and Concepts

A *use case diagram* is a diagram that shows a set of use cases and actors and their relationships.

Common Properties

A use case diagram is just a special kind of diagram and shares the same common properties as do all **other diagrams**• **a name and graphical contents that are a** projection into a model. What distinguishes a use case diagram from all other kinds of diagrams is its particular content.

Contents

Use case diagrams commonly contain

- Use cases
- Actors
- Dependency, generalization, and association relationships

Like all other diagrams, use case diagrams may contain notes and constraints.

Use case diagrams may also contain packages, which are used to group elements of your model into larger chunks. Occasionally, you'll want to place instances of use cases in your diagrams, as well, especially when you want to visualize a specific executing system.

Common Uses

You apply use case diagrams to model the static use case view of a system. This view primarily supports **the behavior of a system**• **the outwardly visible services that the system** provides in the context of its environment.

When you model the static use case view of a system, you'll typically apply use case diagrams in one of two ways.

1. To model the context of a system

Modeling the context of a system involves drawing a line around the whole system and asserting which actors lie outside the system and interact with it. Here, you'll apply use case diagrams to specify the actors and the meaning of their roles.

2. To model the requirements of a system

Modeling the requirements of a system involves specifying what that system should do (from a point of view of outside the system), independent of how that system should do it. Here, you'll apply use case diagrams to specify the desired behavior of the system. In this manner, a use case diagram lets you view the whole system as a black box; you can see what's outside the system and you can see how that system reacts to the things outside, but you can't see how that system works on the inside.

Common Modeling Techniques

Modeling the Context of a System

Given a system• **any system**• **some things will live inside the system, some things will live outside it.** For example, in a credit card validation system, you'll find such things as accounts, transactions, and fraud detection agents inside the system. Similarly, you'll find such things as credit card customers and retail institutions outside the system. The things that live inside the system are responsible for carrying out the behavior that those on the outside expect the system to provide. All those things on the outside that

interact with the system constitute the system's context. This context defines the environment in which that system lives.

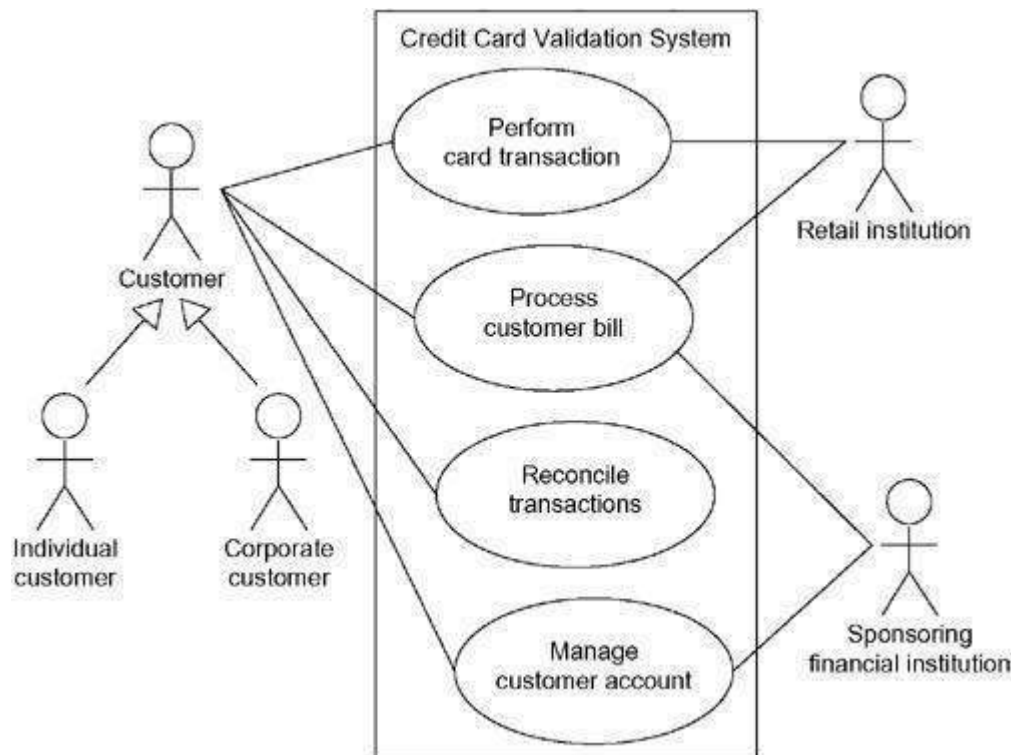
In the UML, you can model the context of a system with a use case diagram, emphasizing the actors that surround the system. Deciding what to include as an actor is important because in doing so you specify a class of things that interact with the system. Deciding what not to include as an actor is equally, if not more, important because that constrains the system's environment to include only those actors that are necessary in the life of the system.

To model the context of a system,

- Identify the actors that surround the system by considering which groups require help from the system to perform their tasks; which groups are needed to execute the system's functions; which groups interact with external hardware or other software systems; and which groups perform secondary functions for administration and maintenance.
- Organize actors that are similar to one another in a generalization/specialization hierarchy.
- Where it aids understandability, provide a stereotype for each such actor.
- Populate a use case diagram with these actors and specify the paths of communication from each actor to the system's use cases.

For example, Figure shows the context of a credit card validation system, with an emphasis on the actors that surround the system. You'll find **Customers**, of which there are two kinds (**Individual customer** and **Corporate customer**). These actors are the roles that humans play when interacting with the system. In this context, there are also actors that represent other institutions, such as **Retail institution** (with which a **Customer** performs a card transaction to buy an item or a service) and **Sponsoring financial institution** (which serves as the clearinghouse for the credit card account). In the real world, these latter two actors are likely software-intensive systems themselves.

Figure Modeling the Context of a System



This same technique applies to modeling the context of a subsystem. A system at one level of abstraction is often a subsystem of a larger system at a higher level of abstraction. Modeling the context of a subsystem is therefore useful when you are building systems of interconnected systems.

Modeling the Requirements of a System

A requirement is a design feature, property, or behavior of a system. When you state a system's requirements, you are asserting a contract, established between those things that lie outside the system and the system itself, which declares what you expect that system to do. For the most part, you don't care how the system does it, you just care *that* it does it. A well-behaved system will carry out all its requirements faithfully, predictably, and reliably. When you build a system, it's important to start with agreement about what that system should do, although you will certainly evolve your understanding of those requirements as you iteratively and incrementally implement the system. Similarly, when you are handed a system to use, knowing how it behaves is essential to using it properly.

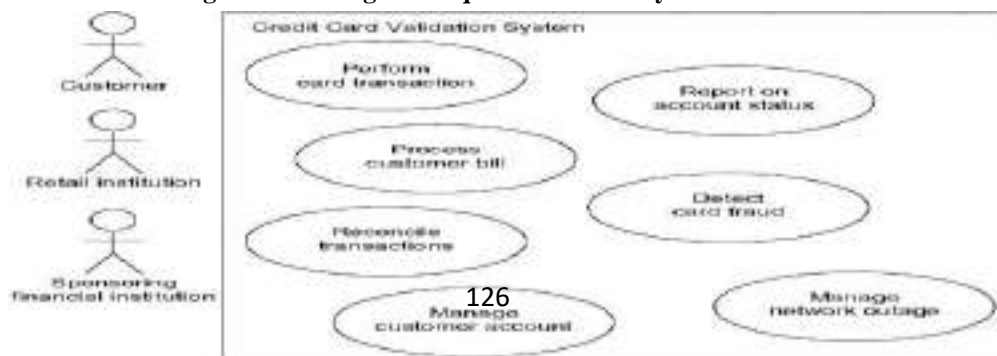
Requirements can be expressed in various forms, from unstructured text to expressions in a formal language, and everything in between. Most, if not all, of a system's functional requirements can be expressed as use cases, and the UML's use case diagrams are essential for managing these requirements.

To model the requirements of a system,

- Establish the context of the system by identifying the actors that surround it.
- For each actor, consider the behavior that each expects or requires the system to provide.
- Name these common behaviors as use cases.
- Factor common behavior into new use cases that are used by others; factor variant behavior into new use cases that extend more main line flows.
- Model these use cases, actors, and their relationships in a use case diagram.
- Adorn these use cases with notes that assert nonfunctional requirements; you may have to attach some of these to the whole system.

Figure expands on the previous use case diagram. Although it elides the relationships among the actors and the use cases, it adds additional use cases that are somewhat invisible to the average customer, yet are essential behaviors of the system. This diagram is valuable because it offers a common starting place for end users, domain experts, and developers to visualize, specify, construct, and document their decisions about the functional requirements of this system. For example, **Detect card fraud** is a behavior important to both the **Retail institution** and the **Sponsoring financial institution**. Similarly, **Report on account status** is another behavior required of the system by the various institutions in its context.

Figure Modeling the Requirements of a System



The requirement modeled by the use case **Manage network outages** is a bit different from all the others because it represents a secondary behavior of the system necessary for its reliable and continuous operation.

This same technique applies to modeling the requirements of a subsystem.

Forward and Reverse Engineering

Most of the UML's other diagrams, including class, component, and statechart diagrams, are clear candidates for forward and reverse engineering because each has an analog in the executable system. Use case diagrams are a bit different in that they reflect rather than specify the implementation of a system, subsystem, or class. Use cases describe how an element behaves, not how that behavior is implemented, so it cannot be directly forward or reverse engineered.

Forward engineering is the process of transforming a model into code through a mapping to an implementation language. A use case diagram can be forward engineered to form tests for the element to which it applies. Each use case in a use case diagram specifies a flow of events (and variants of those flows), **and these flows specify how the element is expected to behave• that's something worthy of testing.**

A well-structured use case will even specify pre- and postconditions that can be used to define a test's initial state and its success criteria. For each use case in a use case diagram, you can create a test case that you can run every time you release a new version of that element, thereby confirming that it works as required before other elements rely on it.

To forward engineer a use case diagram,

- For each use case in the diagram, identify its flow of events and its exceptional flow of events.
- Depending on how deeply you choose to test, generate a test script for each flow, using the flow's preconditions as the test's initial state and its postconditions as its success criteria.
- As necessary, generate test scaffolding to represent each actor that interacts with the use case. Actors that push information to the element or are acted on by the element may either be simulated or substituted by its real-world equivalent.
- Use tools to run these tests each time you release the element to which the use case diagram applies.

Reverse engineering is the process of transforming code into a model through a mapping from a specific implementation language. Automatically reverse engineering a use case diagram is pretty much beyond the state of the art, simply because there is a loss of information when moving from a specification of how an element behaves to how it is implemented. However, you can study an existing system and discern its intended behavior by hand, which you can then put in the form of a use case diagram. Indeed, this is pretty much what you have to do anytime you are handed an undocumented body of software. The UML's use case diagrams simply give you a standard and expressive language in which to state what you discover.

To reverse engineer a use case diagram,

- Identify each actor that interacts with the system.
- For each actor, consider the manner in which that actor interacts with the system, changes the state of the system or its environment, or responds to some event.
- Trace the flow of events in the executable system relative to each actor. Start with primary flows and only later consider alternative paths.
- Cluster related flows by declaring a corresponding use case. Consider modeling variants using

extend relationships, and consider modeling common flows by applying include relationships.

- Render these actors and use cases in a use case diagram, and establish their relationships.

Activity Diagrams

Terms and Concepts

An *activity diagram* shows the flow from activity to activity. An is an ongoing nonatomic execution within a state machine. Activities ultimately result in some *action*, which is made up of executable atomic computations that result in a change in state of the system or the return of a value. Actions encompass calling another operation, sending a signal, creating or destroying an object, or some pure computation, such as evaluating an expression. Graphically, an activity diagram is a collection of vertices and arcs.

Common Properties

An activity diagram is just a special kind of diagram and shares the same common properties as do all **other diagrams**• **a name and graphical contents that are a projection into a model. What distinguishes an** interaction diagram from all other kinds of diagrams is its content.

Contents

Activity diagrams commonly contain

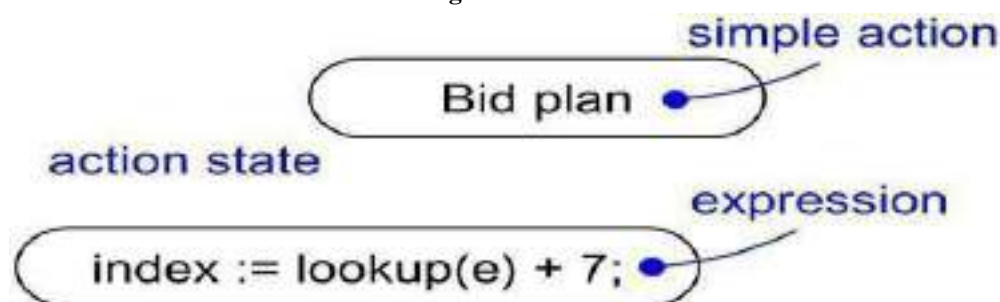
- Activity states and action states
- Transitions
- Objects

Like all other diagrams, activity diagrams may contain notes and constraints.

Action States and Activity States

In the flow of control modeled by an activity diagram, things happen. You might evaluate some expression that sets the value of an attribute or that returns some value. Alternately, you might call an operation on an object, send a signal to an object, or even create or destroy an object. These executable, atomic computations are called action states because they are states of the system, each representing the execution of an action. As [Figure](#) shows, you represent an action state using a lozenge shape (a symbol with horizontal top and bottom and convex sides). Inside that shape, you may write any expression.

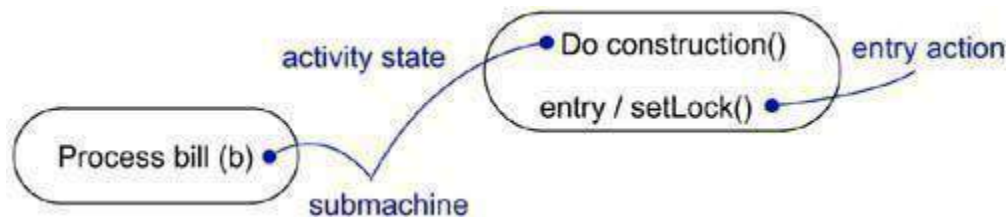
Figure Action States



Action states can't be decomposed. Furthermore, action states are atomic, meaning that events may occur, but the work of the action state is not interrupted. Finally, the work of an action state is generally considered to take insignificant execution time.

In contrast, activity states can be further decomposed, their activity being represented by other activity diagrams. Furthermore, activity states are not atomic, meaning that they may be interrupted and, in general, are considered to take some duration to complete. You can think of an action state as a special case of an activity state. An action state is an activity state that cannot be further decomposed. Similarly, you can think of an activity state as a composite, whose flow of control is made up of other activity states and action states. Zoom into the details of an activity state, and you'll find another activity diagram. As [Figure](#) shows, there's no notational distinction between action and activity states, except that an activity state may have additional parts, such as entry and exit actions (actions which are involved on entering and leaving the state, respectively) and submachine specifications.

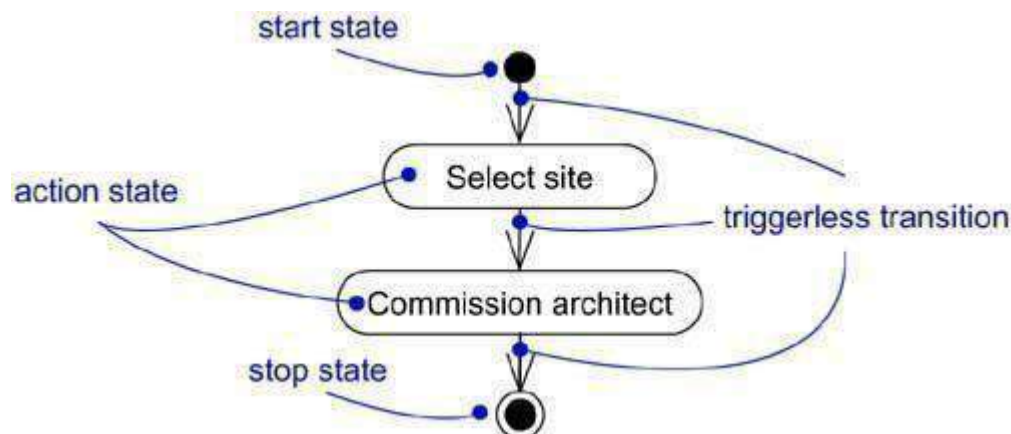
Figure Activity States



Transitions

When the action or activity of a state completes, flow of control passes immediately to the next action or activity state. You specify this flow by using transitions to show the path from one action or activity state to the next action or activity state. In the UML, you represent a transition as a simple directed line, as [Figure](#) shows.

Figure Triggerless Transitions

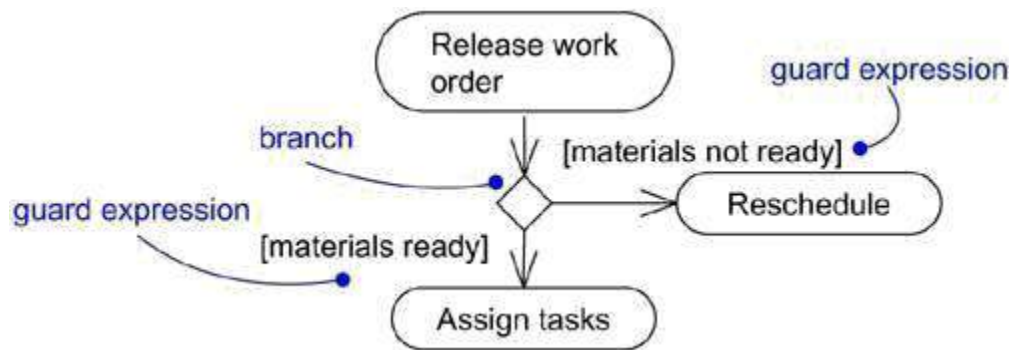


Indeed, a flow of control has to start and end someplace (unless, of course, it's an infinite flow, in which case it will have a beginning but no end). Therefore, as the figure shows, you may specify this initial state (a solid ball) and stop state (a solid ball inside a circle).

Branching

Simple, sequential transitions are common, but they aren't the only kind of path you'll need to model a flow of control. As in a flowchart, you can include a branch, which specifies alternate paths taken based on some Boolean expression. As Figure shows, you represent a branch as a diamond. A branch may have one incoming transition and two or more outgoing ones. On each outgoing transition, you place a Boolean expression, which is evaluated only once on entering the branch. Across all these outgoing transitions, guards should not overlap (otherwise, the flow of control would be ambiguous), but they should cover all possibilities (otherwise, the flow of control would freeze).

Figure Branching



As a convenience, you can use the keyword **else** to mark one outgoing transition, representing the path taken if no other guard expression evaluates to true. You can achieve the effect of iteration by using one action state that sets the value of an iterator, another action state that increments the iterator, and a branch that evaluates if the iteration is finished.

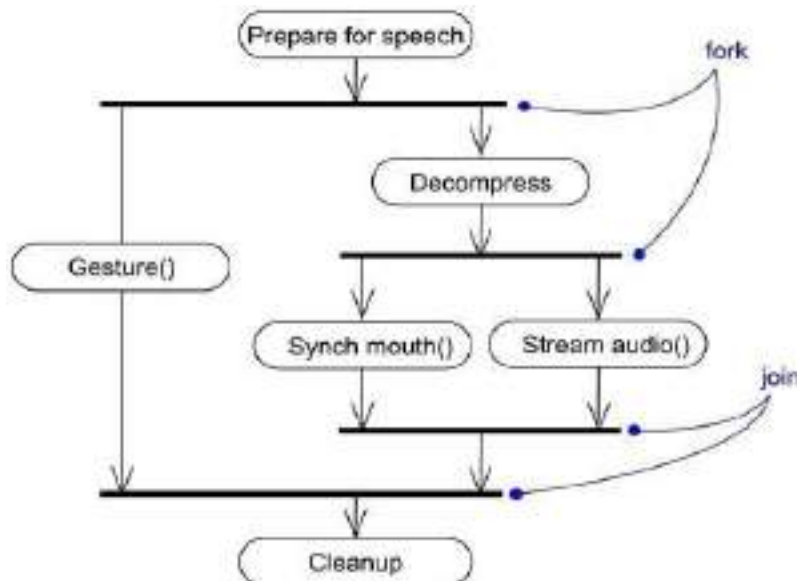
Forking and Joining

Simple and branching sequential transitions are the most common paths you'll find in activity diagrams.

However• especially when you are modeling workflows of business processes• you might encounter flows that are concurrent. In the UML, you use a synchronization bar to specify the forking and joining of these parallel flows of control. A synchronization bar is rendered as a thick horizontal or vertical line.

For example, consider the concurrent flows involved in controlling an audio-animatronic device that mimics human speech and gestures. As Figure shows, a fork represents the splitting of a single flow of control into two or more concurrent flows of control. A fork may have one incoming transition and two or more outgoing transitions, each of which represents an independent flow of control. Below the fork, the activities associated with each of these paths continues in parallel. Conceptually, the activities of each of these flows are truly concurrent, although, in a running system, these flows may be either truly concurrent (in the case of a system deployed across multiple nodes) or sequential yet interleaved (in the case of a system deployed across one node), thus giving only the illusion of true concurrency.

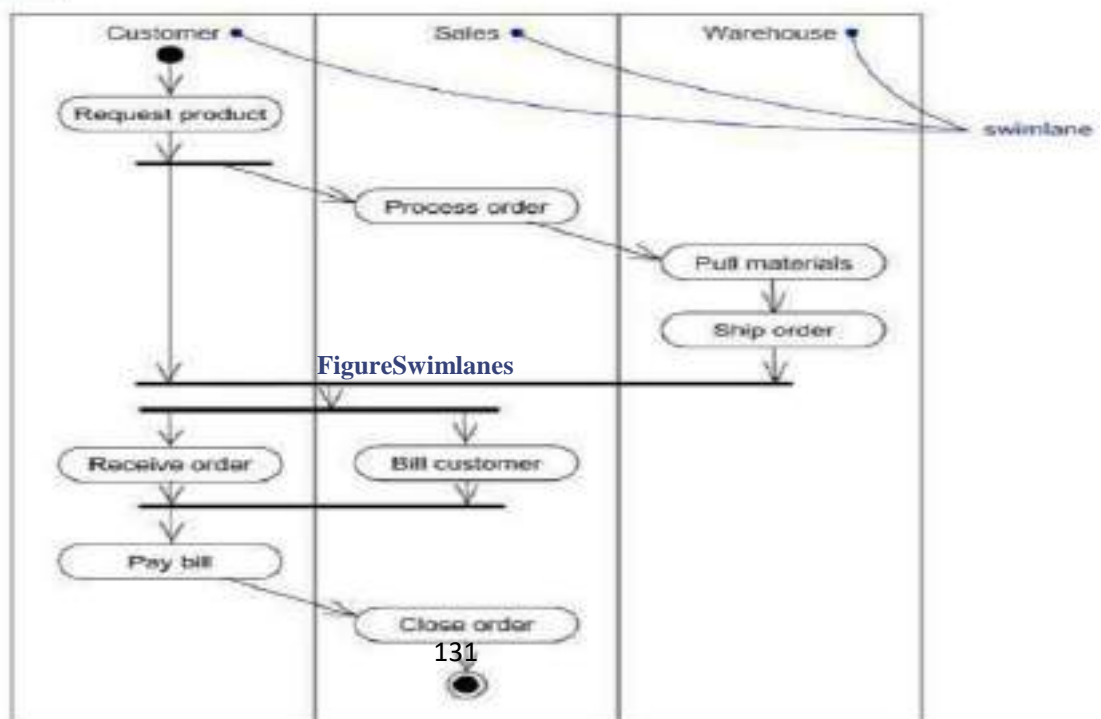
Figure Forking and Joining



As the figure also shows, a join represents the synchronization of two or more concurrent flows of control. A join may have two or more incoming transitions and one outgoing transition. Above the join, the activities associated with each of these paths continues in parallel. At the join, the concurrent flows synchronize, meaning that each waits until all incoming flows have reached the join, at which point one flow of control continues on below the join.

Swimlanes

You'll find it useful, especially when you are modeling workflows of business processes, to partition the activity states on an activity diagram into groups, each group representing the business organization responsible for those activities. In the UML, each group is called a swimlane because, visually, each group is divided from its neighbor by a vertical solid line, as shown in [Figure](#) . A swimlane specifies a locus of activities.



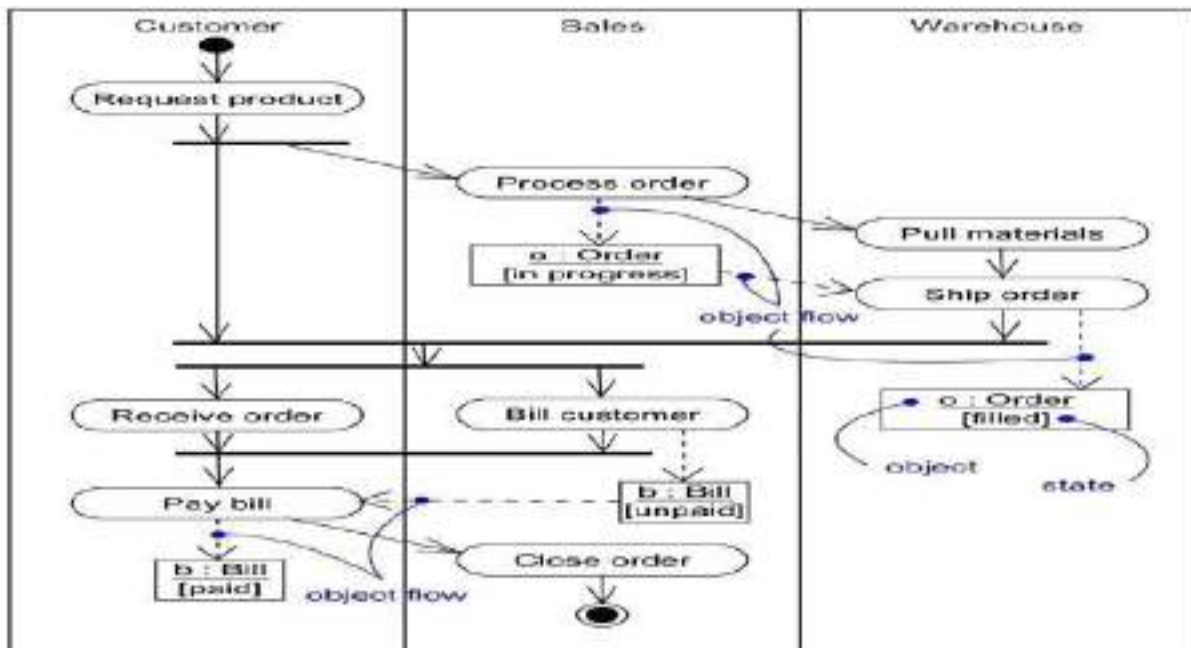
Each swimlane has a name unique within its diagram. A swimlane really has no deep semantics, except that it may represent some real-world entity. Each swimlane represents a high-level responsibility for part of the overall activity of an activity diagram, and each swimlane may eventually be implemented by one or more classes. In an activity diagram partitioned into swimlanes, every activity belongs to exactly one swimlane, but transitions may cross lanes.

Object Flow

Objects may be involved in the flow of control associated with an activity diagram. For example, in the workflow of processing an order as in the previous figure, the vocabulary of your problem space will also include such classes as **Order** and **Bill**. Instances of these two classes will be produced by certain activities (**Process order** will create an **Order** object, for example); other activities may modify these objects (for example, **Ship order** will change the state of the **Order** object to **filled**).

As Figure shows, you can specify the things that are involved in an activity diagram by placing these objects in the diagram, connected using a dependency to the activity or transition that creates, destroys, or modifies them. This use of dependency relationships and objects is called an object flow because it represents the participation of an object in a flow of control.

Figure Object Flow



In addition to showing the flow of an object through an activity diagram, you can also show how its role, state and attribute values change. As shown in the figure, you represent the state of an object by naming its state in brackets below the object's name. Similarly, you can represent the value of an object's attributes by rendering them in a compartment below the object's name.

Common Uses

You use activity diagrams to model the dynamic aspects of a system. These dynamic aspects may involve the activity of any kind of abstraction in any view of a system's architecture, including classes (which includes active classes), interfaces, components, and nodes.

When you use an activity diagram to model some dynamic aspect of a system, you can do so in the context of virtually any modeling element. Typically, however, you'll use activity diagrams in the context of the system as a whole, a subsystem, an operation, or a class. You can also attach activity diagrams to use cases (to model a scenario) and to collaborations (to model the dynamic aspects of a society of objects).

When you model the dynamic aspects of a system, you'll typically use activity diagrams in two ways.

1. To model a workflow

Here you'll focus on activities as viewed by the actors that collaborate with the system. Workflows often lie on the fringe of software-intensive systems and are used to visualize, specify, construct, and document business processes that involve the system you are developing. In this use of activity diagrams, modeling object flow is particularly important.

2. To model an operation

Here you'll use activity diagrams as flowcharts, to model the details of a computation. In this use of activity diagrams, the modeling of branch, fork, and join states is particularly important. The context of an activity diagram used in this way involves the parameters of the operation and its local objects.

Common Modeling Techniques

Modeling a Workflow

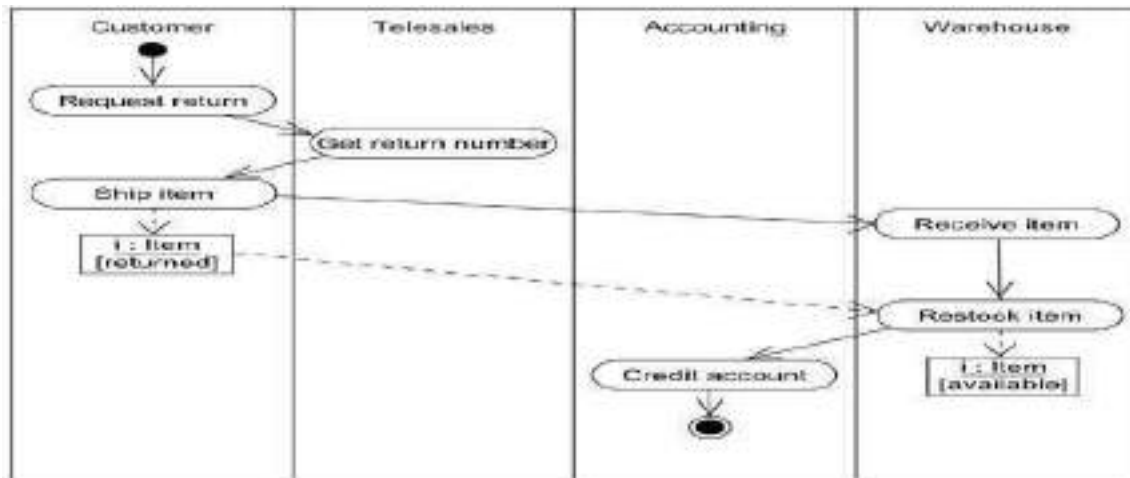
To model a workflow,

- Establish a focus for the workflow. For nontrivial systems, it's impossible to show all interesting workflows in one diagram.
- Select the business objects that have the high-level responsibilities for parts of the overall workflow. These may be real things from the vocabulary of the system, or they may be more abstract. In either case, create a swimlane for each important business object.
- Identify the preconditions of the workflow's initial state and the postconditions of the workflow's final state. This is important in helping you model the boundaries of the workflow.
- Beginning at the workflow's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- For complicated actions, or for sets of actions that appear multiple times, collapse these into activity states, and provide a separate activity diagram that expands on each.
- Render the transitions that connect these activity and action states. Start with the sequential flows in the workflow first, next consider branching, and only then consider forking and joining.
- If there are important objects that are involved in the workflow, render them in the activity diagram, as well. Show their changing values and state as necessary to communicate the intent of the object flow.

For example, Figure shows an activity diagram for a retail business, which specifies the workflow involved when a customer returns an item from a mail order. Work starts with the **Customer** action **Request return** and then flows through **Telesales** (**Get return number**), back to the **Customer** (**Ship item**), then to the **Warehouse** (**Receive item** then **Restock item**), finally ending in **Accounting** (**Credit account**). As the diagram indicates, one significant object (**i**, an instance of **Item**) also flows

the process, changing from the **returned** to the **available** state.

Figure Modeling a Workflow



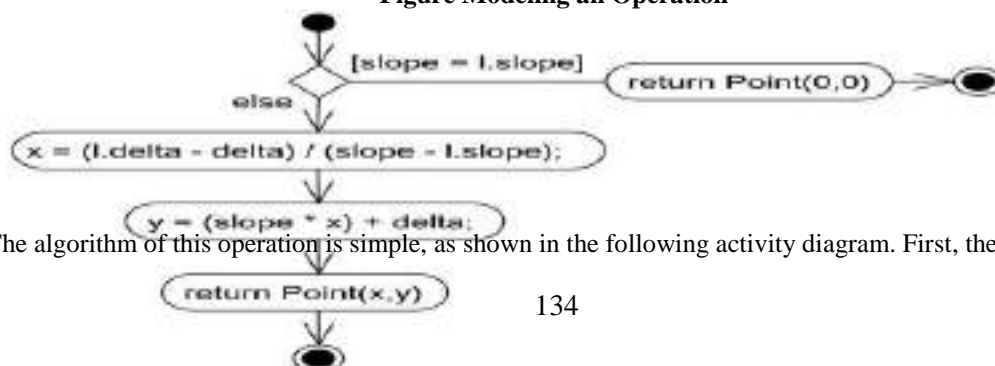
Modeling an Operation

To model an operation,

- Collect the abstractions that are involved in this operation. This includes the operation's parameters (including its return type, if any), the attributes of the enclosing class, and certain neighboring classes.
- Identify the preconditions at the operation's initial state and the postconditions at the operation's final state. Also identify any invariants of the enclosing class that must hold during the execution of the operation.
- Beginning at the operation's initial state, specify the activities and actions that take place over time and render them in the activity diagram as either activity states or action states.
- Use branching as necessary to specify conditional paths and iteration.
- Only if this operation is owned by an active class, use forking and joining as necessary to specify parallel flows of control.

For example, in the context of the class **Line**, Figure shows an activity diagram that specifies the algorithm of the operation **intersection**, whose signature includes one parameter (**l**, an **in** parameter of the class **Line**) and one return value (of the class **Point**). The class **Line** has two attributes of interest: **slope** (which holds the slope of the line) and **delta** (which holds the offset of the line relative to the origin).

Figure Modeling an Operation



The algorithm of this operation is simple, as shown in the following activity diagram. First, there's a guard

that tests whether the **slope** of the current line is the same as the **slope** of parameter **l**. If so, the lines do not intersect, and a **Point** at **(0,0)** is returned. Otherwise, the operation first calculates an **x** value for the point of intersection, then a **y** value; **x** and **y** are both objects local to the operation. Finally, a **Point** at **(x,y)** is returned.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for activity diagrams, especially if the context of the diagram is an operation. For example, using the previous activity diagram, a forward engineering tool could generate the following C++ code for the operation **intersection**.

```
Point Line::intersection (l : Line) {  
    if (slope == l.slope) return Point(0,0);  
    int x = (l.delta - delta) / (slope - l.slope); int y = (slope * x) + delta;  
    return Point(x, y);  
}
```

There's a bit of cleverness here, involving the declaration of the two local variables. A less-sophisticated tool might have first declared the two variables and then set their values.

Reverse engineering (the creation of a model from code) is also possible for activity diagrams, especially if the context of the code is the body of an operation. In particular, the previous diagram could have been generated from the implementation of the class **Line**.

More interesting than the reverse engineering of a model from code is the animation of a model against the execution of a deployed system. For example, given the previous diagram, a tool could animate the action states in the diagram as they were dispatched in a running system. Even better, with this tool also under the control of a debugger, you could control the speed of execution, possibly setting breakpoints to stop the action at interesting points in time to examine the attribute values of individual objects.

UNIT – 4

Advanced Behavioral Modeling

Syllabus :Events and signals,statemachines,processes and Threads ,time and space chart diagrams, Component, Deployment, Component Diagrams and Deployment diagrams

Events and Signals

Terms and Concepts

An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. A *signal* is a kind of event that represents the specification of an asynchronous stimulus communicated between instances.

Kinds of Events

Events may be external or internal. External events are those that pass between the system and its actors. For example, the pushing of a button and an interrupt from a collision sensor are both examples of external events. Internal events are those that pass among the objects that live inside the system. An overflow exception is an example of an internal event.

In the UML, you can model four kinds of events: signals, calls, the passing of time, and a change in state.

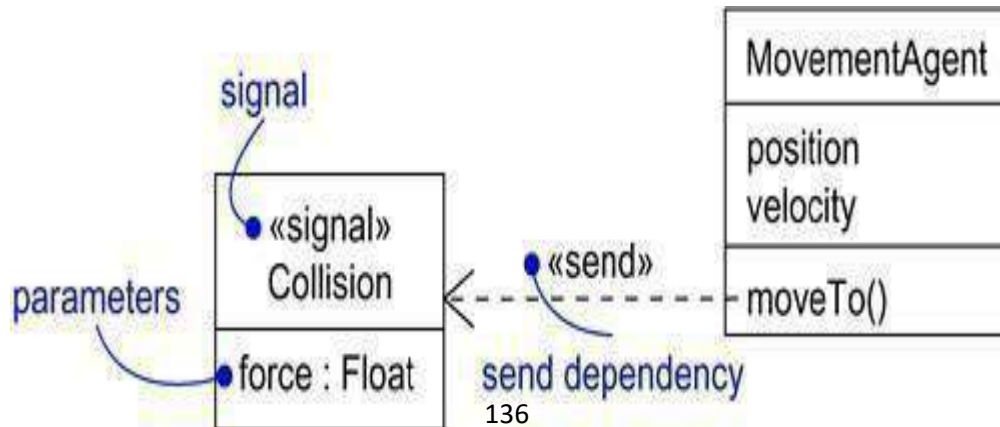
Signals

A signal represents a named object that is dispatched (thrown) asynchronously by one object and then received (caught) by another. Exceptions are supported by most contemporary programming languages and are the most common kind of internal signal that you will need to model.

Signals have a lot in common with plain classes. For example, signals may have instances, although you don't generally need to model them explicitly. Signals may also be involved in generalization relationships, permitting you to model hierarchies of events, some of which are general (for example, the signal **NetworkFailure**) and some of which are specific (for example, a specialization of **NetworkFailure** called **WarehouseServerFailure**). Also as for classes, signals may have attributes and operations.

A signal may be sent as the action of a state transition in a state machine or the sending of a message in an interaction. The execution of an operation can also send signals. In fact, when you model a class or an interface, an important part of specifying the behavior of that element is specifying the signals that its operations can send. In the UML, you model the relationship between an operation and the events that it can send by using a dependency relationship, stereotyped as **send**.

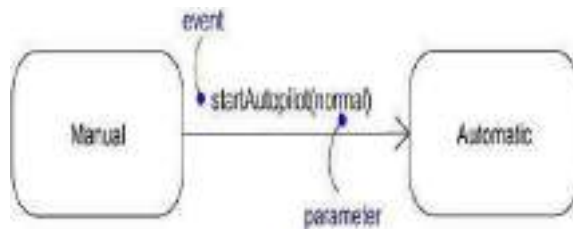
Figure Signals



Call Events

Just as a signal event represents the occurrence of a signal, a call event represents the dispatch of an operation. In both cases, the event may trigger a state transition in a state machine. Whereas a signal is an asynchronous event, a call event is, in general, synchronous. This means that when an object invokes an operation on another object that has a state machine, control passes from the sender to the receiver, the transition is triggered by the event, the operation is completed, the receiver transitions to a new state, and control returns to the sender.

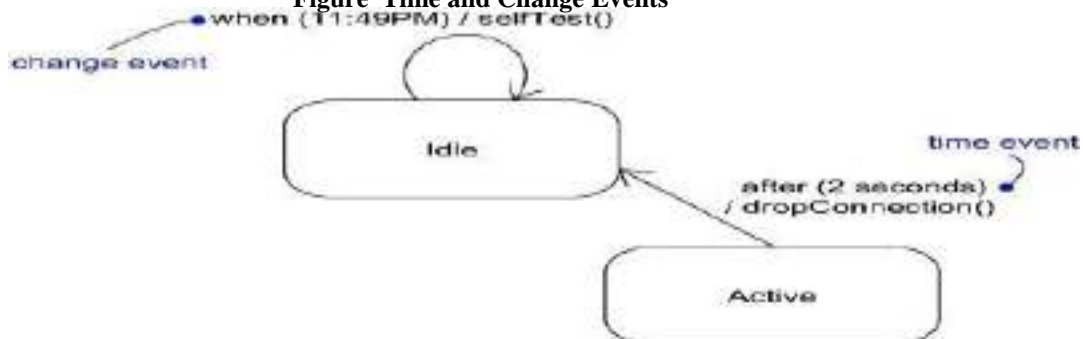
Figure Call Events



Time and Change Events

A time event is an event that represents the passage of time. As Figure shows, in the UML you model a time event by using the keyword **after** followed by some expression that evaluates to a period of time. Such expressions can be simple (for example, **after 2 seconds**) or complex (for example, **after 1 ms since exiting Idle**). Unless you specify it explicitly, the starting time of such an expression is the time since entering the current state.

Figure Time and Change Events



A change event is an event that represents a change in state or the satisfaction of some condition. As Figure shows, in the UML you model a change event by using the keyword **when** followed by some Boolean expression. You can use such expressions to mark an absolute time (such as **when time = 11:59**) or for the continuous test of an expression (for example, **when altitude < 1000**).

Sending and Receiving Events

Signal events and call events involve at least two objects: the object that sends the signal or invokes the operation, and the object to which the event is directed. Because signals are asynchronous, and because asynchronous calls are themselves signals, the semantics of events interact with the semantics of active

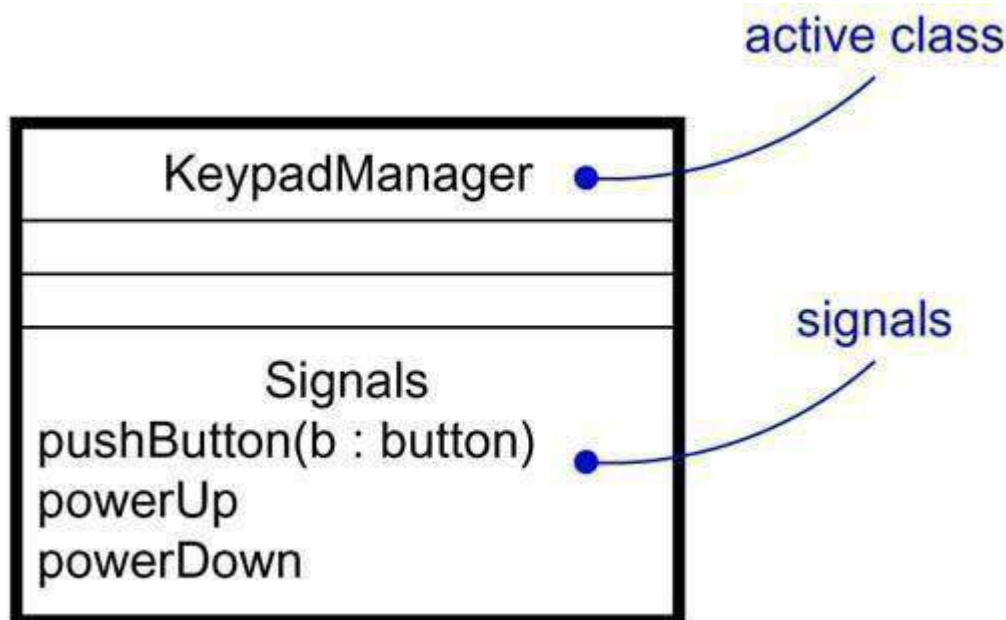
objects and passive objects.

Any instance of any class can send a signal to or invoke an operation of a receiving object. When an object sends a signal, the sender dispatches the signal and then continues along its flow of control, not waiting for any return from the receiver. For example, if an actor interacting with an ATM system sends the signal **pushButton**, the actor may continue along its way independent of the system to which the signal was sent. In contrast, when an object calls an operation, the sender dispatches the operation and then waits for the receiver. For example, in a trading system, an instance of the class **Trader** might invoke the operation **confirmTransaction** on some instance of the class **Trade**, thereby affecting the state of the **Trade** object. If this is a synchronous call, the **Trader** object will wait until the operation is finished.

Any instance of any class can receive a call event or a signal. If this is a synchronous call event, then the sender and the receiver are in a rendezvous for the duration of the operation. This means that the flow of control of the sender is put in lock step with the flow of control of the receiver until the activity of the operation is carried out. If this is a signal, then the sender and receiver do not rendezvous: the sender dispatches the signal but does not wait for a response from the receiver. In either case, this event may be lost (if no response to the event is specified), it may trigger the receiver's state machine (if there is one), or it may just invoke a normal method call.

In the UML, you model the call events that an object may receive as operations on the class of the object. In the UML, you model the named signals that an object may receive by naming them in an extra compartment of the class, as shown in [Figure](#) .

Figure Signals and Active Classes.



Common Modeling Techniques

Modeling a Family of Signals

In most event-driven systems, signal events are hierarchical. For example, an autonomous robot might distinguish between external signals, such as a **Collision**, and internal ones, such as a **HardwareFault**. External and internal signals need not be disjoint, however. Even within these two broad classifications, you might find specializations. For example, **HardwareFault** signals might be further specialized as

BatteryFault and **MovementFault**. Even these might be further specialized, such as **MotorStall**, a kind

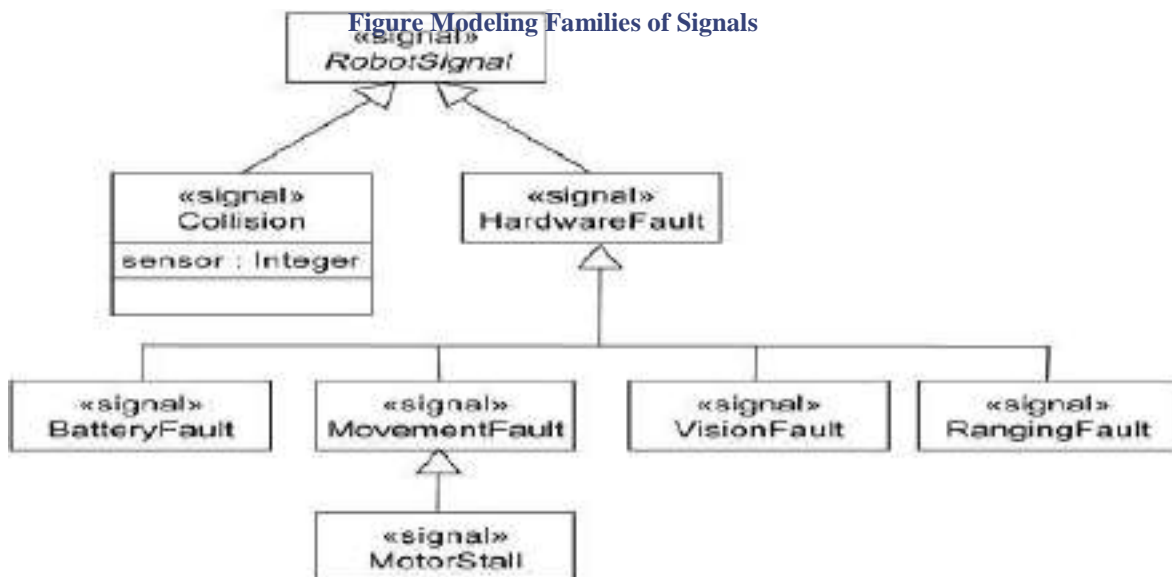
of **MovementFault**.

By modeling hierarchies of signals in this manner, you can specify polymorphic events. For example, consider a state machine with a transition triggered only by the receipt of a **MotorStall**. As a leaf signal in this hierarchy, the transition can be triggered only by that signal, so it is not polymorphic. In contrast, suppose you modeled the state machine with a transition triggered by the receipt of a **HardwareFault**. In this case, the transition is polymorphic and can be triggered by a **HardwareFault** or any of its specializations, including **BatteryFault**, **MovementFault**, and **MotorStall**.

To model a family of signals,

- Consider all the different kinds of signals to which a given set of active objects may respond.
- Look for the common kinds of signals and place them in a generalization/specialization hierarchy using inheritance. Elevate more general ones and lower more specialized ones.
- Look for the opportunity for polymorphism in the state machines of these active objects. Where you find polymorphism, adjust the hierarchy as necessary by introducing intermediate abstract signals.

Figure models a family of signals that may be handled by an autonomous robot. Note that the root signal (**RobotSignal**) is abstract, which means that there may be no direct instances. This signal has two immediate concrete specializations (**Collision** and **HardwareFault**), one of which (**HardwareFault**) is further specialized. Note that the **Collision** signal has one parameter.



Modeling Exceptions

An important part of visualizing, specifying, and documenting the behavior of a class or an interface is specifying the exceptions that its operations can raise. If you are handed a class or an interface, the operations you can invoke will be clear, but the exceptions that each operation may raise will not be clear unless you model them explicitly.

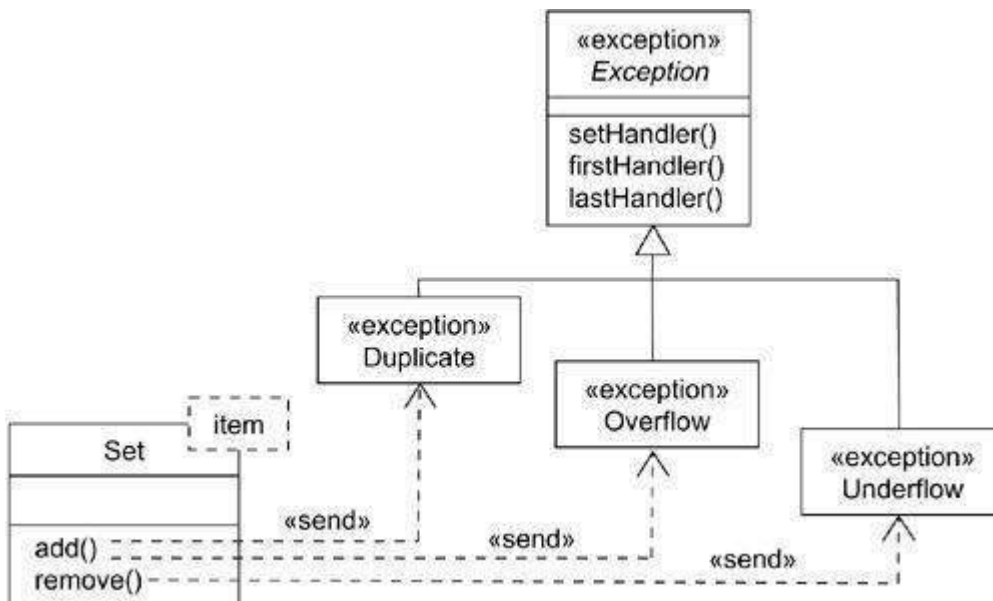
In the UML, exceptions are kinds of signals, which you model as stereotyped classes. Exceptions may be attached to specification operations. Modeling exceptions is somewhat the inverse of modeling a general family of signals. You model a family of signals primarily to specify the kinds of signals an active object may receive; you model exceptions primarily to specify the kinds of exceptions that an object may throw through its operations.

To model exceptions,

- For each class and interface, and for each operation of such elements, consider the exceptional conditions that may be raised.
- Arrange these exceptions in a hierarchy. Elevate general ones, lower specialized ones, and introduce intermediate exceptions, as necessary.
- For each operation, specify the exceptions that it may raise. You can do so explicitly (by showing **send** dependencies from an operation to its exceptions) or you can put this in the operation's specification.

Figure models a hierarchy of exceptions that may be raised by a standard library of container classes, such as the template class **Set**. This hierarchy is headed by the abstract signal **Exception** and includes three specialized exceptions: **Duplicate**, **Overflow**, and **Underflow**. As shown, the **add** operation raises **Duplicate** and **Overflow** exceptions, and the **remove** operation raises only the **Underflow** exception. Alternatively, you could have put these dependencies in the background by naming them in each operation's specification. Either way, by knowing which exceptions each operation may send, you can create clients that use the **Set** class correctly.

Figure Modeling Exceptions



State Machines

Terms and Concepts

A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events. A *state* is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. A *transition* is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. An *activity* is ongoing nonatomic execution within a state machine. An *action* is an executable atomic computation that results in a change in state of the model or the return of a value. Graphically, a state is rendered as a rectangle with rounded corners. A transition is rendered as a solid directed line.

Context

Every object has a lifetime. On creation, an object is born; on destruction, an object ceases to exist. In between, an object may act on other objects (by sending them messages), as well as be acted on (by being the target of a message). In many cases, these messages will be simple, synchronous operation calls. For example, an instance of the class **Customer** might invoke the operation **getAccountBalance** on an instance of the class **BankAccount**. Objects such as these don't need a state machine to specify their behavior because their current behavior does not depend on their past.

In other kinds of systems, you'll encounter objects that must respond to signals, which are asynchronous stimuli communicated between instances. For example, a cellular phone must respond to random phone calls (from other phones), keypad events (from the customer initiating a phone call), and to events from the network (when the phone moves from one call to another). Similarly, you'll encounter objects whose current behavior depends on their past behavior. For example, the behavior of an air-to-air missile guidance system will depend on its current state, such as **NotFlying** (it's not a good idea to launch a missile while it's attached to an aircraft that's still sitting on the ground) or **Searching** (you shouldn't arm the missile until you have a good idea what it's going to hit).

The behavior of objects that must respond to asynchronous stimulus or whose current behavior depends on their past is best specified by using a state machine. This encompasses instances of classes that can receive signals, including many active objects. In fact, an object that receives a signal but has no state machine will simply ignore that signal. You'll also use state machines to model the behavior of entire systems, especially reactive systems, which must respond to signals from actors outside the system.

States

A state is a condition or situation during the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An object remains in a state for a finite amount of time. For example, a **Heater** in a home might be in any of four states: **Idle** (waiting for a command to start heating the house), **Activating** (its gas is on, but it's waiting to come up to temperature), **Active** (its gas and blower are both on), and **ShuttingDown** (its gas is off but its blower is on, flushing residual heat from the system).

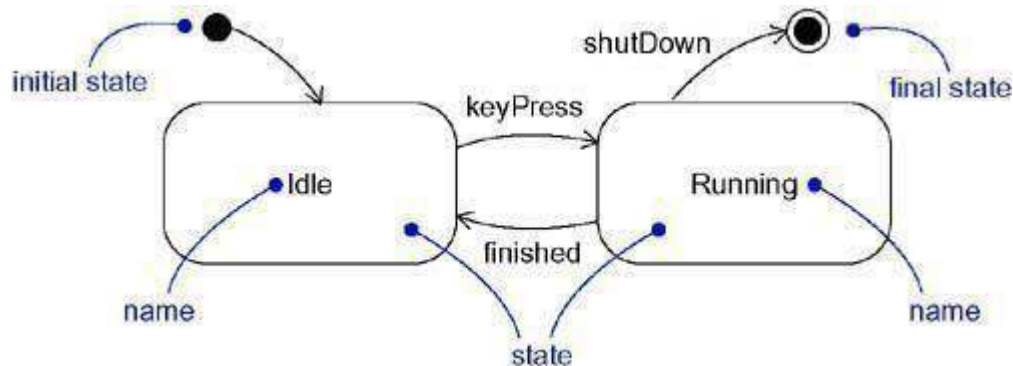
When an object's state machine is in a given state, the object is said to be in that state. For example, an instance of **Heater** might be **Idle** or perhaps **ShuttingDown**.

Astate has several parts.

1. Name	A textual string that distinguishes the state from other states; a state may be anonymous, meaning that it has no name
2. Entry/exit actions	Actions executed on entering and exiting the state, respectively
3. Internal transitions	Transitions that are handled without causing a change in state
4. Substates	The nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates
5. Deferred events	A list of events that are not handled in that state but, rather, are postponed and queued for handling by the object in another state

As [Figure](#) shows, you represent a state as a rectangle with rounded corners.

Figure States



Initial and Final States

As the figure shows, there are two special states that may be defined for an object's state machine. First, there's the initial state, which indicates the default starting place for the state machine or substate. An initial state is represented as a filled black circle. Second, there's the final state, which indicates that the execution of the state machine or the enclosing state has been completed. A final state is represented as a filled black circle surrounded by an unfilled circle.

Note

Initial and final states are really pseudostates. Neither may have the usual parts of a normal state, except for a name. A transition from an initial state to a final state may have the full complement of features, including a guard condition and action (but not a trigger event).

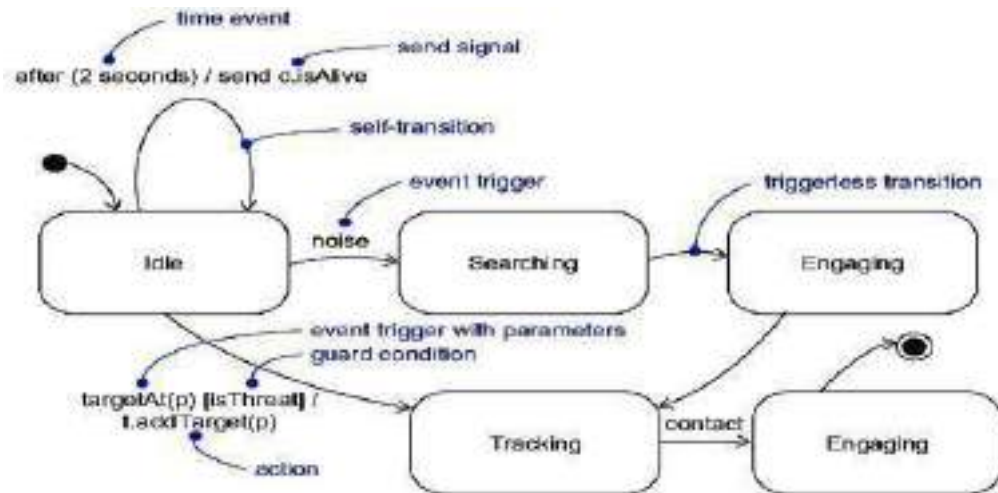
Transitions

A transition is a relationship between two states indicating that an object in the first state will perform

certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is said to fire. Until the transition fires, the object is said to be in the source state; after it fires, it is said to be in the target state. For example, a **Heater** might transition from the **Idle** to the **Activating** state when an event such as **tooCold** (with the parameter **desiredTemp**) occurs.

As Figure shows, a transition is rendered as a solid directed line from the source to the target state. A self-transition is a transition whose source and target states are the same.

Figure Transitions



Event Trigger

An event is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. As shown in the previous figure, events may include signals, calls, the passing of time, or a change in state. A signal or a call may have parameters whose values are available to the transition, including expressions for the guard condition and action.

It is also possible to have a triggerless transition, represented by a transition with no event trigger. A **triggerless transition**• also called a **completion transition**• is triggered implicitly when its source state has completed its activity.

Guard

As the previous figure shows, a guard condition is rendered as a Boolean expression enclosed in square brackets and placed after the trigger event. A guard condition is evaluated only after the trigger event for its transition occurs. Therefore, it's possible to have multiple transitions from the same source state and with the same event trigger, as long as those conditions don't overlap.

A guard condition is evaluated just once for each transition at the time the event occurs, but it may be evaluated again if the transition is retriggered. Within the Boolean expression, you can include conditions about the state of an object (for example, the expression **aHeater in Idle**, which evaluates True if the **Heater** object is currently in the **Idle** state).

Action

An action is an executable atomic computation. Actions may include operation calls (to the object that owns the state machine, as well as to other visible objects), the creation or destruction of another object, or the sending of a signal to an object. As the previous figure shows, there's a special notation for sending a signal• the signal name is prefixed with the keyword **send** as a visual cue.

Activities are discussed in a later section of this chapter; dependencies are discussed in An action is atomic, meaning that it cannot be interrupted by an event and therefore runs to completion. This is in contrast to an activity, which may be interrupted by other events.

Advanced States and Transitions

You can model a wide variety of behavior using only the basic features of states and transitions in the UML. Using these features, you'll end up with flat state machines, which means that your behavioral models will consist of nothing more than arcs (transitions) and vertices (states).

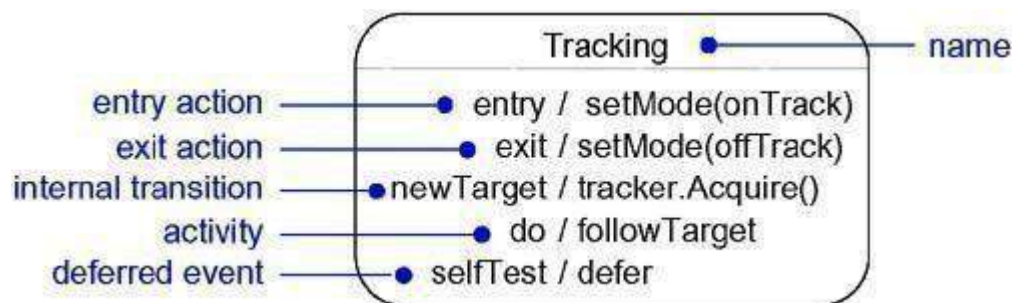
However, the UML's state machines have a number of advanced features that help you to manage complex behavioral models. These features often reduce the number of states and transitions you'll need, and they codify a number of common and somewhat complex idioms you'd otherwise encounter using flat state machines. Some of these advanced features include entry and exit actions, internal transitions, activities, and deferred events.

Entry and Exit Actions

In a number of modeling situations, you'll want to dispatch the same action whenever you enter a state, no matter which transition led you there. Similarly, when you leave a state, you'll want to dispatch the same action no matter which transition led you away. For example, in a missile guidance system, you might want to explicitly announce the system is **onTrack** whenever it's in the **Tracking** state, and **offTrack** whenever it's out of the state. Using flat state machines, you can achieve this effect by putting those actions on every entering and exiting transition, as appropriate. However, that's somewhat error prone; you have to remember to add these actions every time you add a new transition. Furthermore, modifying this action means that you have to touch every neighboring transition.

As Figure shows, the UML provides a shorthand for this idiom. In the symbol for the state, you can include an entry action (marked by the keyword event **entry**) and an exit action (marked by the keyword event **exit**), together with an appropriate action. Whenever you enter the state, its entry action is dispatched; whenever you leave the state, its exit action is dispatched.

Figure Advanced States and Transitions



Internal Transitions

Once inside a state, you'll encounter events you'll want to handle without leaving the state. These are called internal transitions, and they are subtly different from self-transitions. In a self-transition, such as you see in Figure., an event triggers the transition, you leave the state, an action (if any) is dispatched, and then you reenter the same state. Because this transition exits and then enters the state, a self-transition dispatches the state's exit action, then it dispatches the action of the self-transition, and finally, it

dispatches the state's entry action. However, suppose you want to handle the event but don't want to fire the state's entry and exit actions. Using flat state machines, you can achieve that effect, but you have to be diligent about remembering which of a state's transitions have these entry and exit actions and which do not.

As Figure shows, the UML provides a shorthand for this idiom, as well (for example, for the event **newTarget**). In the symbol for the state, you can include an internal transition (marked by an event). Whenever you are in the state and that event is triggered, the corresponding action is dispatched without leaving and then reentering the state. Therefore, the event is handled without dispatching the state's exit and then entry actions.

Activities

When an object is in a state, it generally sits idle, waiting for an event to occur. Sometimes, however, you may wish to model an ongoing activity. While in a state, the object does some work that will continue until it is interrupted by an event. For example, if an object is in the **Tracking** state, it might **followTarget** as long as it is in that state. As Figure shows, in the UML, you use the special **do** transition to specify the work that's to be done inside a state after the entry action is dispatched. The activity of a **do** transition might name another state machine (such as **followTarget**). You can also specify a sequence of actions—for example, **do / op1(a); op2(b);op3(c)**. Actions are never interruptible, but sequences of actions are. In between each action (separated by the semicolon), events may be handled by the enclosing state, which results in transitioning out of the state.

Deferred Events

Consider a state such as **Tracking**. As illustrated in Figure, suppose there's only one transition leading out of this state, triggered by the event **contact**. While in the state **Tracking**, any events other than **contact** and other than those handled by its substates will be lost. That means that the event may occur, but it will be postponed and no action will result because of the presence of that event. In every modeling situation, you'll want to recognize some events and ignore others. You include those you want to recognize as the event triggers of transitions; those you want to ignore you just leave out. However, in some modeling situations, you'll want to recognize some events but postpone a response to them until later. For example, while in the **Tracking** state, you may want to postpone a response to signals such as **selfTest**, perhaps sent by some maintenance agent in the system.

In the UML, you can specify this behavior by using deferred events. A deferred event is a list of events whose occurrence in the state is postponed until a state in which the listed events are not deferred becomes active, at which time they occur and may trigger transitions as if they had just occurred. As you can see in the previous figure, you can specify a deferred event by listing the event with the special action **defer**. In this example, **selfTest** events may happen while in the **Tracking** state, but they are held until the object is in the **Engaging** state, at which time it appears as if they just occurred.

Substates

These advanced features of states and transitions solve a number of common state machine modeling problems. However, there's one more feature of the UML's state machines—substates—that does even more to help you simplify the modeling of complex behaviors. A substate is a state that's nested inside another one. For example, a **Heater** might be in the **Heating** state, but also while in the **Heating** state, there might be a nested state called **Activating**. In this case, it's proper to say that the object is both **Heating** and **Activating**.

A simple state is a state that has no substructure. A state that has substates—that is, nested states—is called a composite state. A composite state may contain either concurrent (orthogonal) or sequential (disjoint) substates. In the UML, you render a composite state just as you do a simple state, but with an optional graphic compartment that shows a nested state machine. Substates may be nested to any level.

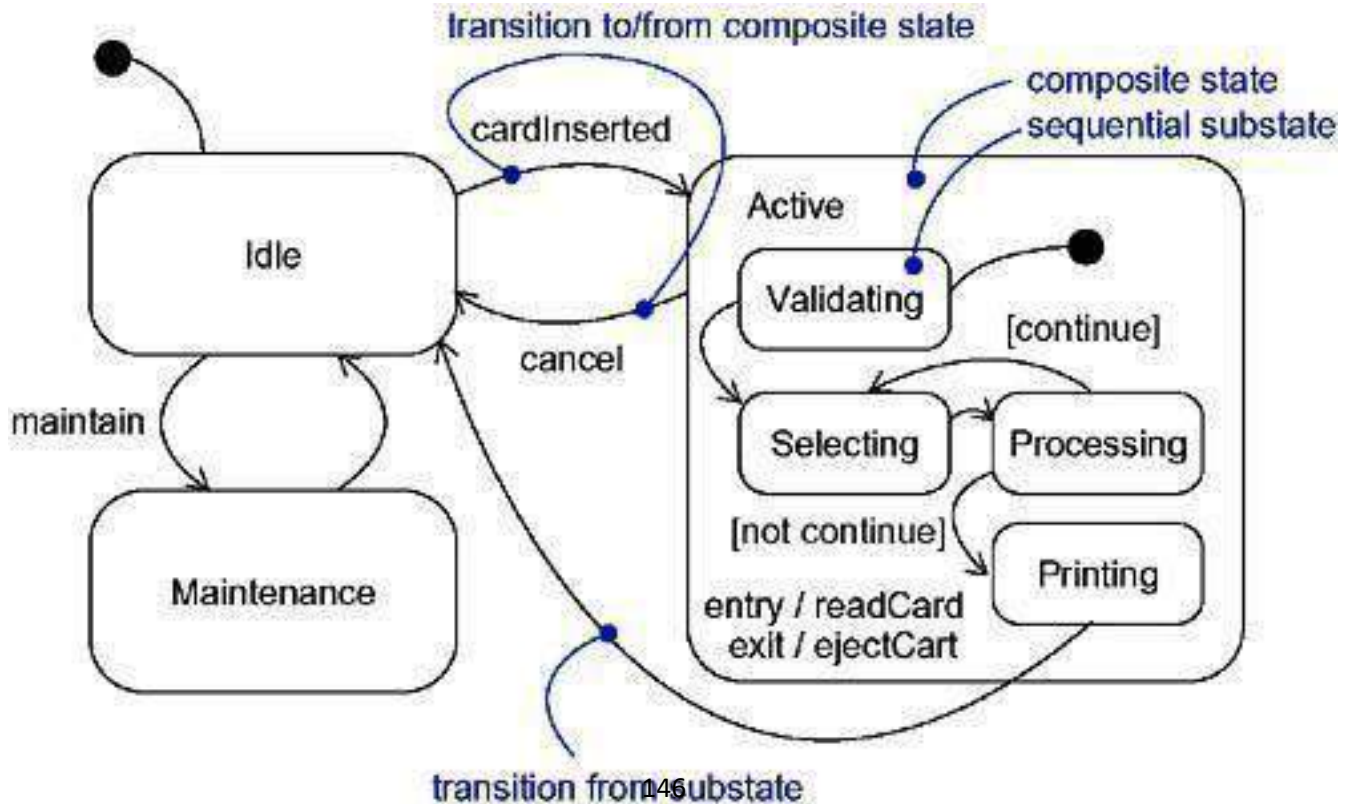
Sequential Substates

Consider the problem of modeling the behavior of an ATM. There are three basic states in which this system might be: **Idle** (waiting for customer interaction), **Active** (handling a customer's transaction), and **Maintenance** (perhaps having its cash store replenished). While **Active**, the behavior of the ATM follows a simple path: Validate the customer, select a transaction, process the transaction, and then print a receipt. After printing, the ATM returns to the **Idle** state. You might represent these stages of behavior as the states **Validating**, **Selecting**, **Processing**, and **Printing**. It would even be desirable to let the customer select and process multiple transactions after **Validating** the account and before **Printing** a final receipt.

The problem here is that, at any stage in this behavior, the customer might decide to cancel the transaction, returning the ATM to its **Idle** state. Using flat state machines, you can achieve that effect, but it's quite messy. Because the customer might cancel the transaction at any point, you'd have to include a suitable transition from every state in the **Active** sequence. That's messy because it's easy to forget to include these transitions in all the right places, and many such interrupting events means you end up with a multitude of transitions zeroing in on the same target state from various sources, but with the same event trigger, guard condition, and action.

Using sequential substates, there's a simpler way to model this problem, as Figure shows. Here, the **Active** state has a substructure, containing the substates **Validating**, **Selecting**, **Processing**, and **Printing**. The state of the ATM changes from **Idle** to **Active** when the customer enters a credit card in the machine. On entering the **Active** state, the entry action **readCard** is performed. Starting with the initial state of the substructure, control passes to the **Validating** state, then to the **Selecting** state, and then to the **Processing** state. After **Processing**, control may return to **Selecting** (if the customer has selected another transaction) or it may move on to **Printing**. After **Printing**, there's a triggerless transition back to the **Idle** state. Notice that the **Active** state has an exit action, which ejects the customer's credit card.

Figure SequentialSubstates



Notice also the transition from the **Active** state to the **Idle** state, triggered by the event **cancel**. In any substate of **Active**, the customer might cancel the transaction, and that returnsthe ATM to the **Idle** state (but only after ejecting the customer's credit card, which is the exit action dispatched on leaving the **Active** state, no matter what caused a transition out of that state). Without substates, you'd need a transition triggered by **cancel** on every substructure state.

Substates such as **Validating** and **Processing** are called sequential, or disjoint, substates. Given a set of disjoint substates in the context of an enclosing composite state, the object is said to be in the composite state and in only one of those substates (or the final state) at a time. Therefore, sequential substates partition the state space of the composite state into disjoint states.

From a source outside an enclosing composite state, a transition may target the composite state or it may target a substate. If its target is the composite state, the nested state machine must include an initial state, to which control passes after entering the composite state and after dispatching its entry action (if any). If its target is the nested state, control passes to the nested state, after dispatching the entry action (if any) of the composite state and then the entry action (if any) of the substate.

A transition leading out of a composite state may have as its source the composite state or a substate. In either case, control first leaves the nested state (and its exit action, if any, is dispatched), then it leaves the composite state (and its exit action, if any, is dispatched). A transition whose source is the composite state essentially cuts short (interrupts) the activity of the nested state machine.

History States

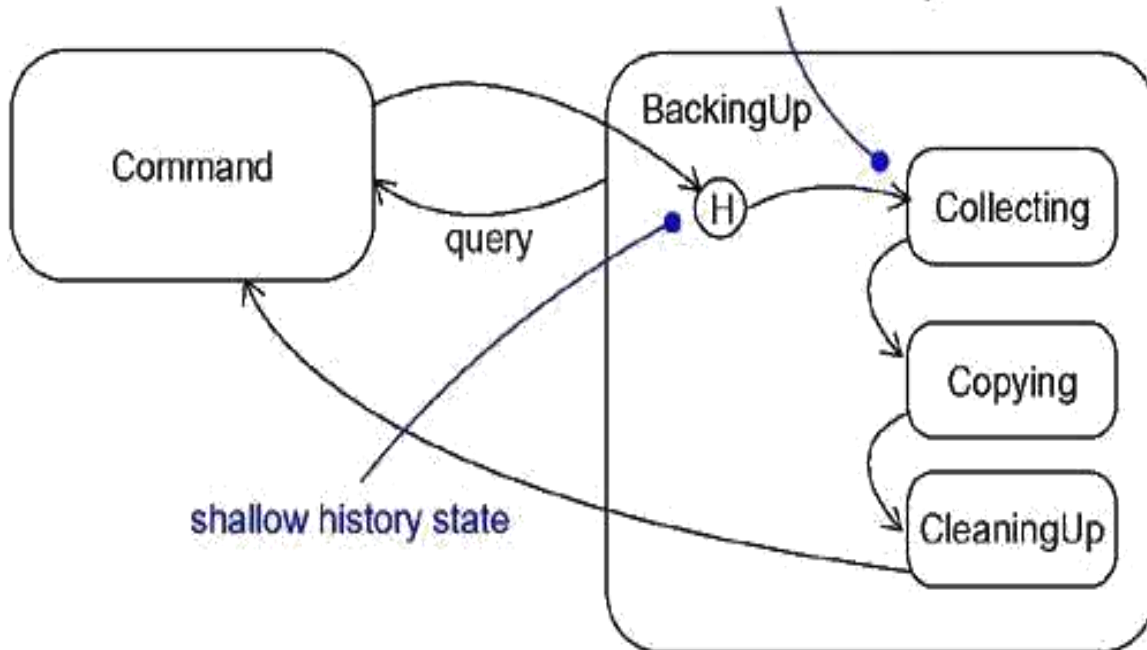
A state machine describes the dynamic aspects of an object whose current behavior depends on its past. A state machine in effect specifies the legal ordering of states an object may go through during its lifetime.

Unless otherwise specified, when a transition enters a composite state, the action of the nested state machine starts over again at its initial state (unless, of course, the transition targets a substate directly). However, there are times you'd like to model an object so that it remembers the last substate that was active prior to leaving the composite state. For example, in modeling the behavior of an agent that does an unattended backup of computers across a network, you'd like it to remember where it was in the process if it ever gets interrupted by, for example, a query from the operator.

Using flat state machines, you can model this, but it's messy. For each sequential substate, you'd need to have its exit action post a value to some variable local to the composite state. Then the initial state to this composite state would need a transition to every substate with a guard condition, querying the variable. In this way, leaving the composite state would cause the last substate to be remembered; entering the composite state would transition to the proper substate. That's messy because it requires you to remember to touch every substate and to set an appropriate exit action. It leaves you with a multitude of transitions fanning out from the same initial state to different target substates with very similar (but different) guard conditions.

In the UML, a simpler way to model this idiom is by using history states. A history state allows a composite state that contains sequential substates to remember the last substate that was active in it prior to the transition from the composite state. As [Figure](#) shows, you represent a shallow history state as a small circle containing the symbol **H**.

Figure History State
initial state for first entry



If you want a transition to activate the last substate, you show a transition from outside the composite state directly to the history state. The first time you enter a composite state, it has no history. This is the meaning of the single transition from the history state to a sequential substate such as **Collecting**. The target of this transition specifies the initial state of the nested statemachine the first time it is entered. Continuing, suppose that while in the **BackingUp** state and the **Copying** state, the **query** event is posted. Control leaves **Copying** and **BackingUp** (dispatching their exit actions as necessary) and returns to the **Command** state. When the action of **Command** completes, the triggerless transition returns to the history state of the composite state **BackingUp**. This time, because there is a history to the nested state machine, control passes back to the **Copying** state• thus bypassing the **Collecting** state• because **Copying** was the last substate active prior to the transition from **BackingUp**.

In either case, if a nested state machine reaches a final state, it loses its stored history and behaves as if it had not yet been entered for the first time.

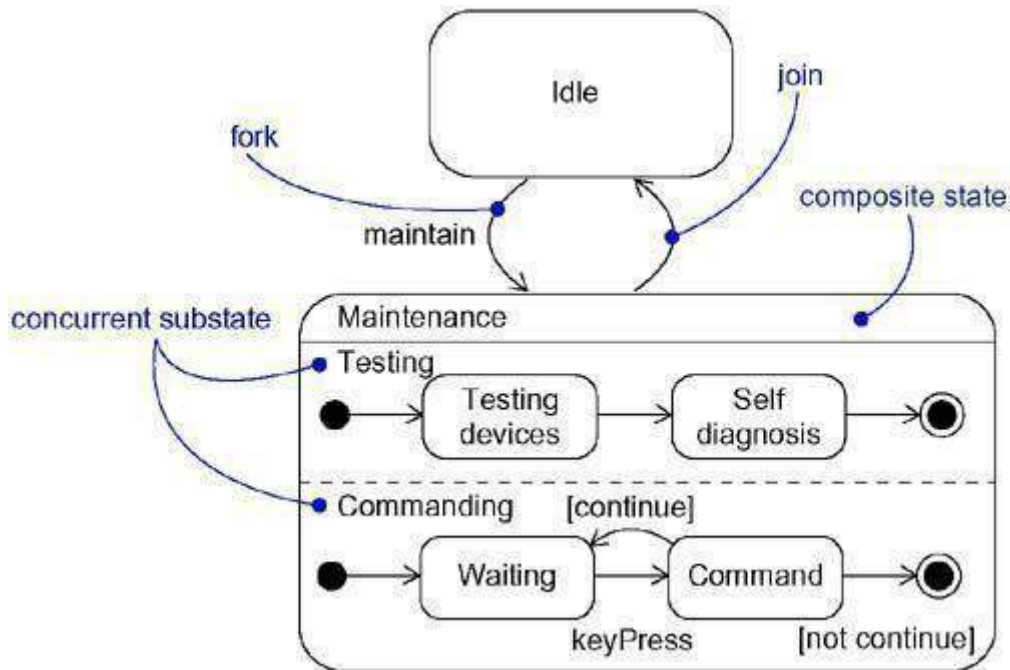
Concurrent Substates

Sequential substates are the most common kind of nested state machine you'll encounter. In certain modeling situations, however, you'll want to specify concurrent substates. These substates let you specify two or more state machines that execute in parallel in the context of the enclosing object.

For example, Figure shows an expansion of the **Maintenance** state from Figure. **Maintenance** is decomposed into two concurrent substates, **Testing** and **Commanding**, shown by nesting them in the **Maintenance** state but separating them from one another with a dashed line. Each of these concurrent

substates is further decomposed into sequential substates. When control passes from the **Idle** to the **Maintenance** state, control then forks to two concurrent flows• the enclosing object will be in the **Testing** state and the **Commanding** state. Furthermore, while in the **Commanding** state, the enclosing object will be in the **Waiting** or the **Command** state.

Figure Concurrent Substates



Execution of these two concurrent substates continues in parallel. Eventually, each nested state machine reaches its final state. If one concurrent substate reaches its final state before the other, control in that substate waits at its final state. When both nested state machines reach their final state, control from the two concurrent substates joins back into one flow.

Whenever there's a transition to a composite state decomposed into concurrent substates, control forks into as many concurrent flows as there are concurrent substates. Similarly, whenever there's a transition from a composite substate decomposed into concurrent substates, control joins back into one flow. This holds true in all cases. If all concurrent substates reach their final state, or if there is an explicit transition out of the enclosing composite state, control joins back into one flow.

Common Modeling Techniques

Modeling the Lifetime of an Object

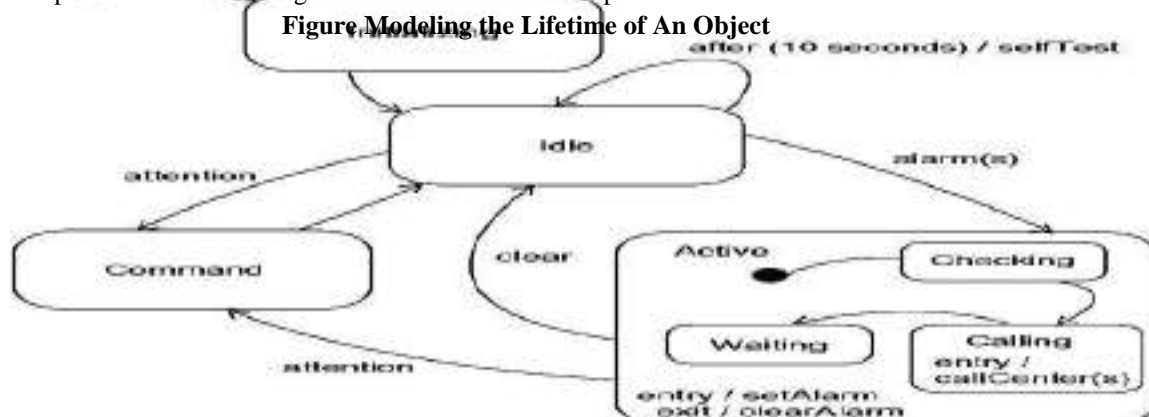
The most common purpose for which you'll use state machines is to model the lifetime of an object, especially instances of classes, use cases, and the system as a whole. Whereas interactions model the behavior of a society of objects working together, a state machine models the behavior of a single object over its lifetime, such as you'll find with user interfaces, controllers, and devices.

When you model the lifetime of an object, you essentially specify three things: the events to which the object can respond, the response to those events, and the impact of the past on current behavior. Modeling the lifetime of an object also involves deciding on the order in which the object can meaningfully respond to events, starting at the time of the object's creation and continuing until its destruction.

To model the lifetime of an object,

- Set the context for the state machine, whether it is a class, a use case, or the system as a whole.
 1. If the context is a class or a use case, collect the neighboring classes, including any parents of the class and any classes reachable by associations or dependences. These neighbors are candidate targets for actions and are candidates for including in guard conditions.
 2. if the context is the system as a whole, narrow your focus to one behavior of the system. Theoretically, every object in the system may be a participant in a model of the system's lifetime, and except for the most trivial systems, a complete model would be intractable.
- Establish the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.
- Decide on the events to which this object may respond. If already specified, you'll find these in the object's interfaces; if not already specified, you'll have to consider which objects may interact with the object in your context, and then which events they may possibly dispatch.
- Starting from the initial state to the final state, lay out the top-level states the object may be in. Connect these states with transitions triggered by the appropriate events. Continue by adding actions to these transitions.
- Identify any entry or exit actions (especially if you find that the idiom they cover is used in the state machine).
- Expand these states as necessary by using substates.
- Check that all events mentioned in the state machine match events expected by the interface of the object. Similarly, check that all events expected by the interface of the object are handled by the state machine. Finally, look to places where you explicitly want to ignore events.
- Check that all actions mentioned in the state machine are sustained by the relationships, methods, and operations of the enclosing object.
- Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses. Be especially diligent in looking for unreachable states and states in which the machine may get stuck.
- After rearranging your state machine, check it against expected sequences again to ensure that you have not changed the object's semantics.

For example, Figure shows the state machine for the controller in a home security system, which is responsible for monitoring various sensors around the perimeter of the house.



In the lifetime of this controller class, there are four main states: **Initializing** (the controller is starting up), **Idle** (the controller is ready and waiting for alarms or commands from the user), **Command** (the controller is processing commands from the user), and **Active** (the controller is processing an alarm condition). When the controller object is first created, it moves first to the **Initializing** state and then unconditionally to the **Idle** state. The details of these two states are not shown, other than the self-transition with the time event in the **Idle** state. This kind of time event is common in embedded systems, which often have a heartbeat timer that causes a periodic check of the system's health.

Control passes from the **Idle** state to the **Active** state on receipt of an **alarm** event (which includes the parameter **s**, identifying the sensor that was tripped). On entering the **Active** state, **setAlarm** is dispatched as the entry action, and control then passes first to the **Checking** state (validating the alarm), then to the **Calling** state (calling the alarm company to register the alarm), and finally to the **Waiting** state. The **Active** and **Waiting** states are exited only upon **clearing** the alarm, or by the user signaling the controller for **attention**, presumably to issue a command.

Notice that there is no final state. That, too, is common in embedded systems, which are intended to run continuously.

Processes and thread

Terms and Concepts

An *active object* is an object that owns a process or thread and can initiate control activity. An *active class* is a class whose instances are active objects. A *process* is a heavyweight flow that can execute concurrently with other processes. A *thread* is a lightweight flow that can execute concurrently with other threads within the same process. Graphically, an active class is rendered as a rectangle with thick lines. Processes and threads are rendered as stereotyped active classes (and also appear as sequences in interaction diagrams).

Flow of Control

In a purely sequential system, there is one flow of control. This means that one thing, and one thing only, can take place at a time. When a sequential program starts, control is rooted at the beginning of the program and operations are dispatched one after another. Even if there are concurrent things happening among the actors outside the system, a sequential program will process only one event at a time, queuing or discarding any concurrent external events. This is why it's called a flow of control. If you trace the execution of a sequential program, you'll see the locus of execution flow from one statement to another, in sequential order. You might see actions that branch, loop, and jump about, and if there is any recursion or iteration, you see the flow circle back on itself. Nonetheless, in a sequential system, there would be a single flow of execution.

In a concurrent system, there is more than one flow of control• that is, more than one thing can take place at a time. In a concurrent system, there are multiple simultaneous flows of control, each rooted at the head of an independent process or a thread. If you take a snapshot of a concurrent system while it's running, you'll logically see multiple loci of execution.

In the UML, you use an active class to represent a process or thread that is the root of an independent flow of control and that is concurrent with all peer flows of control.

Classes and Events

Active classes are just classes, albeit ones with a very special property. An active class represents an independent flow of control, whereas a plain class embodies no such flow. In contrast to active classes, plain classes are implicitly called passive because they cannot independently initiate control activity.

You use active classes to model common families of processes or threads. In technical terms, this means **that an active object• an instance of an active class• reifies (is a manifestation of) a process or thread**. By modeling concurrent systems with active objects, you give a name to each independent flow of control. When an active object is created, the associated flow of control is started; when the active object is destroyed, the associated flow of control is terminated.

Active classes share the same properties as all other classes. Active classes may have instances. Active classes may have attributes and operations. Active classes may participate in dependency, generalization, and association (including aggregation) relationships. Active classes may use any of the UML's extensibility mechanisms, including stereotypes, tagged values, and constraints. Active classes may be the realization of interfaces. Active classes may be realized by collaborations, and the behavior of an active class may be specified by using state machines.

In your diagrams, active objects may appear wherever passive objects appear. You can model the collaboration of active and passive objects by using interaction diagrams (including sequence and collaboration diagrams). An active object may appear as the target of an event in a state machine.

Speaking of state machines, both passive and active objects may send and receive signal events and call events.

Standard Elements

All of the UML's extensibility mechanisms apply to active classes. Most often, you'll use tagged values to extend active class properties, such as specifying the scheduling policy of the active class.

The UML defines two standard stereotypes that apply to active classes.

1. process	Specifies a heavyweight flow that can execute concurrently with other processes
2. thread	Specifies a lightweight flow that can execute concurrently with other threads within the same process

The distinction between a process and a thread arises from the two different ways a flow of control may be managed by the operating system of the node on which the object resides.

A process is heavyweight, which means that it is a thing known to the operating system itself and runs in an independent address space. Under most operating systems, such as Windows and Unix, each program runs as a process in its own address space. In general, all processes on a node are peers of one another, contending for all the same resources accessible on the node. Processes are never nested inside one another. If the node has multiple processors, then true concurrency on that node is possible. If the node has only one processor, there is only the illusion of true concurrency, carried out by the underlying operating system.

A thread is lightweight. It may be known to the operating system itself. More often, it is hidden inside a heavier-weight process and runs inside the address space of the enclosing process. In Java, for example, a thread is a child of the class **Thread**. All the threads that live in the context of a process are peers of one another, contending for the same resources accessible inside the process. Threads are never nested inside one another. In general, there is only the illusion of true concurrency among threads because it is processes, not threads, that are scheduled by a node's operating system.

Communication

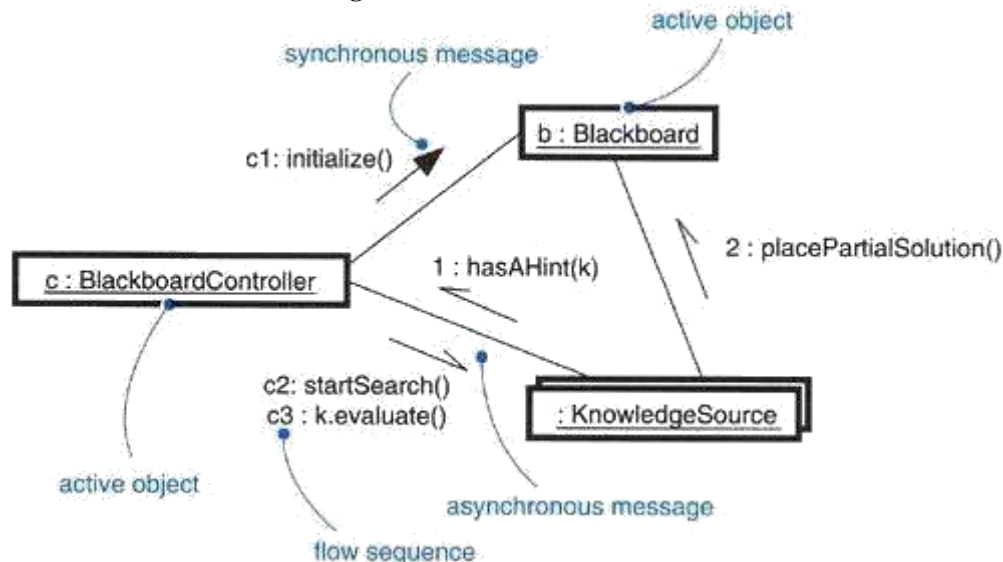
When objects collaborate with one another, they interact by passing messages from one to the other. In a system with both active and passive objects, there are four possible combinations of interaction that you must consider.

First, a message may be passed from one passive object to another. Assuming there is only one flow of control passing through these objects at a time, such an interaction is nothing more than the simple invocation of an operation.

Second, a message may be passed from one active object to another. When that happens, you have interprocess communication, and there are two possible styles of communication. First, one active object might synchronously call an operation of another. That kind of communication has rendezvous semantics, which means that the caller calls the operation; the caller waits for the receiver to accept the call; the operation is invoked; a return object (if any) is passed back to the caller; and then the two continue on their independent paths. For the duration of the call, the two flows of controls are in lock step. Second, one active object might asynchronously send a signal or call an operation of another object. That kind of communication has mailbox semantics, which means that the caller sends the signal or calls the operation and then continues on its independent way. In the meantime, the receiver accepts the signal or call whenever it is ready (with intervening events or calls queued) and continues on its way after it is done. This is called a mailbox because the two objects are not synchronized; rather, one object drops off a message for the other.

In the UML, you render a synchronous message as a full arrow and an asynchronous message as a half arrow, as in Figure .

Figure Communication



Third, a message may be passed from an active object to a passive object. A difficulty arises if more than one active object at a time passes their flow of control through one passive object. In that situation, you have to model the synchronization of these two flows very carefully, as discussed in the next section.

Fourth, a message may be passed from a passive object to an active one. At first glance, this may seem illegal, but if you remember that every flow of control is rooted in some active object, you'll understand

that a passive object passing a message to an active object has the same semantics as an active object passing a message to an active object.

Synchronization

Visualize for a moment the multiple flows of control that weave through a concurrent system. When a flow passes through an operation, we say that at a given moment, the locus of control is in the operation. If that operation is defined for some class, we can also say that at a given moment, the locus of control is in a specific instance of that class. You can have multiple flows of control in one operation (and therefore in one object), and you can have different flows of control in different operations (but still result in multiple flows of control in the one object).

The problem arises when more than one flow of control is in one object at the same time. If you are not careful, anything more than one flow will interfere with another, corrupting the state of the object. This is the classical problem of mutual exclusion. A failure to deal with it properly yields all sorts of race conditions and interference that cause concurrent systems to fail in mysterious and unrepeatable ways.

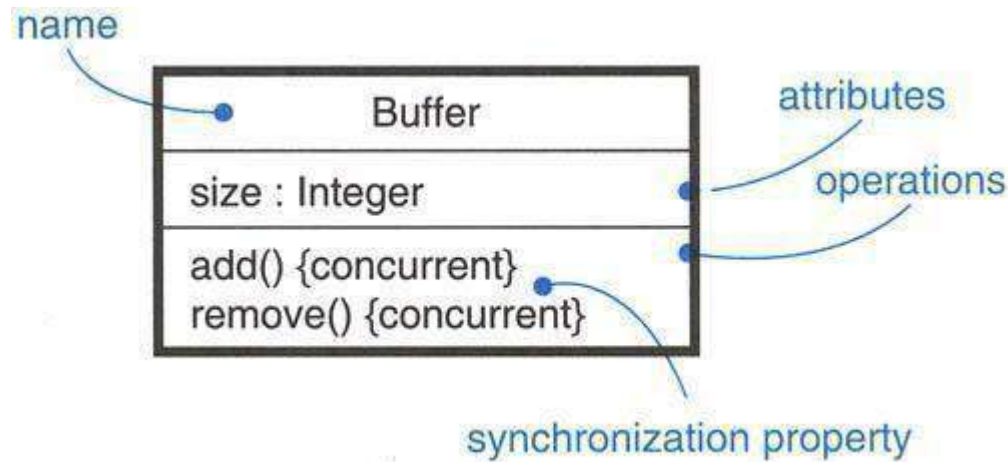
The key to solving this problem in object-oriented systems is by treating an object as a critical region. There are three alternatives to this approach, each of which involves attaching certain synchronization properties to the operations defined in a class. In the UML, you can model all three approaches.

1. Sequential	Callers must coordinate outside the object so that only one flow is in the object at a time. In the presence of multiple flows of control, the semantics and integrity of the object cannot be guaranteed.
2. Guarded	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by sequentializing all calls to all of the object's guarded operations. In effect, exactly one operation at a time can be invoked on the object, reducing this to sequential semantics.
3.	The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic.
Concurrent	

Some programming languages support these constructs directly. Java, for example, has the **synchronized** property, which is equivalent to the UML's **concurrent** property. In every language that supports concurrency, you can build support for all these properties by constructing them out of semaphores.

As Figure shows, you can attach these properties to an operation, which you can render in the UML by using constraint notation.

Figure Synchronization



Note

It is possible to model variations of these synchronization primitives by using constraints. For example, you might modify the **concurrent** property by allowing multiple simultaneous readers but only a single writer.

Process Views

Active objects play an important role in visualizing, specifying, constructing, and documenting a system's process view. The process view of a system encompasses the threads and processes that form the system's concurrency and synchronization mechanisms. This view primarily addresses the performance, scalability, and throughput of the system. With the UML, the static and dynamic aspects of this view are captured in the same kinds of diagrams as **for the design view—that is, class diagrams, interaction diagrams, activity diagrams, and statechart diagrams, but** with a focus on the active classes that represent these threads and processes.

Common Modeling Techniques

Modeling Multiple Flows of Control

Building a system that encompasses multiple flows of control is hard. Not only do you have to decide how best to divide work across concurrent active objects, but once you've done that, you also have to devise the right mechanisms for communication and synchronization among your system's active and passive objects to ensure that they behave properly in the presence of these multiple flows. For that reason, it helps to visualize the way these flows interact with one another. You can do that in the UML by applying class diagrams (to capture their static semantics) and interaction diagrams (to capture their dynamic semantics) containing active classes and objects.

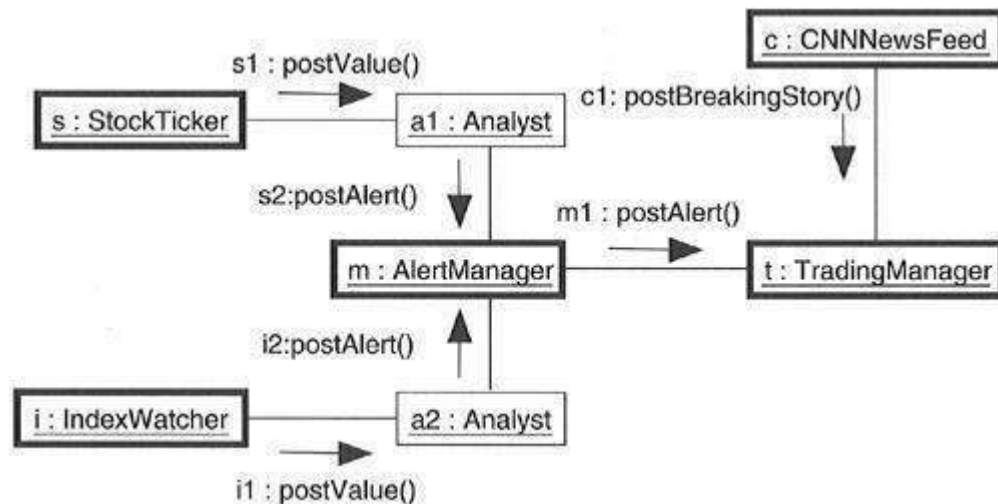
To model multiple flows of control,

- Identify the opportunities for concurrent action and reify each flow as an active class. Generalize common sets of active objects into an active class. Be careful not to over-engineer the process view of your system by introducing too much concurrency.

- Consider a balanced distribution of responsibilities among these active classes, then examine the other active and passive classes with which each collaborates statically. Ensure that each active class is both tightly cohesive and loosely coupled relative to these neighboring classes and that each has the right set of attributes, operations, and signals.
- Capture these static decisions in class diagrams, explicitly highlighting each active class.
- Consider how each group of classes collaborates with one another dynamically. Capture those decisions in interaction diagrams. Explicitly show active objects as the root of such flows. Identify each related sequence by identifying it with the name of the active object.
- Pay close attention to communication among active objects. Apply synchronous and asynchronous messaging, as appropriate.
- Pay close attention to synchronization among these active objects and the passive objects with which they collaborate. Apply sequential, guarded, or concurrent operation semantics, as appropriate.

For example, Figure shows part of the process view of a trading system. You'll find three objects that push information into the system concurrently: a **StockTicker**, an **IndexWatcher**, and a **CNNNewsFeed** (named **s**, **i**, and **c**, respectively). Two of these objects (**s** and **i**) communicate with their own **Analyst** instances (**a1** and **a2**). At least as far as this model goes, the **Analyst** can be designed under the simplifying assumption that only one flow of control will be active in its instances at a time. Both **Analyst** instances, however, communicate simultaneously with an **AlertManager** (named **m**). Therefore, **m** must be designed to preserve its semantics in the presence of multiple flows. Both **m** and **c** communicate simultaneously with **t**, a **TradingManager**. Each flow is given a sequence number that is distinguished by the flow of control that owns it.

Figure Modeling Flows of Control



Note

Interaction diagrams such as these are useful in helping you to visualize where two flows of control might cross paths and, therefore, where you must pay particular attention to the problems of communication and synchronization. Tools are permitted to offer even more distinct visual cues, such as by coloring each flow in a distinct way.

In diagrams such as this, it's also common to attach corresponding state machines, with orthogonal states showing the detailed behavior of each active object.

Modeling Interprocess Communication

As part of incorporating multiple flows of control in your system, you also have to consider the mechanisms by which objects that live in separate flows communicate with one another. Across threads (which live in the same address space), objects may communicate via signals or call events, the latter of which may exhibit either asynchronous or synchronous semantics. Across processes (which live in separate address spaces), you usually have to use different mechanisms.

The problem of interprocess communication is compounded by the fact that, in distributed systems, processes may live on separate nodes. Classically, there are two approaches to interprocess communication: message passing and remote procedure calls. In the UML, you still model these as asynchronous or synchronous events, respectively. But because these are no longer simple in-process calls, you need to adorn your designs with further information.

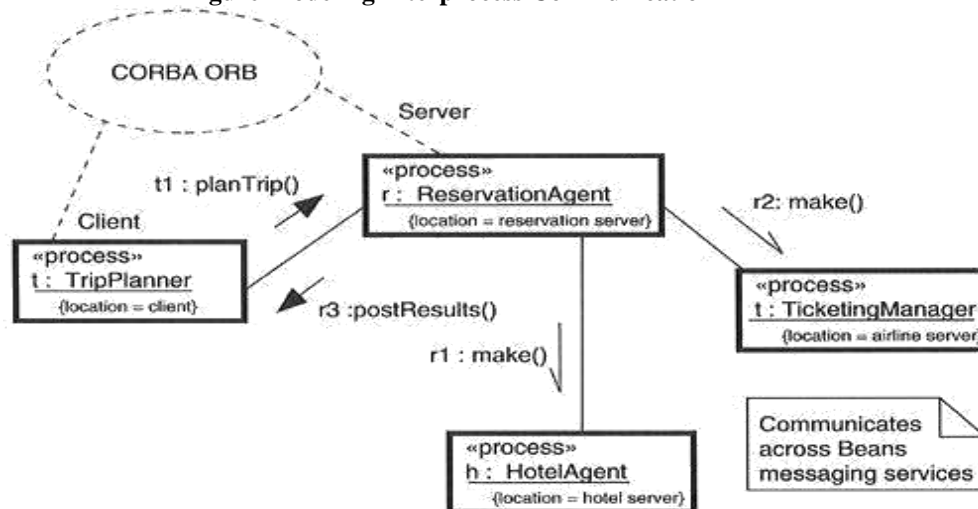
To model interprocess communication,

- Model the multiple flows of control.
- Consider which of these active objects represent processes and which represent threads. Distinguish them using the appropriate stereotype.
- Model messaging using asynchronous communication; model remote procedure calls using synchronous communication.
- Informally specify the underlying mechanism for communication by using notes, or more formally by using collaborations.

Figure shows a distributed reservation system with processes spread across four nodes. Each object is marked using the **process** stereotype. Each object is also marked with a **location** tagged value, specifying its physical location. Communication among the **ReservationAgent**, **TicketingManager**, and **HotelAgent** is asynchronous. Modeled with a note, communication is described as building on a Java Beans messaging service. Communication between the **TripPlanner** and the **ReservationSystem** is synchronous. The semantics of their interaction is found in the collaboration named **CORBA ORB**. The

TripPlanner acts as a **client**, and the **ReservationAgent** acts as a **server**. By zooming into the collaboration, you'll find the details of how this server and client collaborate.

Figure Modeling Interprocess Communication



Time and Space

Terms and Concepts

A *timing mark* is a denotation for the time at which an event occurs. Graphically, a timing mark is formed as an expression from the name given to the message (which is typically different from the name of the action dispatched by the message). A *time expression* is an expression that evaluates to an absolute or relative value of time. A *timing constraint* is a semantic statement about the relative or absolute value of time. Graphically, a timing constraint is **rendered as for any constraint• that is, a string enclosed by brackets** and generally connected to an element by a dependency relationship. *Location* is the placement of a component on a node. Graphically, location is rendered as a tagged **value•** that is, a string enclosed by brackets and placed below an element's name, or as the nesting of components inside nodes.

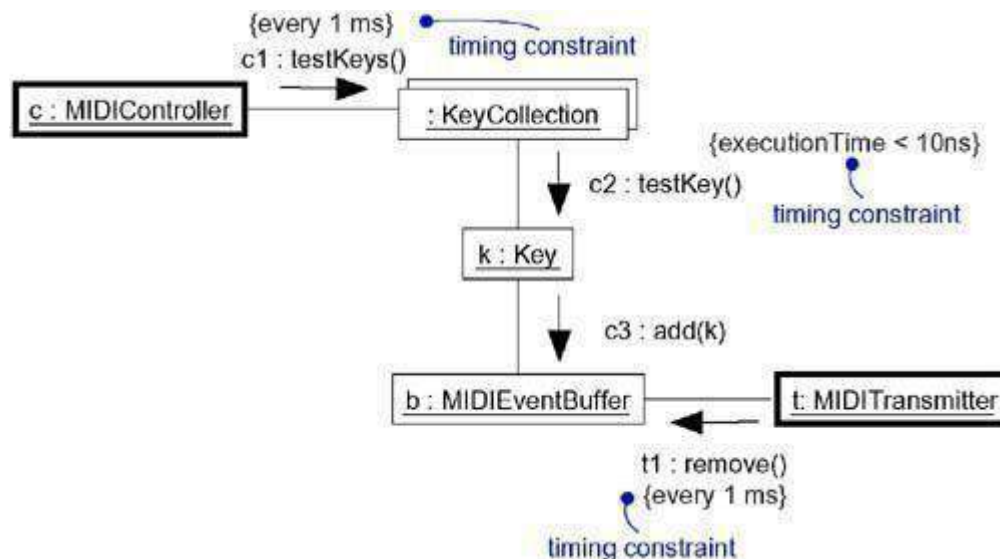
Time

Real time systems are, by their very name, time-critical systems. Events may happen at regular or irregular times; the response to an event must happen at predictable absolute times or at predictable times relative to the event itself.

The passing of messages represents the dynamic aspect of any system, so when you model the time-critical nature of a system with the UML, you can give a name to each message in an interaction to be used as a timing mark. Messages in an interaction are usually not given names. They are mainly rendered with the name of an event, such as a signal or a call. As a result, you can't use the event name to write an expression because the same event may trigger different messages. If the designated message is ambiguous, use the explicit name of the message in a timing mark to designate the message you want to mention in a time expression. A timing mark is nothing more than an expression

formed from the name of a message in an interaction. Given a message name, you can refer to any of three functions **of that message• that is, startTime,stopTime, andexecutionTime**. You can then use these functions to specify arbitrarily complex time expressions, perhaps even using weights or offsets that are either constants or variables (as long as those variables can be bound at execution time). Finally, as shown in Figure_, you can place these time expressions in a timing constraint to specify the timing behavior of the system. As constraints, you can render them by placing them adjacent to the appropriate message, or you can explicitly attach them using dependency relationships.

Figure Time

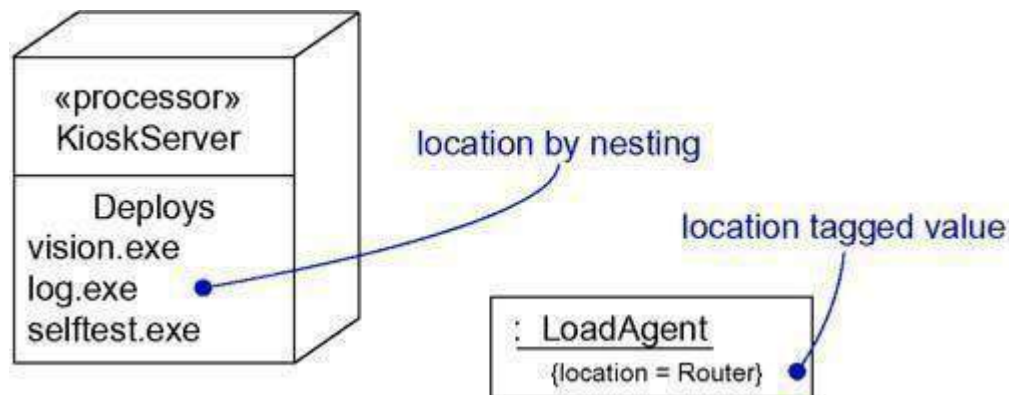


Location

Distributed systems, by their nature, encompass components that are physically scattered among the nodes of a system. For many systems, components are fixed in place at the time they are loaded on the system; in other systems, components may migrate from node to node.

In the UML, you model the deployment view of a system by using deployment diagrams that represent the topology of the processors and devices on which your system executes. Components such as executables, libraries, and tables reside on these nodes. Each instance of a node will own instances of certain components, and each instance of a component will be owned by exactly one instance of a node (although instances of the same kind of component may be spread across different nodes). For example, as Figure shows, the executable component **vision.exe** may reside on the node named **KioskServer**.

Figure Location



Instances of plain classes may reside on a node, as well. For example, as Figure shows, an instance of the class **LoadAgent** lives on the node named **Router**.

As the figure illustrates, you can model the location of an element in two ways in the UML. First, as shown for the **KioskServer**, you can physically nest the element (textually or graphically) in a extra compartment in its enclosing node. Second, as shown for the **LoadAgent**, you can use the defined tagged value **location** to designate the node on which the class instance resides.

You'll typically use the first form when it's important for you to give a visual cue in your diagrams about the spatial separation and grouping of components. Similarly, you'll use the second form when modeling the location of an element is important, but secondary, to the diagram at hand, such as when you want to visualize the passing of messages among instances.

Common Modeling Techniques

Modeling Timing Constraints

Modeling the absolute time of an event, modeling the relative time between events, and modeling the time it takes to carry out an action are the three primary time-critical properties of real time systems for which you'll use timing constraints.

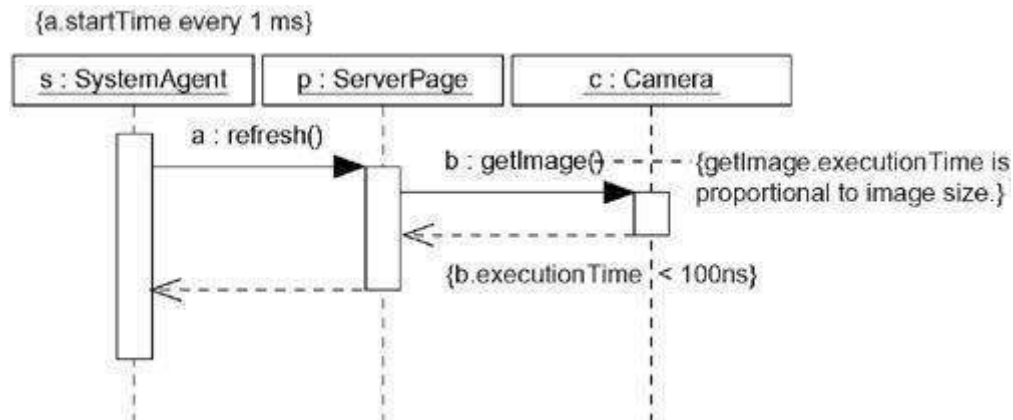
To model timing constraints,

- For each event in an interaction, consider whether it must start at some absolute time. Model that real time property as a timing constraint on the message.
- For each interesting sequence of messages in an interaction, consider whether there is an associated maximum relative time for that sequence. Model that real time property as a timing constraint on the sequence.

- For each time critical operation in each class, consider its time complexity. Model those semantics as timing constraints on the operation.

For example, as shown in Figure the left-most constraint specifies the repeating start time the call event **refresh**. Similarly, the center timing constraint specifies the maximum duration for calls to **getImage**. Finally, the right-most constraint specifies the time complexity of the call event **getImage**.

Figure Modeling Timing Constraint



Note

Observe that **executionTime** may be applied to actions such as **getImage**, as well as to timing marks such as **a** and **b**. Also, timing constraints such as these may be written as free-form text. If you want to specify your semantics more precisely, you can use the UML's Object Constraint Language (OCL), described further in *The Unified Modeling Language Reference Manual*.

Often, you'll choose short names for messages, so that you don't confuse them with operation names.

Modeling the Distribution of Objects

When you model the topology of a distributed system, you'll want to consider the physical placement of both components and class instances. If your focus is the configuration management of the deployed system, modeling the distribution of components is especially important in order to visualize, specify, construct, and document the placement of physical things such as executables, libraries, and tables. If your focus is the functionality, scalability, and throughput of the system, modeling the distribution of objects is what's important.

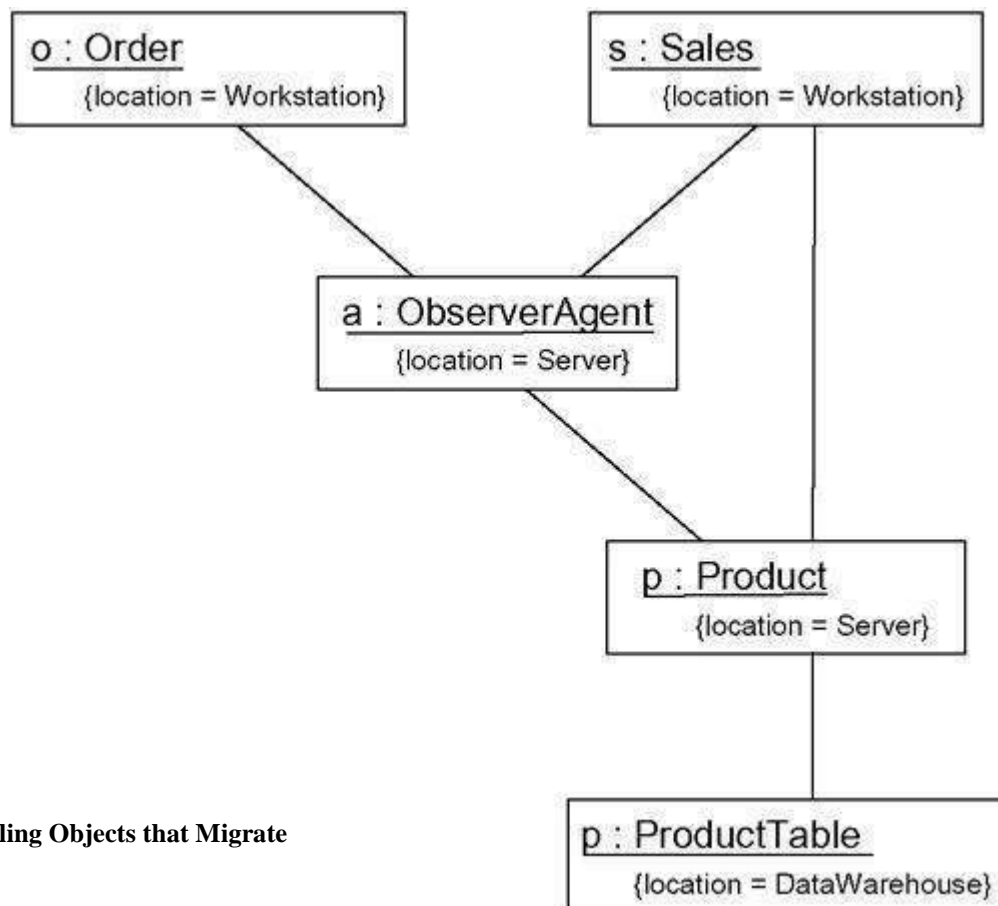
Deciding how to distribute the objects in a system is a wicked problem, and not just because the problems of distribution interact with the problems of concurrency. Naive solutions tend to yield profoundly poor performance, and over-engineering solutions aren't much better. In fact, they are probably worse because they usually end up being brittle.

To model the distribution of objects,

- For each interesting class of objects in your system, consider its locality of reference. In other words, consider all its neighbors and their locations. A tightly coupled locality will have neighboring objects close by; a loosely coupled one will have distant objects (and thus, there will be latency in communicating with them). Tentatively allocate objects closest to the actors that manipulate them.
- Next consider patterns of interaction among related sets of objects. Co-locate sets of objects that have high degrees of interaction, to reduce the cost of communication. Partition sets of objects that have low degrees of interaction.
- Next consider the distribution of responsibilities across the system. Redistribute your objects to balance the load of each node.
- Consider also issues of security, volatility, and quality of service, and redistribute your objects as appropriate.
- Render this allocation in one of two ways:
 1. By nesting objects in the nodes of a deployment diagram
 2. By explicitly indicating the location of the object as a tagged value

Figure provides an object diagram that models the distribution of certain objects in a retail system. The value of this diagram is that it lets you visualize the physical distribution of certain key objects. As the diagram shows, two objects reside on a **Workstation** (the **Order** and **Sales** objects), two objects reside on a **Server** (the **ObserverAgent** and the **Product** objects), and one object resides on a **DataWarehouse** (the **ProductTable** object).

Figure Modeling the Distribution of Objects



Modeling Objects that Migrate

For many distributed systems, components and objects, once loaded on the system, stay put. For their lifetime, from creation to destruction, they never leave the node on which they were born. There are certain classes of distributed systems, however, for which things move about, usually for one of two reasons.

First, you'll find objects migrating in order to move closer to actors and other objects they need to work with to do their job better. For example, in a global shipping system, you'd see objects that represent ships, containers, and manifests moving from node to node to track their physical counterpart. If you have a ship in Hong Kong, it makes for better locality of reference to put the object representing the ship, its containers, and its manifest on a node in Hong Kong. When that ship sails to San Diego, you'd want to move the associated objects, as well.

Second, you'll find objects migrating in response to the failure of a node or connection or to balance the load across multiple nodes. For example, in an air traffic control system, the failure of one node cannot be allowed to stall a nation's entire operations. Rather, a failure-tolerant system such as this will migrate elements to other nodes. Performance and throughput may be reduced,

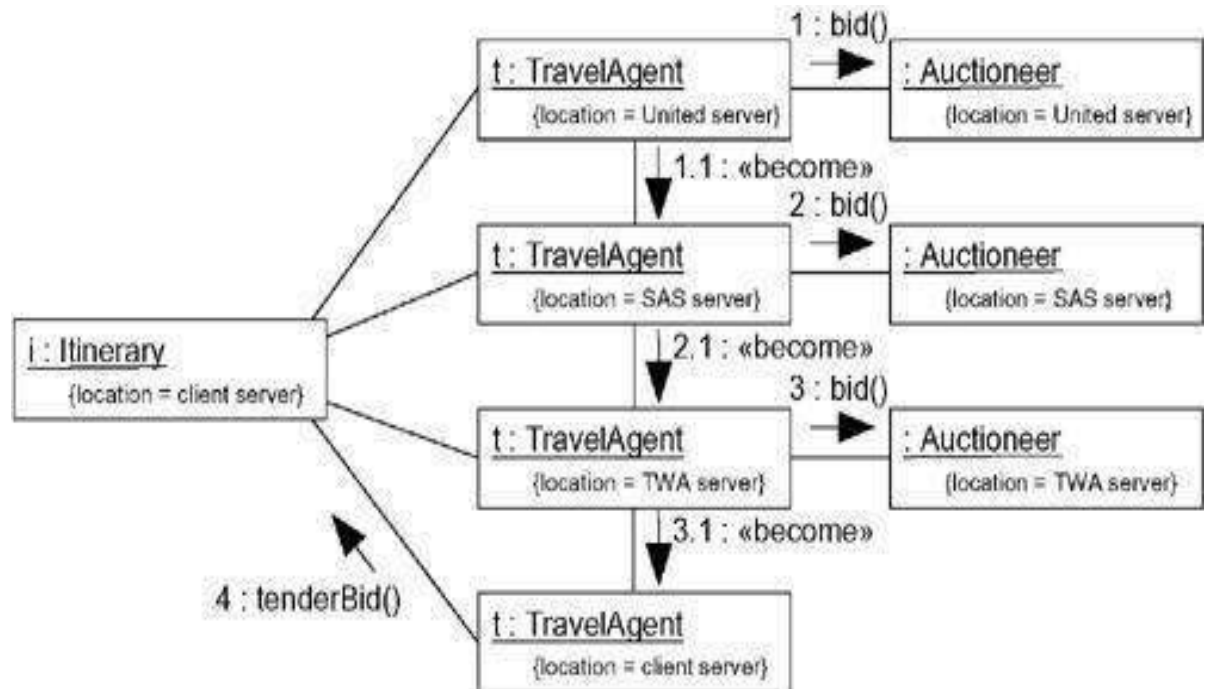
but safe functionality will be preserved. Similarly, and especially in Web-based systems that must deal with unpredictable peaks in demand, you'll often want to build mechanisms to automatically balance the processing load, perhaps by migrating components and objects to underused nodes.

Deciding how to migrate the objects in a system is an even more wicked problem than simple static distribution because migration raises difficult problems of synchronization and preservation of identity.

To model the migration of objects,

Figure provides a collaboration diagram that models the migration of a Web agent that moves from node to node, collecting information and bidding on resources in order to automatically deliver a lowest-cost travel ticket. Specifically, this diagram shows an instance (named **t**) of the class **TravelAgent** migrating from one server to another. Along the way, the object interacts with anonymous **Auctioneer** instances at each node, eventually delivering a bid for the **Itinerary** object, located on the **client server**.

Figure Modeling Objects that Migrate



Statechart Diagrams

Terms and Concepts

A *statechart diagram* shows a state machine, emphasizing the flow of control from state to state. A *state machine* is a behavior that specifies the sequences of states an object goes through during its lifetime in response to events, together with its responses to those events. A *state* is a condition or situation in the life of an object during which it satisfies some condition, performs some activity, or waits for some event. An *event* is the specification of a significant occurrence that has a location in time and space. In the context of state machines, an event is an occurrence of a stimulus that can trigger a state transition. A *transition* is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. An *activity* is ongoing nonatomic execution within a state machine. An *action* is an executable atomic computation that results in a change in state of the model or the return of a value. Graphically, a statechart diagram is a collection of vertices and arcs.

Common Properties

A statechart diagram is just a special kind of diagram and shares the same common properties as do all other **diagrams**• that is, a name and graphical contents that are a projection into a model. What distinguishes a statechart diagram from all other kinds of diagrams is its content.

Contents

Statechart diagrams commonly contain

- Simple states and composite states
- Transitions, including events and actions distinguishes an activity diagram from a statechart diagram is that an activity diagram is basically a projection of the elements found in an activity graph, a special case of a state machine in which all or most states are activity states and in which all or most transitions are triggered by completion of activities in the source state.

Common Uses

You use statechart diagrams to model the dynamic aspects of a system. These dynamic aspects may involve the event- ordered behavior of any kind of object in any view of a system's architecture, including classes (which includes active classes), interfaces, components, and nodes.

When you use a statechart diagram to model some dynamic aspect of a system, you do so in the context of virtually any modeling element. Typically, however, you'll use statechart diagrams in the context of the system as a whole, a subsystem, or a class. You can also attach statechart diagrams to use cases (to model a scenario).

When you model the dynamic aspects of a system, a class, or a use case, you'll typically use statechart diagrams in one way.

- To model reactive objects

A reactive — or event-driven — object is one whose behavior is best characterized by its response to events dispatched from outside its context. A reactive object is typically idle until it receives an event. When it receives an event, its response usually depends on previous events. After the object responds to an event, it becomes idle again, waiting for the next event. For these kinds of objects, you'll focus on the stable states of that object, the events that trigger a transition from state to state, and the actions that occur on each state change.

Common Modeling Technique

Modeling Reactive Objects

The most common purpose for which you'll use statechart diagrams is to model the behavior of reactive objects, especially instances of classes, use cases, and the system as a whole. Whereas interactions model the behavior of a society of objects working together, a statechart diagram models the behavior of a single object over its lifetime. Whereas an activity diagram models the flow of control from activity to activity, a statechart diagram models the flow of control from event to event.

When you model the behavior of a reactive object, you essentially specify three things: the stable states in which that object may live, the events that trigger a transition from state to state, and the actions that occur on each state change. Modeling the behavior of a reactive object also involves modeling the lifetime of an object, starting at the time of the object's creation and continuing until its destruction, highlighting the stable states in which the object may be found.

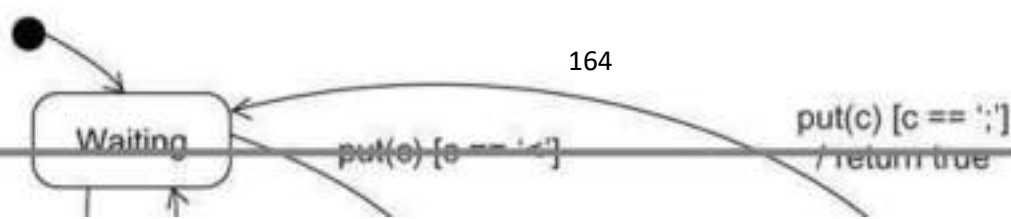
A stable state represents a condition in which an object may exist for some identifiable period of time. When an event occurs, the object may transition from state to state. These events may also trigger self- and internal transitions, in which the source and the target of the transition are the same state. In reaction to an event or a state change, the object may respond by dispatching an action.

To model a reactive object,

- Choose the context for the state machine, whether it is a class, a use case, or the system as a whole.
- Choose the initial and final states for the object. To guide the rest of your model, possibly state the pre- and postconditions of the initial and final states, respectively.
- Decide on the stable states of the object by considering the conditions in which the object may exist for some identifiable period of time. Start with the high-level states of the object and only then consider its possible substates.
- Decide on the meaningful partial ordering of stable states over the lifetime of the object.
- Decide on the events that may trigger a transition from state to state. Model these events as triggers to transitions that move from one legal ordering of states to another.
- Attach actions to these transitions (as in a Mealy machine) and/or to these states (as in a Moore machine).
- Consider ways to simplify your machine by using substates, branches, forks, joins, and history states.
- Check that all states are reachable under some combination of events.
- Check that no state is a dead end from which no combination of events will transition the object out of that state.
- Trace through the state machine, either manually or by using tools, to check it against expected sequences of events and their responses.

For example, Figure shows the statechart diagram for parsing a simple context-free language, such as you might find in systems that stream in or stream out messages to XML. In this case, the machine is designed to parse a stream of characters that match the syntax

Figure Modeling Reactive Objects



The first string represents a tag; the second string represents the body of the message. Given a stream of characters, only well-formed messages that follow this syntax may be accepted.

As the figure shows, there are only three stable states for this state machine: **Waiting**, **GettingToken**, and **GettingBody**. This statechart is designed as a Mealy machine, with actions tied to transitions. In fact, there is only one event of interest in this state machine, the invocation of **put** with the actual parameter **c** (a character). While **Waiting**, this machine throws away any character that does not designate the start of a token (as specified by the guard condition). When the start of a token is received, the state of the object changes to **GettingToken**. While in that state, the machine saves any character that does not designate the end of a token (as specified by the guard condition). When the end of a token is received, the state of the object changes to **GettingBody**. While in that state, the machine saves any character that does not designate the end of a message body (as specified by the guard condition). When the end of a message is received, the state of the object changes to **Waiting**, and a value is returned indicating that the message has been parsed (and the machine is ready to receive another message). Note that this statechart specifies a machine that runs continuously; there is no final state.

Forward and Reverse Engineering

Forward engineering (the creation of code from a model) is possible for statechart diagrams, especially if the context of the diagram is a class. For example, using the previous statechart diagram, a forward engineering tool could generate the following Java code for the class

MessageParser.

```
class MessageParser { public
  boolean put(char c) { switch (state) {
    case Waiting:
      if (c == '<') {
        state = GettingToken;
        token = new StringBuffer(); body = new
        StringBuffer();
      }
      break;
```

```

        caseGettingToken : if (c == '>')
            state = GettingBody; else
            token.append(c);
            break;
        caseGettingBody : if (c == ';')
            state = Waiting; else
            body.append(c); return true;
    }
    return false;
}
StringBuffergetToken() { return token;
}
StringBuffergetBody() { return body;
}
private
    final static int Waiting = 0; final static intGettingToken =
    1; final static intGettingBody = 2; int state = Waiting;
    StringBuffer token, body;
}

```

This requires a little cleverness. The forward engineering tool must generate the necessary private attributes and final static constants.

Reverse engineering (the creation of a model from code) is theoretically possible, but practically not very useful. The choice of what constitutes a meaningful state is in the eye of the designer. Reverse engineering tools have no capacity for abstraction and therefore cannot automatically produce meaningful statechart diagrams. More interesting than the reverse engineering of a model from code is the animation of a model against the execution of a deployed system. For example, given the previous diagram, a tool could animate the states in the diagram as they were reached in the running system. Similarly, the firing of transitions could be animated, showing the receipt of events and the resulting dispatch of actions. Under the control of a debugger, you could control the speed of execution, setting breakpoints to stop the action at interesting states to examine the attribute values of individual objects.

Components

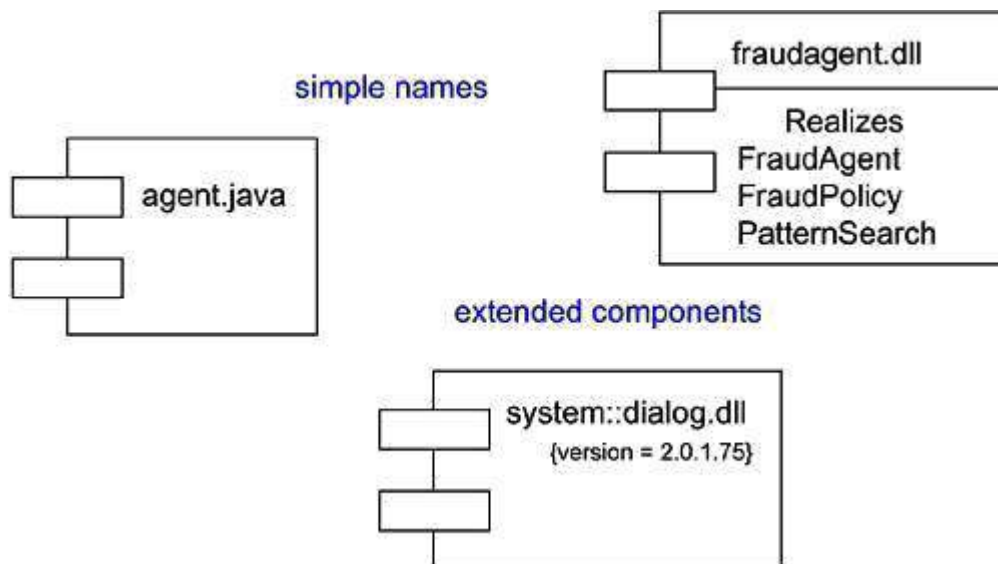
Terms and Concepts

A *component* is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Graphically, a component is rendered as a rectangle with tabs.

Names

Every component must have a name that distinguishes it from other components. A *name* is a textual string. That name alone is known as a *simple name*; a *path name* is the component name prefixed by the name of the package in which that component lives. A component is typically drawn showing only its name, as in Figure. Just as with classes, you may draw components adorned with tagged values or with additional compartments to expose their details, as you see in the figure.

Figure Simple and Extended Component



Components and Classes

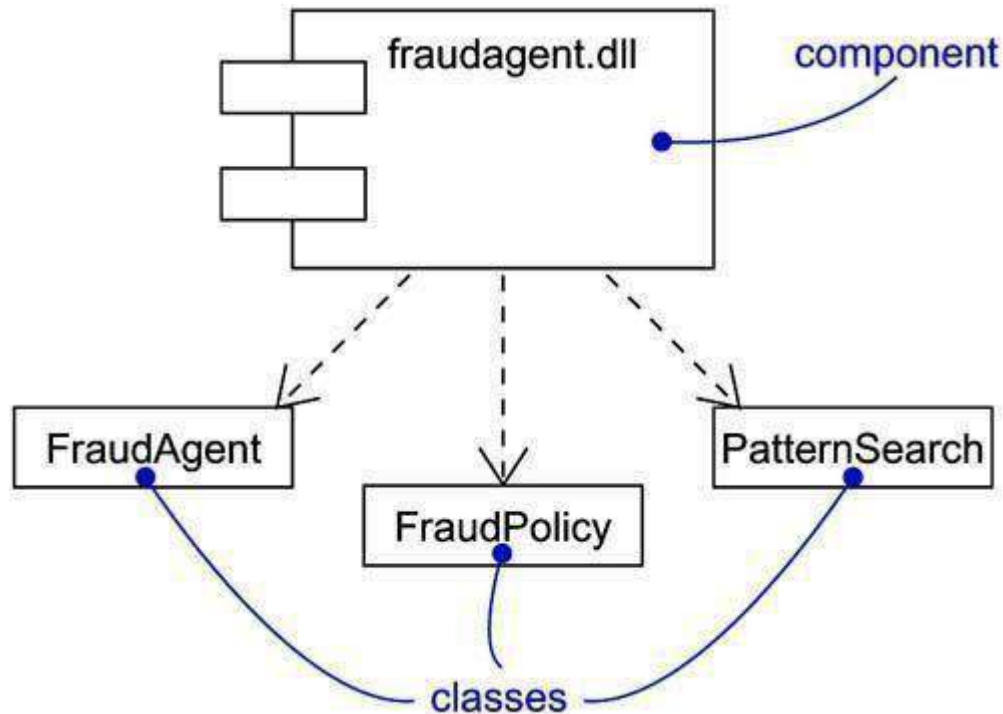
In many ways, components are like classes: Both have names; both may realize a set of interfaces; both may participate in dependency, generalization, and association relationships; both may be nested; both may have instances; both may be participants in interactions. However, there are some significant differences between components and classes.

- Classes represent logical abstractions; components represent physical things that live in the world of bits. In short, components may live on nodes, classes may not.
- Components represent the physical packaging of otherwise logical components and are at a different level of abstraction.
- Classes may have attributes and operations directly. In general, components only have operations that are reachable only through their interfaces.

The first difference is the most important. When modeling a system, deciding if you should use a class or a component involves **a simple decision• if the thing** you are modeling lives directly on a node, use a component; otherwise, use a class.

The second difference suggests a relationship between classes and components. In particular, a component is the physical implementation of a set of other logical elements, such as classes and collaborations. As Figure shows, the relationship between a component and the classes it implements can be shown explicitly by using a dependency relationship. Most of the time, you'll never need to visualize these relationships graphically. Rather, you will keep them as a part of the component's specification.

Figure Components and Classes



The third difference points out how interfaces bridge components and classes. As described in more detail in the next section, components and classes may both realize an interface, but a component's services are usually available only through its interfaces.

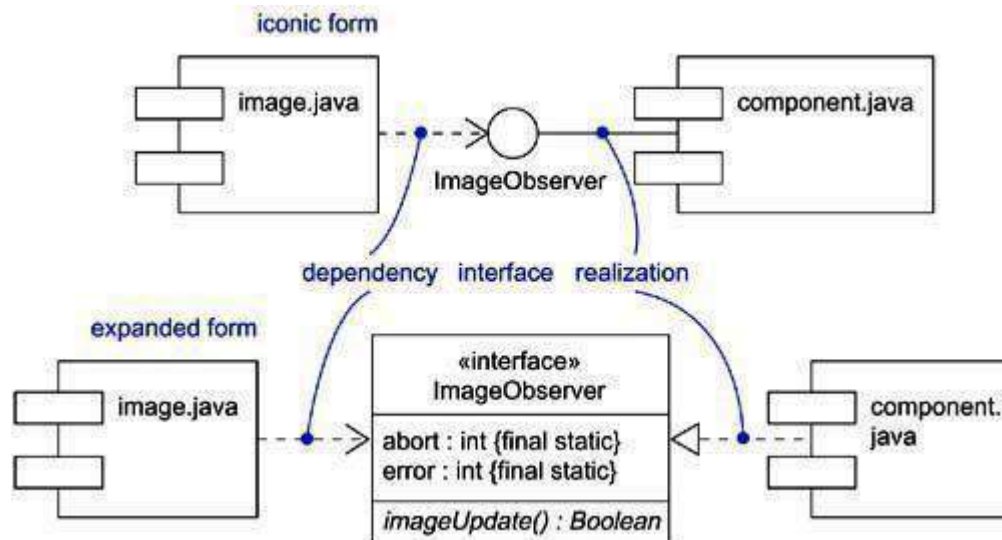
Components and Interfaces

An interface is a collection of operations that are used to specify a service of a class or a component. The relationship between component and interface is important. All the most common component-based operating system facilities (such as COM+, CORBA, and Enterprise Java Beans) use interfaces as the glue that binds components together.

Using one of these facilities, you decompose your physical implementation by specifying interfaces that represent the major seams in the system. You then provide components that realize the interfaces, along with other components that access the services through their interfaces. This mechanism permits you to deploy a system whose services are somewhat location-independent and, as discussed in the next section, replaceable.

As Figure indicates, you can show the relationship between a component and its interfaces in one of two ways. The first (and most common) style renders the interface in its elided, iconic form. The component that realizes the interface is connected to the interface using an elided realization relationship. The second style renders the interface in its expanded form, perhaps revealing its operations. The component that realizes the interface is connected to the interface using a full realization relationship. In both cases, the component that accesses the services of the other component through the interface is connected to the interface using a dependency relationship.

Figure Components and Interfaces



An interface that a component realizes is called an *export interface*, meaning an interface that the component provides as a service to other components. A component may provide many export interfaces. The interface that a component uses is called an *import interface*, meaning an interface that the component conforms to and so builds on. A component may conform to many import interfaces. Also, a component may both import and export interfaces.

A given interface may be exported by one component and imported by another. The fact that this interface lies between the two components breaks the direct dependency between the components. A component that uses a given interface will function properly no matter what component realizes that interface. Of course, a component can be used in a context if and only if all its import interfaces are provided by the export interfaces of other components.

Binary Replaceability

The basic intent of every component-based operating system facility is to permit the assembly of systems from binary replaceable parts. This means that you can create a system out of components and then evolve that system by adding new components and replacing old ones, without rebuilding the system. Interfaces are the key to making this happen. When you specify an interface, you can drop into the executable system any component that conforms to or provides that interface. You can extend the system by making the components provide new services through other interfaces, which, in turn, other components can discover and use. These semantics explain the intent behind the definition of components in the UML. A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces.

First, a component is *physical*. It lives in the world of bits, not concepts.

Second, a component is *replaceable*. **A component is substitutable• it is possible to replace a component** with another that conforms to the same interfaces. Typically, the mechanism of inserting or replacing a component to form a run time system is transparent to the component user and is enabled by object models (such as COM+ and Enterprise Java Beans) that require little or no intervening transformation or by tools that automate the mechanism.

Third, a component is *part of a system*. A component rarely stands alone. Rather, a given component collaborates with other components and in so doing exists in the architectural or technology context in which it is intended to be used. A component is logically and physically cohesive and thus denotes a meaningful structural and/or behavioral

chunk of a larger system. A component may be reused across many systems. Therefore, a component represents a fundamental building block on which systems can be designed and composed. **This definition is recursive• a system at one level of abstraction may simply be a component at a higher level of abstraction.**

Fourth, as discussed in the previous section, a component *conforms to and provides the realization of a set of interfaces*.

Kinds of Components

Three kinds of components may be distinguished.

First, there are *deployment components*. These are the components necessary and sufficient to form an executable system, such as dynamic libraries (DLLs) and executables (EXEs). The UML's definition of component is broad enough to address classic object models, such as COM+, CORBA, and Enterprise Java Beans, as well as alternative object models, perhaps involving dynamic Web pages, database tables, and executables using proprietary communication mechanisms.

Second, there are *work product components*. These components are essentially the residue of the development process, consisting of things such as source code files and data files from which deployment components are created. These components do not directly participate in an executable system but are the work products of development that are used to create the executable system.

Third are *execution components*. These components are created as a consequence of an executing system, such as a COM+ object, which is instantiated from a DLL.

Organizing Components

You can organize components by grouping them in packages in the same manner in which you organize classes.

You can also organize components by specifying dependency, generalization, association (including aggregation), and realization relationships among them.

Standard Elements

All the UML's extensibility mechanisms apply to components. Most often, you'll use tagged values to extend component properties (such as specifying the version of a development component) and stereotypes to specify new kinds of components (such as operating system-specific components).

The UML defines five standard stereotypes that apply to components:

1. executable	Specifies a component that may be executed on a node
2. library	Specifies a static or dynamic object library
3. table	Specifies a component that represents a database table
4. file	Specifies a component that represents a document containing source code or Data
5. document	Specifies a component that represents a document

Common Modeling Techniques

Modeling Executables and Libraries

The most common purpose for which you'll use components is to model the deployment components that make up your implementation. If you are deploying a trivial system whose implementation consists of exactly one executable file, you will not need to do any component modeling. If, on the other hand, the system you are deploying is made up of several executables and associated object libraries, doing component modeling will help you to visualize, specify, construct, and document the decisions you've made about the physical system. Component modeling is even more important if you want to control the versioning and configuration management of these parts as your system evolves.

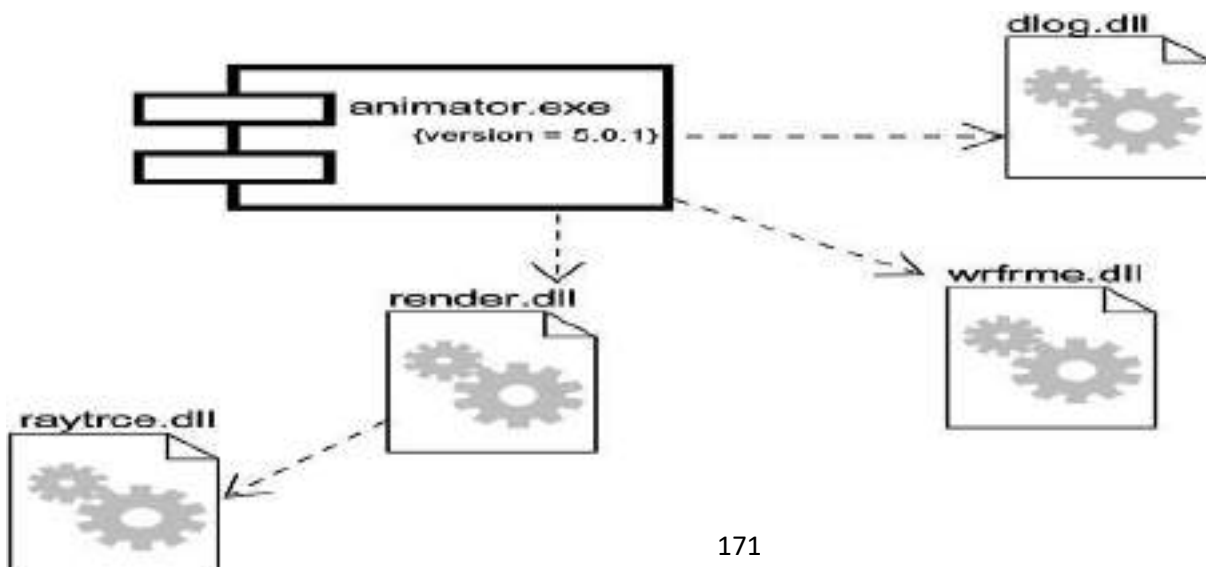
For most systems, these deployment components are drawn from the decisions you make about how to segment the physical implementation of your system. These decisions will be affected by a number of technical issues (such as your choice of component-based operating system facilities), configuration management issues (such as your decisions about which parts will likely change over time), and reuse issues (that is, deciding which components you can reuse in or from other systems).

To model executables and libraries,

- Identify the partitioning of your physical system. Consider the impact of technical, configuration management, and reuse issues.
- Model any executables and libraries as components, using the appropriate standard elements. If your implementation introduces new kinds of components, introduce a new appropriate stereotype.
- If it's important for you to manage the seams in your system, model the significant interfaces that some components use and others realize.
- As necessary to communicate your intent, model the relationships among these executables, libraries, and interfaces. Most often, you'll want to model the dependencies among these parts in order to visualize the impact of change.

For example, Figure shows a set of components drawn from a personal productivity tool that runs on a single personal computer. This figure includes one executable (**animator.exe**, with a tagged value noting its version number) and four libraries (**dlog.dll**, **wrfrme.dll**, **render.dll**, and **raytrce.dll**), all of which use the UML's standard elements for executables and libraries, respectively. This diagram also presents the dependencies among these components.

Figure Modeling Executables and Libraries



As your models get bigger, you will find that many components tend to cluster together in groups that are conceptually and semantically related. In the UML, you can use packages to model these clusters of components. For larger systems that are deployed across several computers, you'll want to model the way your components are distributed by asserting the nodes on which they are located.

Modeling Tables, Files, and Documents

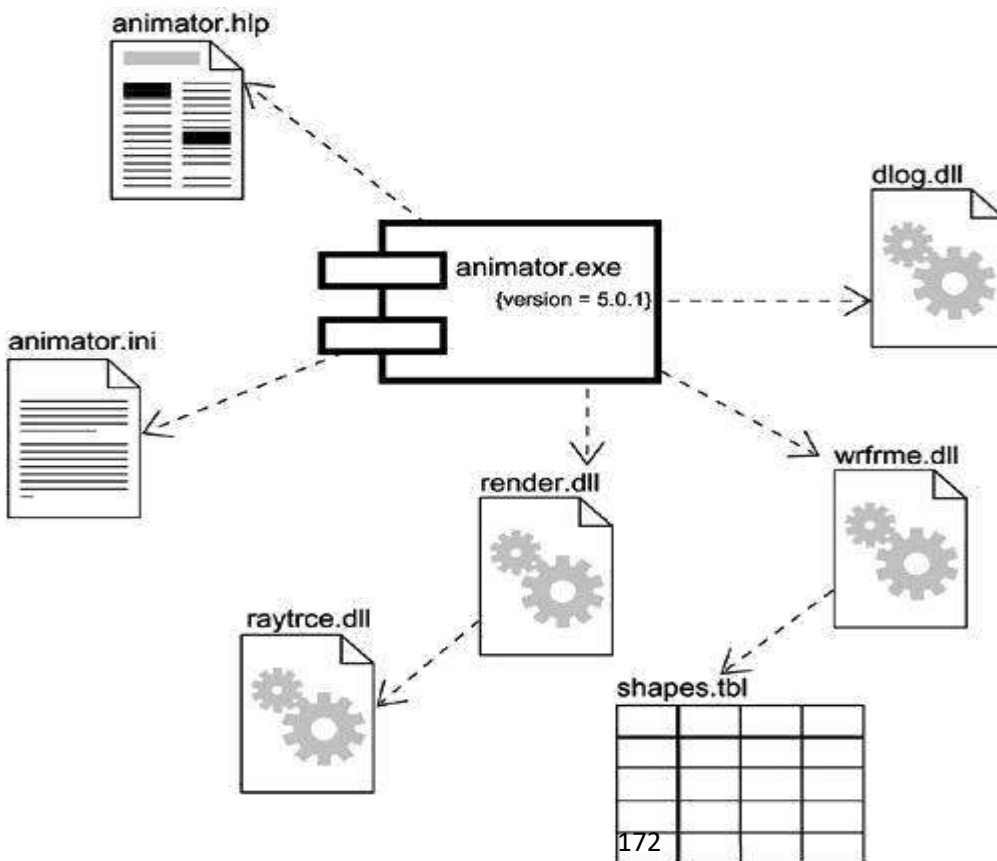
Modeling the executables and libraries that make up the physical implementation of your system is useful, but often you'll find there are a host of ancillary deployment components that are neither executables nor libraries and yet are critical to the physical deployment of your system. For example, your implementation might include data files, help documents, scripts, log files, initialization files, and installation/removal files. Modeling these components is an important part of controlling the configuration of your system. Fortunately, you can use UML components to model all of these artifacts.

To model tables, files, and documents,

- Identify the ancillary components that are part of the physical implementation of your system.
- Model these things as components. If your implementation introduces new kinds of artifacts, introduce a new appropriate stereotype.
- As necessary to communicate your intent, model the relationships among these ancillary components and the other executables, libraries, and interfaces in your system. Most often, you'll want to model the dependencies among these parts in order to visualize the impact of change.

For example, Figure builds on the previous figure and shows the tables, files, and documents that are part of the deployed system surrounding the executable **animator.exe**. This figure includes one document (**animator.hlp**), one simple file (**animator.ini**), and one database table (**shapes.tbl**), all of which use the UML's standard elements for components, files, and tables, respectively.

Figure Modeling Tables, Files, and Documents



Modeling databases can get complicated when you start dealing with multiple tables, triggers, and stored procedures. To visualize, specify, construct, and document these features, you'll need to model the logical schema, as well as the physical databases.

Modeling an API

If you are a developer who's assembling a system from component parts, you'll often want to see the application programming interfaces (APIs) that you can use to glue these parts together. APIs represent the programmatic seams in your system, which you can model using interfaces and components.

An API is essentially an interface that is realized by one or more components. As a developer, you'll really care only about the interface itself; which component realizes an interface's operations is not relevant as long as *some* component realizes it. From a system configuration management perspective, though, these realizations are important because you need to ensure that, when you publish an API, there's some realization available that carries out the API's obligations. Fortunately, with the UML, you can model both perspectives.

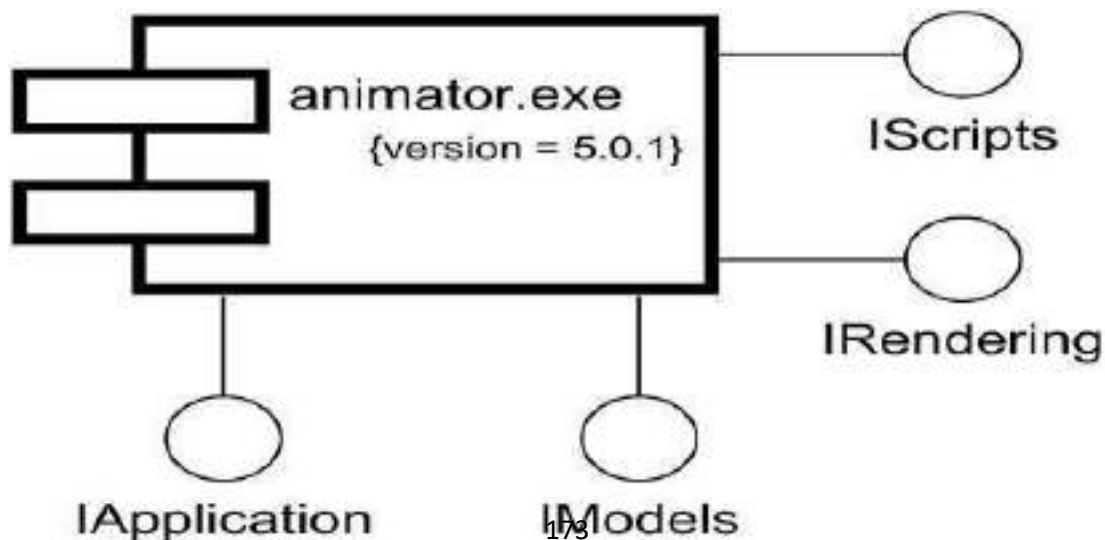
The operations associated with any semantically rich API will be fairly extensive, so most of the time you won't need to visualize all these operations at once. Instead, you'll tend to keep the operations in the backplane of your models and use interfaces as handles with which you can find these sets of operations. If you want to construct executable systems against these APIs, you will need to add enough detail so that your development tools can compile against the properties of your interfaces. Along with the signatures of each operation, you'll probably also want to include use cases that explain how to use each interface.

To model an API,

- Identify the programmatic seams in your system and model each seam as an interface, collecting the attributes and operations that form this edge.
- Expose only those properties of the interface that are important to visualize in the given context; otherwise, hide these properties, keeping them in the interface's specification for reference, as necessary.
- Model the realization of each API only insofar as it is important to show the configuration of a specific implementation.

Figure exposes the APIs of the executable in the previous two figures. You'll see four interfaces that form the API of the executable: **IApplication**, **IModels**, **IRendering**, and **IScripts**.

Figure Modeling an API



Modeling Source Code

The most common purpose for which you'll use components is to model the physical parts that make up your implementation. This also includes the modeling of all the ancillary parts of your deployed system, including tables, files, documents, and APIs. The second most common purpose for which you'll use components is to model the configuration of all the source code files that your development tools use to create these components. These represent the work product components of your development process.

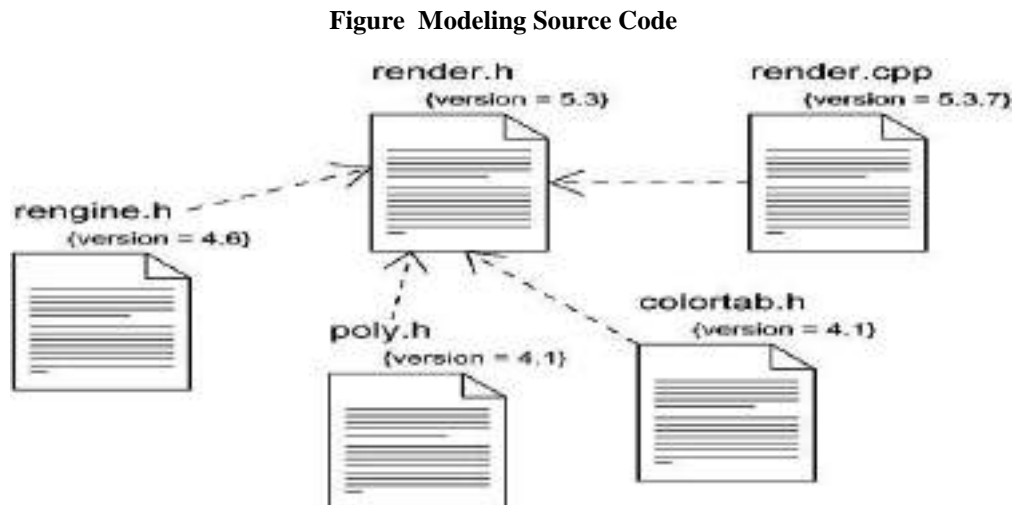
Modeling source code graphically is particularly useful for visualizing the compilation dependencies among your source code files and for managing the splitting and merging of groups of these files when you fork and join development paths. In this manner, UML components can be the graphical interface to your configuration management and version control tools.

For most systems, source code files are drawn from the decisions you make about how to segment the files your development environment needs. These files are used to store the details of your classes, interfaces, collaborations, and other logical elements as an intermediate step to creating the physical, binary components that are derived from these elements by your tools. Most of the time, these tools will impose a style of organization (one or two files per class is common), but you'll still want to visualize the relationships among these files. How you organize groups of these files using packages and how you manage versions of these files is driven by your decisions about how to manage change.

To model source code,

- Depending on the constraints imposed by your development tools, model the files used to store the details of all your logical elements, along with their compilation dependencies.
- If it's important for you to bolt these models to your configuration management and version control tools, you'll want to include tagged values, such as version, author, and check in/check out information, for each file that's under configuration management.
- As far as possible, let your development tools manage the relationships among these files, and use the UML only to visualize and document these relationships.

For example, Figure shows some source code files that are used to build the library **render.dll** from the previous examples. This figure includes four header files (**render.h**, **engine.h**, **poly.h**, and **colortab.h**) that represent the source code for the specification of certain classes. There is also one implementation file (**render.cpp**) that represents the implementation of one of these headers.



As your models get bigger, you will find that many source code files tend to cluster together in groups that are conceptually and semantically related. Most of the time, your development tools will place these groups in separate directories. In the UML, you can use packages to model these clusters of source code files.

In the UML, it is possible to visualize the relationship of a class to its source code file and, in turn, the relationship of a source code file to its executable or library by using trace relationships. However, you'll rarely need to go to this detail of modeling.

Deployment

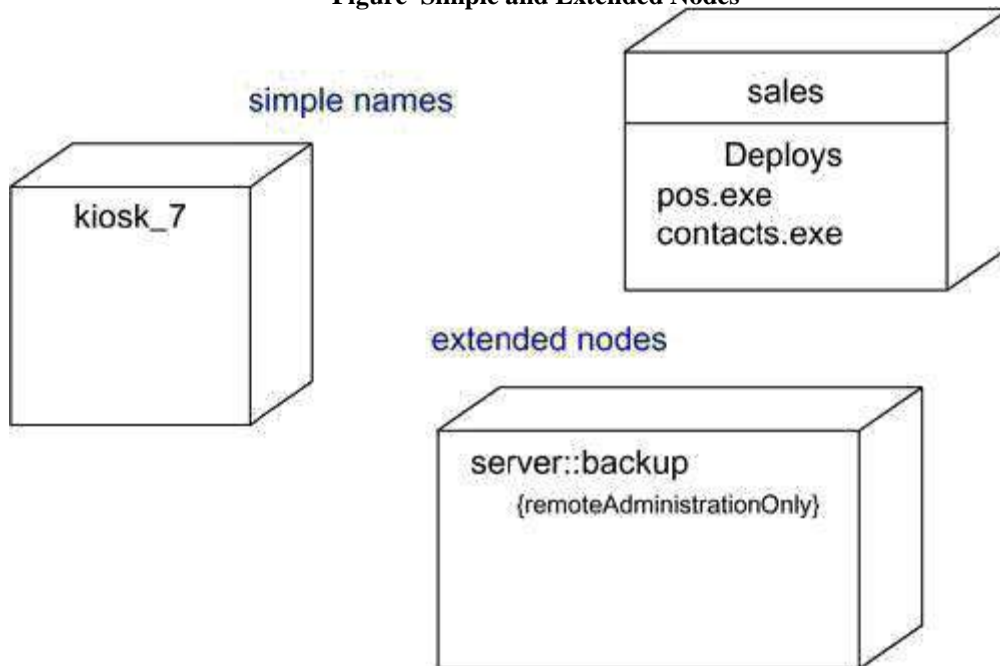
Terms and Concepts

A *node* is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. Graphically, a node is rendered as a cube.

Names

Every node must have a name that distinguishes it from other nodes. A *name* is a textual string. That name alone is known as a *simple name*; a *path name* is the node name prefixed by the name of the package in which that node lives. A node is typically drawn showing only its name, as in Figure . Just as with classes, you may draw nodes adorned with tagged values or with additional compartments to expose their details.

Figure Simple and Extended Nodes



Note

A node name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the colon, which is used to separate a node name and the name of its enclosing package) and may continue over several lines. In practice, node names are short nouns or noun phrases drawn from the vocabulary of the implementation.

Nodes and Components

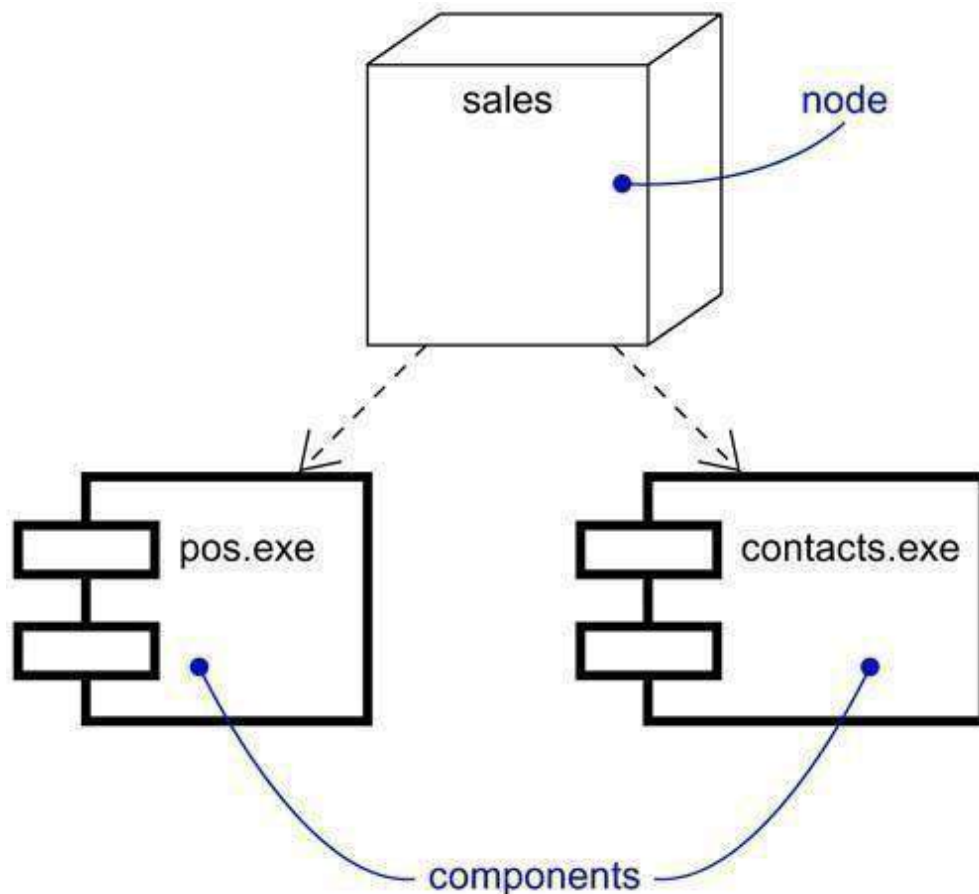
In many ways, nodes are a lot like components: Both have names; both may participate in dependency, generalization, and association relationships; both may be nested; both may have instances; both may be participants in interactions. However, there are some significant differences between nodes and components.

- Components are things that participate in the execution of a system; nodes are things that execute components.
- Components represent the physical packaging of otherwise logical elements; nodes represent the physical deployment of components.

This first difference is the most important. Simply put, nodes execute components; components are things that are executed by nodes.

The second difference suggests a relationship among classes, components, and nodes. In particular, a component is the materialization of a set of other logical elements, such as classes and collaborations, and a node is the location upon which components are deployed. A class may be implemented by one or more components, and, in turn, a component may be deployed on one or more nodes. As Figure shows, the relationship between a node and the components it deploys can be shown explicitly by using a dependency relationship. Most of the time, you won't need to visualize these relationships graphically but will keep them as a part of the node's specification.

Figure Nodes and Components



A set of objects or components that are allocated to a node as a group is called a *distribution unit*.

Note

Nodes are also class-like in that you can specify attributes and operations for them. For example, you might specify that a node provides the attributes **processorSpeed** and **memory**, as well as the operations **turnOn**, **turnOff**, and **suspend**.

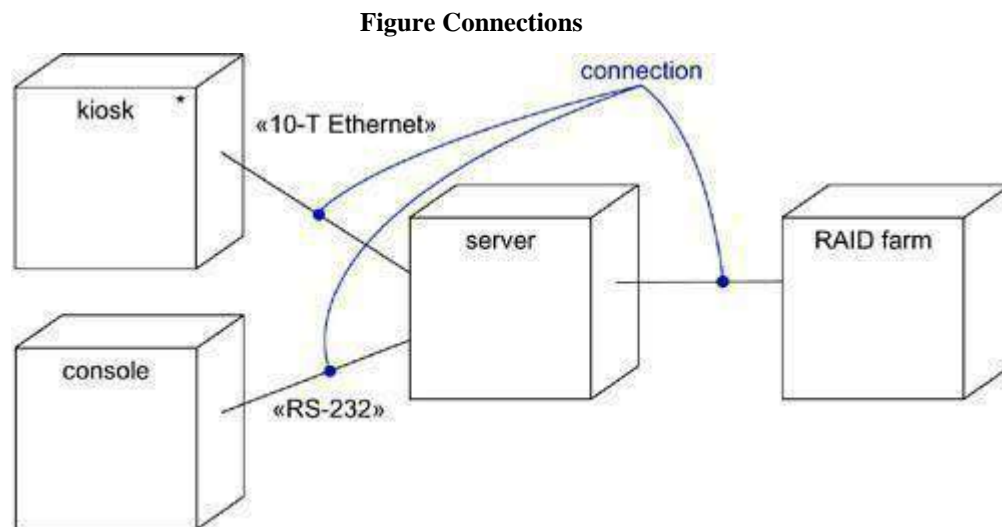
Organizing Nodes

You can organize nodes by grouping them in packages in the same manner in which you can organize classes and components.

You can also organize nodes by specifying dependency, generalization, and association (including aggregation) relationships among them.

Connections

The most common kind of relationship you'll use among nodes is an association. In this context, an association represents a physical connection among nodes, such as an Ethernet connection, a serial line, or a shared bus, as Figure shows. You can even use associations to model indirect connections, such as a satellite link between distant processors.



Because nodes are class-like, you have the full power of associations at your disposal. This means that you can include roles, multiplicity, and constraints. As in the previous figure, you should stereotype these associations if you want to model new kinds of **connections**• for example, to distinguish between a 10-T Ethernet connection and an RS-232 serial connection.

Common Modeling Techniques

Modeling Processors and Devices

Modeling the processors and devices that form the topology of a stand-alone, embedded, client/server, or distributed system is the most common use of nodes. Because all of the UML's extensibility mechanisms apply to nodes, you will often use stereotypes to specify new kinds of nodes that you can use to represent specific kinds of processors and devices. A *processor* is a

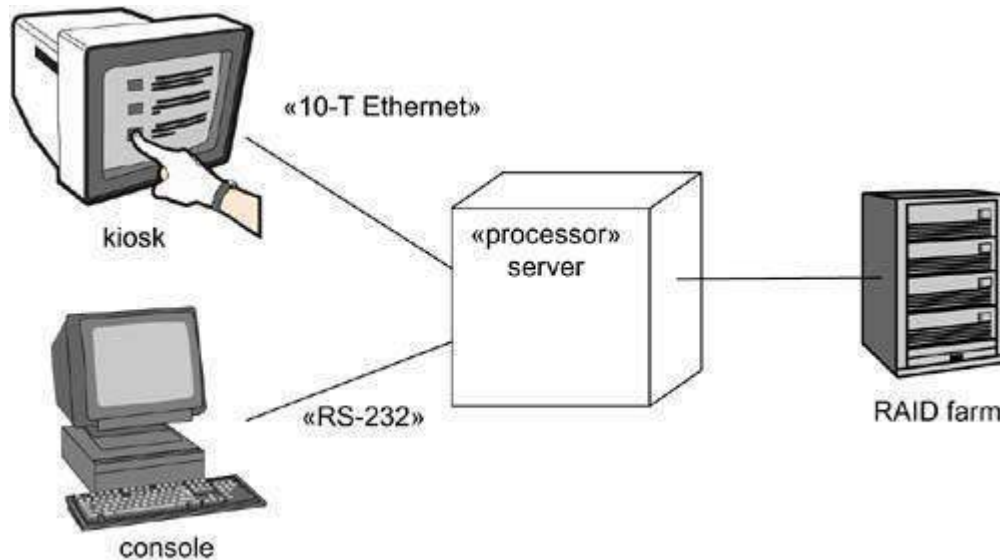
node that has processing capability, meaning that it can execute a component. A *device* is a node that has no processing capability (at least, none that are modeled at this level of abstraction) and, in general, represents something that interfaces to the real world.

To model processors and devices,

- Identify the computational elements of your system's deployment view and model each as a node. If these elements represent generic processors and devices, then stereotype them as such. If they are kinds of processors and devices that are part of the vocabulary of your domain, then specify an appropriate stereotype with an icon for each.
- As with class modeling, consider the attributes and operations that might apply to each node.

For example, Figure takes the previous diagram and stereotypes each node. The **server** is a node stereotyped as a generic processor; the **kiosk** and the **console** are nodes stereotyped as special kinds of processors; and the **RAID farm** is a node stereotyped as a special kind of device.

Figure Processors and Devices



Modeling the Distribution of Components

When you model the topology of a system, it's often useful to visualize or specify the physical distribution of its components across the processors and devices that make up the system.

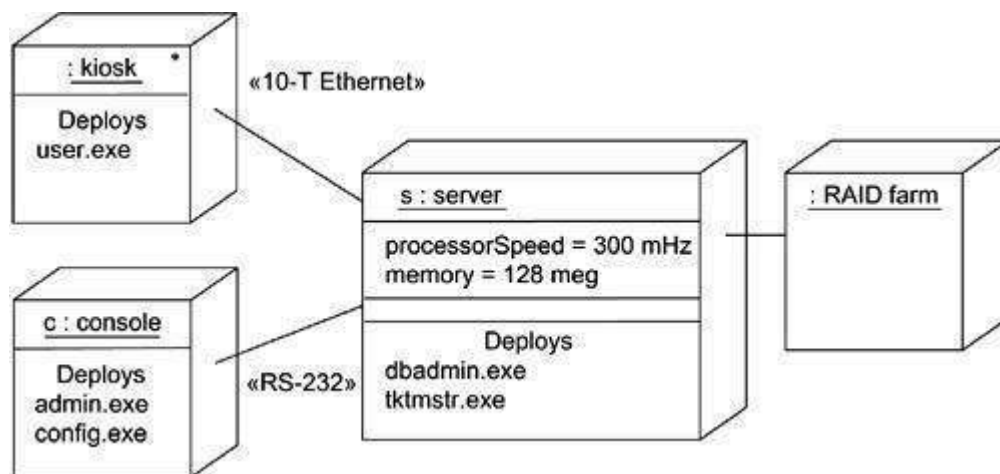
To model the distribution of components,

- For each significant component in your system, allocate it to a given node.

- Consider duplicate locations for components. It's not uncommon for the same kind of component (such as specific executables and libraries) to reside on multiple nodes simultaneously.
- Render this allocation in one of three ways.
 1. Don't make the allocation visible, but leave it as part of the backplane of your **model**• that is, in each node's specification.
 2. Using dependency relationships, connect each node with the components it deploys.
 3. List the components deployed on a node in an additional compartment.

Using the third approach, Figure takes the earlier diagrams and specifies the executable components that reside on each node. This diagram is a bit different from the previous ones in that it is an object diagram, visualizing specific instances of each node. In this case, the **RAID farm** and **kiosk** instances are both anonymous and the other two instances are named (c for the **console** and s for the **server**). Each processor in this figure is rendered with an additional compartment showing the component it deploys. The **server** object is also rendered with its attributes (**processorSpeed** and **memory**) and their values visible.

Figure Modeling the Distribution of Components.



Components need not be statically distributed across the nodes in a system. In the UML, it is possible to model the dynamic migration of components from node to node, as in an agent-based system or a high-reliability system that involves clustered servers and replicated databases.

Collaborations

Component Diagrams

Terms and Concepts

A *component diagram* shows a set of components and their relationships. Graphically, a component diagram is a collection of vertices and arcs.

Common Properties

A component diagram is just a special kind of diagram and shares the same common properties as do all **other diagrams**• a **name and graphical** contents that are a projection into a model. What distinguishes a component diagram from all other kinds of diagrams is its particular content.

Contents

Component diagrams commonly contain

- Components
- Interfaces
- Dependency, generalization, association, and realization relationships Like

all other diagrams, component diagrams may contain notes and constraints.

Component diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place instances in your component diagrams, as well, especially when you want to visualize one instance of a family of component-based systems.

Common Uses

You use component diagrams to model the static implementation view of a system. This view primarily supports the configuration management of a system's parts, made up of components that can be assembled in various ways to produce a running system.

When you model the static implementation view of a system, you'll typically use component diagrams in one of four ways.

1. To model source code

With most contemporary object- oriented programming languages, you'll cut code using integrated development environments that store your source code in files. You can use component diagrams to model the configuration management of these files, which represent work-product components.

2. To model executable releases

A release is a relatively complete and consistent set of artifacts delivered to an internal or external user. In the context of components, a release focuses on the parts necessary to deliver a running system. When you model a release using component diagrams, you are visualizing, specifying, and documenting the decisions about the physical parts **that constitute your software• that is, its deployment components.**

3. To model physical databases

Think of a physical database as the concrete realization of a schema, living in the world of bits. Schemas, in effect, offer an API to persistent information; the model of a physical database represents the storage of that information in the tables of a relational database or the pages of an object-oriented database. You use component diagrams to represent these and other kinds of physical databases.

4. To model adaptable systems

Some systems are quite static; their components enter the scene, participate in an execution, and then depart. Other systems are more dynamic, involving mobile agents or components that migrate for purposes of load balancing and failure recovery. You use component diagrams in conjunction with some of the UML's diagrams for modeling behavior to represent these kinds of systems.

Common Modeling Techniques

Modeling Source Code

If you develop software in Java, you'll usually save your source code in **.java** files. If you develop software using C++, you'll typically store your source code in header files (**.h** files) and bodies (**.cpp** files). If you use IDL to develop COM+ or CORBA applications, one interface from your design view will often expand into four source code files: the interface itself, the client proxy, the server stub, and a bridge class. As your application grows, no matter which language you use, you'll find yourself organizing these files into larger groups. Furthermore, during the construction phase of development, you'll probably end up creating new versions of some of these files for each new incremental release you produce, and you'll want to place these versions under the control of a configuration management system.

Much of the time, you will not need to model this aspect of a system directly. Instead, you'll let your development environment keep track of these files and their relationships. Sometimes, however, it's helpful to visualize these source code files and their relationships using component diagrams. Component diagrams used in this way typically contain only work-product components stereotyped as files, together with dependency relationships. For example, you might reverse engineer a set of source code files to visualize their web of compilation dependencies. You can go in the other direction by specifying the relationships among your source code files and then using those models as input to compilation tools, such as **make** on Unix. Similarly, you might want to use component diagrams to visualize the history of a set of source code files that are under configuration management. By extracting information from your configuration management system, such as the number of times a source code file has been checked out over a period of time, you can use that information to color component diagrams, showing "hot spots" of change among your source code files and areas of architectural churn.

To model a system's source code,

- Either by forward or reverse engineering, identify the set of source code files of interest and model them as components stereotyped as files.
- For larger systems, use packages to show groups of source code files.
- Consider exposing a tagged value indicating such information as the version number of the source code file, its author, and the date it was last changed. Use tools to manage the value of this tag.
- Model the compilation dependencies among these files using dependencies. Again, use tools to help generate and manage these dependencies.

For example, Figure shows five source code files. **signal.h** is a header file. Three of its versions are shown, tracing from new versions back to their older ancestors. Each variant of this source code file is rendered with a tagged value exposing its version number.

Figure Modeling Source Code



This header file (**signal.h**) is used by two other files (**interp.cpp** and **.signal.cpp**), both of which are bodies. One of these files (**interp.cpp**) has a compilation dependency to another header (**irq.h**); in turn, **device.cpp** has a compilation dependency to **interp.cpp**. Given this component diagram, it's easy to trace the impact of changes. For example, changing the source code file **signal.h** will require the recompilation of three other files: **signal.cpp**, **interp.cpp**, and transitively, **device.cpp**. As this diagram also shows, the file **irq.h** is not affected.

Diagrams such as this can easily be generated by reverse engineering from the information held by your development environment's configuration management tools.

Modeling an Executable Release

Releasing a simple application is easy: You throw the bits of a single executable file on a disk, and your users just run that executable. For these kinds of applications, you don't need component diagrams because there's nothing difficult to visualize, specify, construct, or document.

Releasing anything other than a simple application is not so easy. You need the main executable (usually, a **.exe** file), but you also need all its ancillary parts, such as libraries (commonly **.dll** files if you are working in the context of COM+, or **.class** and **.jar** files if you are working in the context of Java), databases, help files, and resource files. For distributed systems, you'll likely have multiple executables and other parts scattered across various nodes. If you are working with a system of applications, you'll find that some of these components are unique to each application but that many are shared among applications. As you evolve your system, controlling the configuration of these many components **becomes an important activity• and a more difficult one** because changes in the components associated with one application may affect the operation of other applications.

For this reason, you use component diagrams to visualize, specify, construct, and document the configuration of your executable releases, encompassing the deployment components that form each release and the relationships among those components. You can use component diagrams to forward engineer a new system and to reverse engineer an existing one.

When you create component diagrams such as these, you actually just model a part of the things and relationships that make up your system's implementation view. For this reason, each component diagram should focus on one set of components at a time.

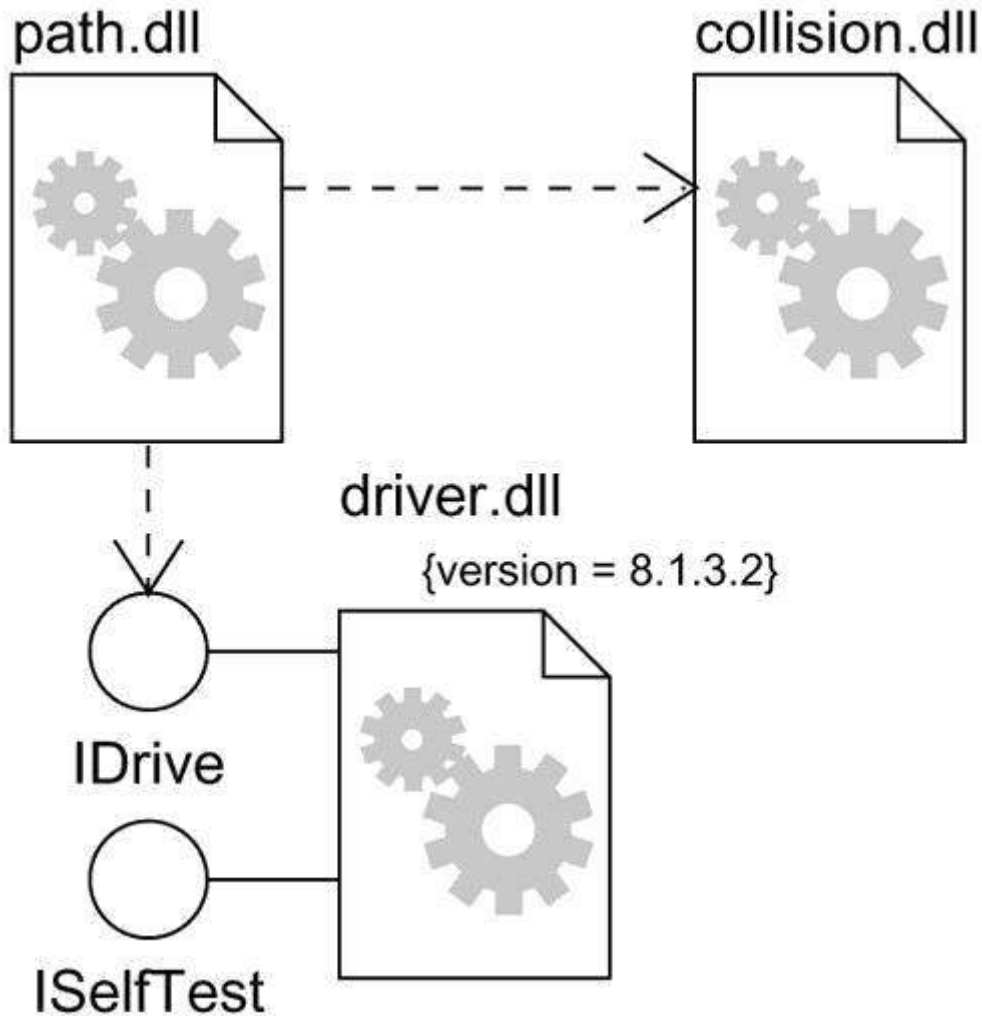
To model an executable release,

- Identify the set of components you'd like to model. Typically, this will involve some or all the components that live on one node, or the distribution of these sets of components across all the nodes in the system.
- Consider the stereotype of each component in this set. For most systems, you'll find a small number of different kinds of components (such as executables, libraries, tables, files, and documents). You can use the UML's extensibility mechanisms to provide visual cues for these stereotypes.
- For each component in this set, consider its relationship to its neighbors. Most often, this will involve interfaces that are exported (realized) by certain components and then imported (used) by others. If you want to expose the seams in your system, model these interfaces explicitly. If you want your model at a higher level of abstraction, elide these relationships by showing only dependencies among the components.

For example, Figure models part of the executable release for an autonomous robot. This figure focuses on the deployment components associated with the robot's driving and calculation functions. You'll find one component (**driver.dll**) that exports an interface (**IDrive**) that is, in turn, imported by another component (**path.dll**). **driver.dll** exports one other interface (**ISelfTest**) that is probably used by other

components in the system, although they are not shown here. There's one other component shown in this diagram (**collision.dll**), and it, too, exports a set of interfaces, although these details are elided: **path.dll** is shown with a dependency directly to **collision.dll**.

Figure Modeling an Executable Release



There are many more components involved in this system. However, this diagram only focuses on those deployment components that are directly involved in moving the robot. Note that in this component-based architecture, you could replace a specific version of **driver.dll** with another that realized the same (and perhaps additional) interfaces, and **path.dll** would still function properly. If you want to be explicit about the operations that **driver.dll** realizes, you could always render its interface using class notation, stereotyped as **»interface**.

Modeling a Physical Database

A logical database schema captures the vocabulary of a system's persistent data, along with the semantics of their relationships. Physically, these things are stored in a database for later retrieval, either a relational database, an object-oriented one, or a hybrid object/relational database. The UML is well suited to modeling physical databases, as well as logical database schemas.

Physical database design is beyond the scope of this book; the focus here is simply to show you how you can model databases and tables using the UML. Mapping a logical database schema to an object-oriented

database is straightforward because even complex inheritance lattices can be made persistent directly. Mapping a logical database schema to a relational database is not so simple, however. In the presence of inheritance, you have to make decisions about how to map classes to tables. Typically, you can apply one or a combination of three strategies.

1. Define a separate table for each class. This is a simple but naive approach because it introduces maintenance headaches when you add new child classes or modify your parent classes.
2. Collapse your inheritance lattices so that all instances of any class in a hierarchy has the same state. The downside with this approach is that you end up storing superfluous information for many instances.
3. Separate parent and child states into different tables. This approach best mirrors your inheritance lattice, but the downside is that traversing your data will require many cross-table joins.

When designing a physical database, you also have to make decisions about how to map operations defined in your logical database schema. Object- oriented databases make the mapping fairly transparent. But, with relational databases, you have to make some decisions about how these logical operations are implemented. Again, you have some choices.

1. For simple CRUD (create, read, update, delete) operations, implement them with standard SQL or ODBC calls.
2. For more-complex behavior (such as business rules), map them to triggers or stored procedures.

Given these general guidelines, to model a physical database,

- Identify the classes in your model that represent your logical database schema.
- Select a strategy for mapping these classes to tables. You will also want to consider the physical distribution of your databases. Your mapping strategy will be affected by the location in which you want your data to live on your deployed system.
- To visualize, specify, construct, and document your mapping, create a component diagram that contains components stereotyped as tables.
- Where possible, use tools to help you transform your logical design into a physical design.

Figure shows a set of database tables drawn from an information system for a school. You will find one database (**school.db**, rendered as a component stereotyped as **database**) that's composed of five tables: **student**, **class**, **instructor**, **department**, and **course** (rendered as a component stereotyped as **table**, one of the UML's standard elements). In the corresponding logical database schema, there was no inheritance, so mapping to this physical database design is straightforward.

Figure Modeling a Physical Database



Although not shown in this example, you can specify the contents of each table. Components can have attributes, so a common idiom when modeling physical databases is to use these attributes to specify the columns of each table. Similarly, components can have operations, and these can be used to denote stored procedures.

Modeling Adaptable Systems

All the component diagrams shown thus far have been used to model static views. Their components spend their entire lives on one node. This is the most common situation you'll encounter, but especially in the domain of complex, distributed systems, you'll need to model dynamic views. For example, you might have a system that replicates its databases across several nodes, switching the one that is the primary database when a server goes down. Similarly, if you are modeling a globally distributed 24x7 operation (that is, a system that's up 24 hours a day, 7 days a week), you will likely encounter mobile agents, components that migrate from node to node to carry out some transaction. To model these dynamic views, you'll need to use a combination of component diagrams, object diagrams, and interaction diagrams.

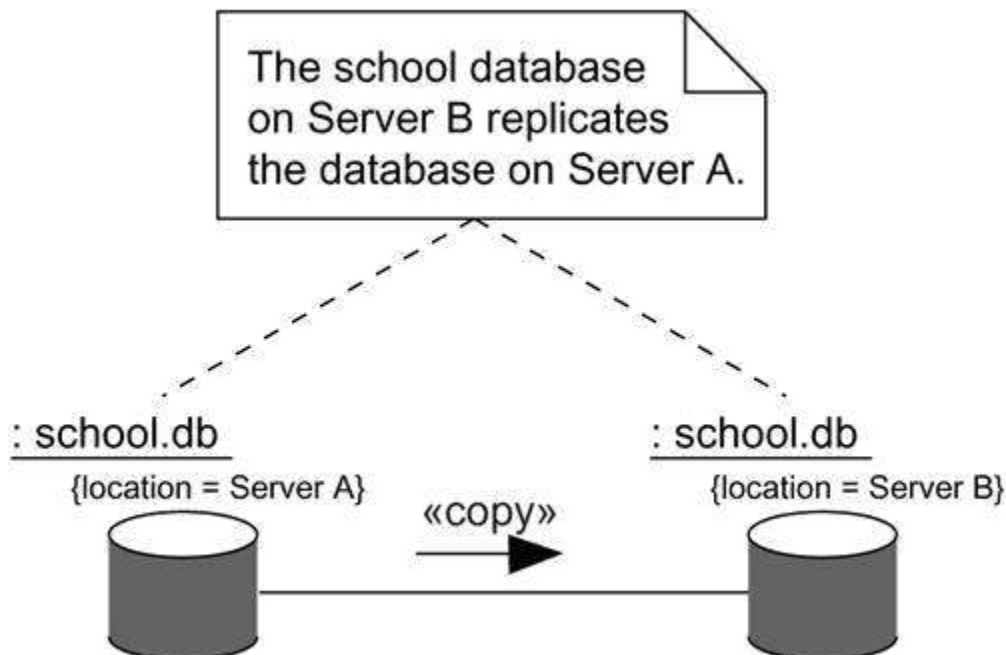
To model an adaptable system,

Consider the physical distribution of the components that may migrate from node to node. You can specify the location of a component instance by marking it with a location tagged value, which you can then render in a component diagram (although, technically speaking, a diagram that contains only instances is an object diagram).

- If you want to model the actions that cause a component to migrate, create a corresponding interaction diagram that contains component instances. You can illustrate a change of location by drawing the same instance more than once, but with different values for its location tagged value.

For example, Figure models the replication of the database from the previous figure. We show two instances of the component **school.db**. Both instances are anonymous, and both have a different value for their location tagged value. There's also a note, which explicitly specifies which instance replicates the other.

Figure Modeling Adaptable Systems



If you want to show the details of each database, you can render them in their canonical form• a

component stereotyped as a **database**. Although not shown here, you could use an interaction diagram to model the dynamics of switching from one primary database to another.

Forward and Reverse Engineering

Forward engineering and reverse engineering components are pretty direct, because components are themselves physical things (executables, libraries, tables, files, and documents) that are therefore close to the running system. When you forward engineer a class or a collaboration, you really forward engineer to a component that represents the source code, binary library, or executable for that class or collaboration. Similarly, when you reverse engineer source code, binary libraries, or executables, you really reverse engineer to a component or set of components that, in turn, trace to classes or collaborations.

Choosing to forward engineer (the creation of code from a model) a class or collaboration to source code, a binary library, or an executable is a mapping decision you have to make. You'll want to take your logical models to source code if you are interested in controlling the configuration management of files that are then manipulated by a development environment. You'll want to take your logical models directly to binary libraries or executables if you are interested in managing the components that you'll actually deploy on a running system. In some cases, you'll want to do both. A class or collaboration may be denoted by source code, as well as by a binary library or executable.

To forward engineer a component diagram,

- For each component, identify the classes or collaborations that the component implements.
- Choose the target for each component. Your choice is basically between source code (a form that can be manipulated by development tools) or a binary library or executable (a form that can be dropped into a running system).
- Use tools to forward engineer your models.

Reverse engineering (the creation of a model from code) a component diagram is not a perfect process because there is always a loss of information. From source code, you can reverse engineer back to classes; this is the most common thing you'll do. Reverse engineering source code to components will uncover compilation dependencies among those files. For binary libraries, the best you can hope for is to denote the library as a component and then discover its interfaces by reverse engineering. This is the second most common thing you'll do with component diagrams. In fact, this is a useful way to approach a set of new libraries that may be otherwise poorly documented. For executables, the best you can hope **for is to denote the executable as a component and then disassemble its code• something you'll rarely** need to do unless you work in assembly language.

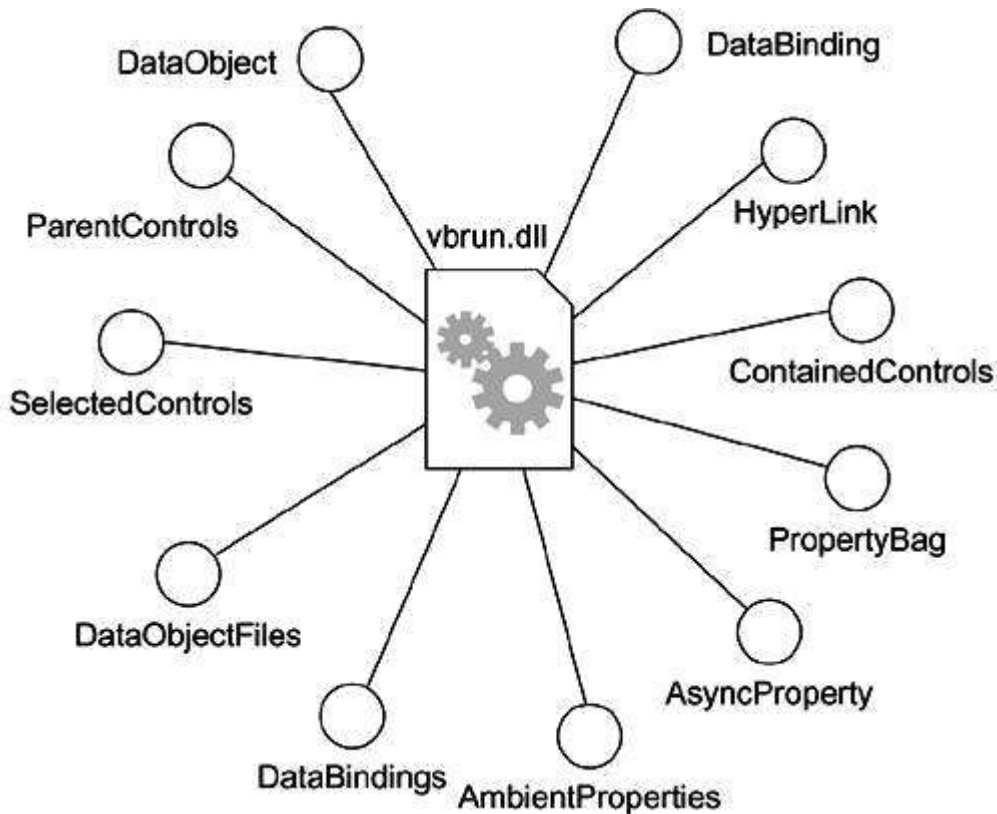
To reverse engineer a component diagram,

- Choose the target you want to reverse engineer. Source code can be reverse engineered to components and then classes. Binary libraries can be reverse engineered to uncover their interfaces. Executables can be reverse engineered the least.
- Using a tool, point to the code you'd like to reverse engineer. Use your tool to generate a new model or to modify an existing one that was previously forward engineered.
- Using your tool, create a component diagram by querying the model. For example, you might start with one or more components, then expand the diagram by following relationships or neighboring components. Expose or hide the details of the contents of this component diagram as necessary to communicate your intent.

For example, Figure provides a component diagram that represents the reverse engineering of the ActiveX component **vbrun.dll**. As the figure shows, the component realizes 11 interfaces. Given this diagram, you

can begin to understand the semantics of the component by next exploring the details of its interfaces.

Figure Reverse Engineering



Especially when you reverse engineer from source code, and sometimes when you reverse engineer from binary libraries and executables, you'll do so in the context of a configuration management system. This means that you'll often be working with specific versions of files or libraries, with all versions of a configuration compatible with one another. In these cases, you'll want to include a tagged value that represents the component version, which you can derive from your configuration management system. In this manner, you can use the UML to visualize the history of a component across various releases.

Deployment Diagrams

Terms and Concepts

A *deployment diagram* is a diagram that shows the configuration of run time processing nodes and the components that live on them. Graphically, a deployment diagram is a collection of vertices and arcs.

Common Properties

A deployment diagram is just a special kind of diagram and shares the same common properties as all **other diagrams**• a name and graphical contents that are a projection into a model. What distinguishes a deployment diagram from all other kinds of diagrams is its particular content.

Contents

Deployment diagrams commonly contain

- Nodes
- Dependency and association relationships

Like all other diagrams, deployment diagrams may contain notes and constraints.

Deployment diagrams may also contain components, each of which must live on some node. Deployment diagrams may also contain packages or subsystems, both of which are used to group elements of your model into larger chunks. Sometimes, you'll want to place instances in your deployment diagrams, as well, especially when you want to visualize one instance of a family of hardware topologies.

Note

In many ways, a deployment diagram is just a special kind of class diagram, which focuses on a system's nodes.

Common Uses

You use deployment diagrams to model the static deployment view of a system. This view primarily addresses the distribution, delivery, and installation of the parts that make up the physical system.

There are some kinds of systems for which deployment diagrams are unnecessary. If you are developing a piece of software that lives on one machine and interfaces only with standard devices on that machine that are already managed by the host operating system (for example, a personal computer's keyboard, display, and modem), you can ignore deployment diagrams. On the other hand, if you are developing a piece of software that interacts with devices that the host operating system does not typically manage or that is physically distributed across multiple processors, then using deployment diagrams will help you reason about your system's software-to-hardware mapping.

When you model the static deployment view of a system, you'll typically use deployment diagrams in one of three ways.

1. To model embedded systems

An embedded system is a software-intensive collection of hardware that interfaces with the physical world. Embedded systems involve software that controls devices such as motors, actuators, and displays and that, in turn, is controlled by external stimuli such as sensor input, movement, and temperature changes. You can use deployment diagrams to model the devices and processors that comprise an embedded system.

2. To model client/server systems

A client/server system is a common architecture focused on making a clear separation of concerns between the system's user interface (which lives on the client) and the system's persistent data (which lives on the server). Client/server systems are one end of the continuum of distributed systems and require you to make decisions about the network connectivity of clients to servers and about the physical distribution of your system's software components across the nodes. You can model the topology of such systems by using deployment diagrams.

3. To model fully distributed systems

At the other end of the continuum of distributed systems are those that are widely, if not globally, distributed, typically encompassing multiple levels of servers. Such systems are often hosts to multiple versions of software components, some of which may even migrate from node to node. Crafting such systems requires you to make decisions that enable the continuous change in the system's topology. You can use deployment diagrams to visualize the system's current topology and distribution of components to reason about the impact of changes on that topology.

Common Modeling Techniques

Modeling an Embedded System

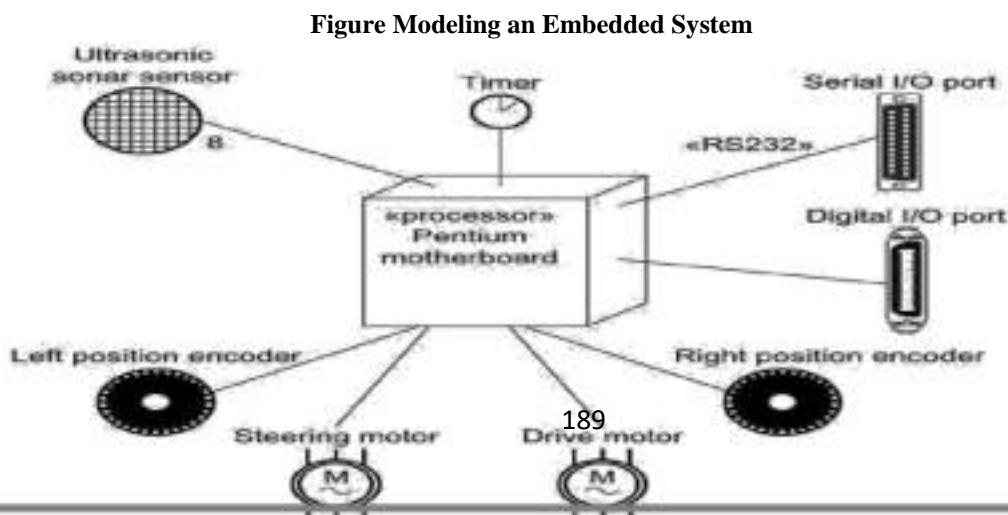
Developing an embedded system is far more than a software problem. You have to manage the physical world in which there are moving parts that break and in which signals are noisy and behavior is nonlinear. When you model such a system, you have to take into account its interface with the real world, and that means reasoning about unusual devices, as well as nodes.

Deployment diagrams are useful in facilitating the communication between your project's hardware engineers and software developers. By using nodes that are stereotyped to look like familiar devices, you can create diagrams that are understandable by both groups. Deployment diagrams are also helpful in reasoning about hardware/software trade-offs. You'll use deployment diagrams to visualize, specify, construct, and document your system engineering decisions.

To model an embedded system,

- Identify the devices and nodes that are unique to your system.
- Provide visual cues, especially for unusual devices, by using the UML's extensibility mechanisms to define system-specific stereotypes with appropriate icons. At the very least, you'll want to distinguish processors (which contain software components) and devices (which, at that level of abstraction, don't directly contain software).
- Model the relationships among these processors and devices in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.
- As necessary, expand on any intelligent devices by modeling their structure with a more detailed deployment diagram.

For example, Figure shows the hardware for a simple autonomous robot. You'll find one node (**Pentium motherboard**) stereotyped as a processor.



Surrounding this node are eight devices, each stereotyped as a device and rendered with an icon that offers a clear visual cue to its real-world equivalent.

Modeling a Client/Server System

The moment you start developing a system whose software no longer resides on a single processor, you are faced with a host of decisions: How do you best distribute your software components across these nodes? How do they communicate? How do you deal with failure and noise? At one end of the spectrum of distributed systems, you'll encounter client/server systems, in which there's a clear separation of concerns between the system's user interface (typically managed by the client) and its data (typically managed by the server).

There are many variations on this theme. For example, you might choose to have a thin client, meaning that it has a limited amount of computational capacity and does little more than manage the user interface and visualization of information. Thin clients may not even host a lot of components but, rather, may be designed to load components from the server, as needed, as with Enterprise Java Beans. On the other hand, you might choose to have a thick client, meaning that it has a goodly amount of computational capacity and does more than just visualization. A thick client typically carries out some of the system's logic and business rules. The choice between thin and thick clients is an architectural decision that's influenced by a number of technical, economic, and political factors.

Either way, partitioning a system into its client and server parts involves making some hard decisions about where to physically place its software components and how to impose a balanced distribution of responsibilities among those components. For example, most management information systems are essentially three-tier architectures, which means that the system's GUI, business logic, and database are physically distributed. Deciding where to place the system's GUI and database are usually fairly obvious, so the hard part lies in deciding where the business logic lives.

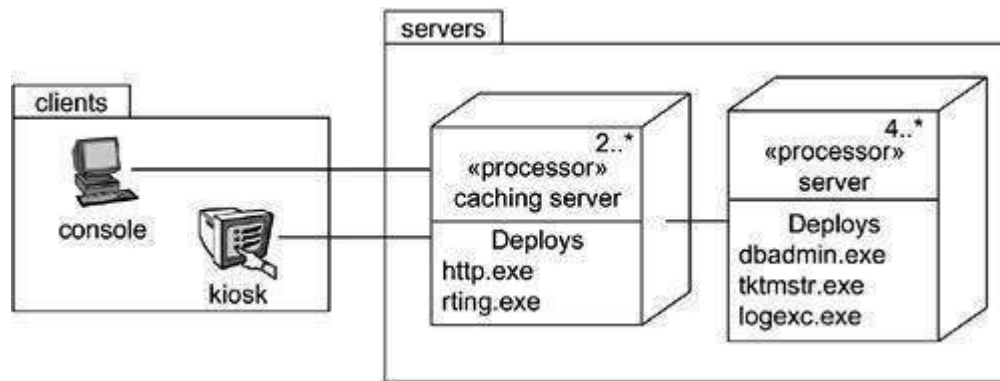
You can use the UML's deployment diagrams to visualize, specify, and document your decisions about the topology of your client/server system and how its software components are distributed across the client and server. Typically, you'll want to create one deployment diagram for the system as a whole, along with other, more detailed, diagrams that drill down to individual segments of the system.

To model a client/server system,

- Identify the nodes that represent your system's client and server processors.
- Highlight those devices that are germane to the behavior of your system. For example, you'll want to model special devices, such as credit card readers, badge readers, and display devices other than monitors, because their placement in the system's hardware topology are likely to be architecturally significant.
- Provide visual cues for these processors and devices via stereotyping.
- Model the topology of these nodes in a deployment diagram. Similarly, specify the relationship between the components in your system's implementation view and the nodes in your system's deployment view.

For example, Figure shows the topology of a human resources system, which follows a classical client/server architecture. This figure illustrates the client/server split explicitly by using the packages named **client** and **server**. The client package contains two nodes (**console** and **kiosk**), both of which are stereotyped and are visually distinguishable. The server package contains two kinds of nodes (**caching server** and **server**), and both of these have been adorned with some of the components that reside on each. Note also that **caching server** and **server** are marked with explicit multiplicities, specifying how many instances of each are expected in a particular deployed configuration. For example, this diagram indicates that there may be two or more **caching servers** in any deployed instance of the system.

Figure Modeling a Client/Server System



Modeling a Fully Distributed System

Distributed systems come in many forms, from simple two-processor systems to those that span many geographically dispersed nodes. The latter are typically never static. Nodes are added and removed as network traffic changes and processors fail; new and faster communication paths may be established in parallel with older, slower channels that are eventually decommissioned.

Not only may the topology of these systems change, but the distribution of their software components may change, as well. For example, database tables may be replicated across servers, only to be moved, as traffic dictates. For some global systems, components may follow the sun, migrating from server to server as the business day begins in one part of the world and ends in another.

Visualizing, specifying, and documenting the topology of fully distributed systems such as these are valuable activities for the systems administrator, who must keep tabs on an enterprise's computing assets. You can use the UML's deployment diagrams to reason about the topology of such systems. When you document fully distributed systems using deployment diagrams, you'll want to expand on the details of the system's networking devices, each of which you can represent as a stereotyped node.

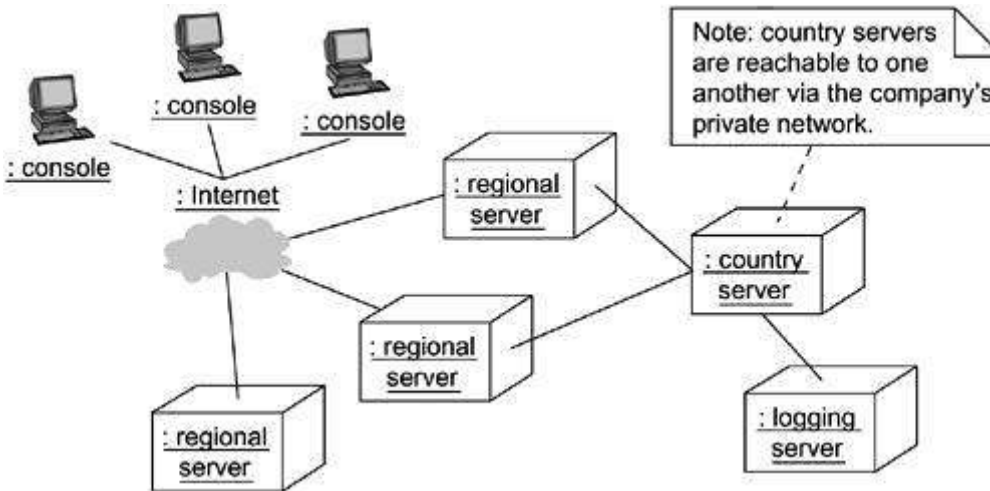
To model a fully distributed system,

- Identify the system's devices and processors as for simpler client/server systems.
- If you need to reason about the performance of the system's network or the impact of changes to the network, be sure to model these communication devices to the level of detail sufficient to make these assessments.
- Pay close attention to logical groupings of nodes, which you can specify by using packages.
- Model these devices and processors using deployment diagrams. Where possible, use tools that discover the topology of your system by walking your system's network.
- If you need to focus on the dynamics of your system, introduce use case diagrams to specify the kinds of behavior you are interested in, and expand on these use cases with interaction diagrams.

Figure shows the topology of a fully distributed system. This particular deployment diagram is also an object diagram, for it contains only instances. You can see three consoles (anonymous instances of the stereotyped node **console**), which are linked to the **Internet** (clearly a singleton node). In turn, there are three instances of **regional servers**, which serve as front ends of **country servers**, only one of which is shown. As the note indicates, country servers are connected to one another, but their relationships are not

shown in this diagram.

Figure Modeling a Fully Distributed System



In this diagram, the Internet has been reified as a stereotyped node.

Forward and Reverse Engineering

There's only a modest amount of forward engineering (the creation of code from models) that you can do with deployment diagrams. For example, after specifying the physical distribution of components across the nodes in a deployment diagram, it is possible to use tools that then push these components out to the real world. For system administrators, using the UML in this way helps you visualize what can be a very complicated task.

Reverse engineering (the creation of models from code) from the real world back to deployment diagrams is of tremendous value, especially for fully distributed systems that are under constant change. You'll want to supply a set of stereotyped nodes that speak the language of your system's network administrators, in order to tailor the UML to their domain. The advantage of using the UML is that it offers a standard language that addresses not only their needs, but the needs of your project's software developers, as well.

To reverse engineer a deployment diagram,

- Choose the target that you want to reverse engineer. In some cases, you'll want to sweep across your entire network; in others, you can limit your search.
- Choose also the fidelity of your reverse engineering. In some cases, it's sufficient to reverse engineer just to the level of all the system's processors; in others, you'll want to reverse engineer the system's networking peripherals, as well.
- Use a tool that walks across your system, discovering its hardware topology. Record that topology in a deployment model.
- Along the way, you can use similar tools to discover the components that live on each node, which you can also record in a deployment model. You'll want to use an intelligent search, for even a basic personal computer can contain gigabytes of components, many of which may not be relevant to your system.
- Using your modeling tools, create a deployment diagram by querying the model. For example,

you might start with visualizing the basic client/server topology, then expand on the diagram by populating certain nodes with components of interest that live on them. Expose or hide the details of the contents of this deployment diagram as necessary to communicate your intent.

UNIT-5

The unified library Application

INTRODUCTION:

USECASE DIAGRAM:

A behavioral diagram that shows a set of use cases and actors and their relation ships.

USECASE:

It is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor.

An use case is used to structure the behavioral things in a model and it is rendered as an ellipse with solid line along with its name.

Actor:

It specifies a coherent set of roles that users of use cases play when interacting with these use cases.

CLASS DIAGRAM:

A structural diagram that shows a set of classes, interfaces, collaborations, and their relationships.

OBJECT DIAGRAM:

A structural diagram that shows a set of objects and their relationships.

SEQUENCE DIAGRAM:

A behavioral diagram that shows an interaction, emphasizing the time ordering of messages.

COLLABORATION DIAGRAM:

A behavioural diagram that shows an interaction, emphasizing the structural organization of the objects that send and receive meessages.

STATECHART DIAGRAM:

A behavioural diagram that shows a state machine, emphasizing the event-ordered behavior of an object.

ACTIVITY DIAGRAM :

An activity diagram represents the execution state of a mechanism as a sequence of steps grouped sequentially as parallel conrol flow branches.It is a variant of state chart diagrams organized according to actions and internal behavior of a method or a usecase.

Activity diagram s are used to model the dynamic aspects of system.



 Forking

 Joining

COMPONENT DIAGRAM:

Component diagrams are basically used to model static view of the system. This can be achieved by modeling various physical components like libraries, tables, files etc. which are residing within a node.

Component diagrams are very essential for constructing executable systems. This can be done using concepts of forward and reverse engineering. The graphical representation of a component diagrams basically include collection of vertices and arcs.

DEPLOYMENT DIAGRAM:

Deployment is the stage of development that describes the configuration of the running system in a real time environment. For deployment, decisions should be made about configuration parameters, performance, resource allocation, distribution and concurrency. The component developed or reused should be deployed on some set of hardware for execution. Nodes are used to model the topology of the hardware on which the system executes. A node usually represents a processor or a device on which components can be deployed.

Case Study 1: Library Application

1. Problem Statement:

A lightweight set of features for the first version of the library application might look like this:

It is a support system for a library. The library lends books and magazines to borrowers, who are registered in the system, as are the books and magazines. The library handles the purchase of new titles for the library. Popular titles are bought in multiple copies. Old books and magazines are removed when they are out of date or in poor condition. The librarian is an employee of the library who interacts with the customers (borrowers) and whose work is supported by the system. A borrower can reserve a book or magazine that is not currently available in the library, so that when **it's returned** or purchased by the library, that borrower is notified. The reservation is cancelled when the borrower checks out the book or magazine or through an explicit canceling procedure. The librarian can easily create, update, and delete information about the titles, borrowers, loans, and reservations in the system. The system can run on all popular Web browser platforms (Internet Explorer 5.1+, Netscape 4.0+, and so on). The system is easy to extend with new functionality.

2. Identification of actors and use cases:

The use cases in the library system are as follows:

- Login
- Search ■
- Browse
- Make Reservation

- Remove Reservation
- Checkout Item
- Return Item
- Manage Titles
- Manage Items
- Manage Borrowers
- Manage Librarians
- Assume Identity of Borrower

The outline of the basic flow for the use case Checkout Item (which means that a Borrower can check out an Item) is described as follows:

1. The borrower chooses **to perform a “Search”** for desired titles/
2. The system prompts the borrower to enter Search criteria.
3. The borrower specifies the search criteria and submits the search.
4. The system locates matching titles and displays them to the borrower.
5. The borrower selects a title to check out.
6. The system displays the details of the title, as well as whether or not there is an available item to be checked out.
7. The borrower confirms that he or she wishes to checkout the item.
8. The system checks out the item.
9. Steps 1 to 8 can be repeated as often as desired by the borrower.
10. The borrower completes checkout.
11. The system notifies a librarian that the borrower has concluded the checkout item session and displays instructions for the borrower to collect the contents.

3: Identification of actors and use cases:

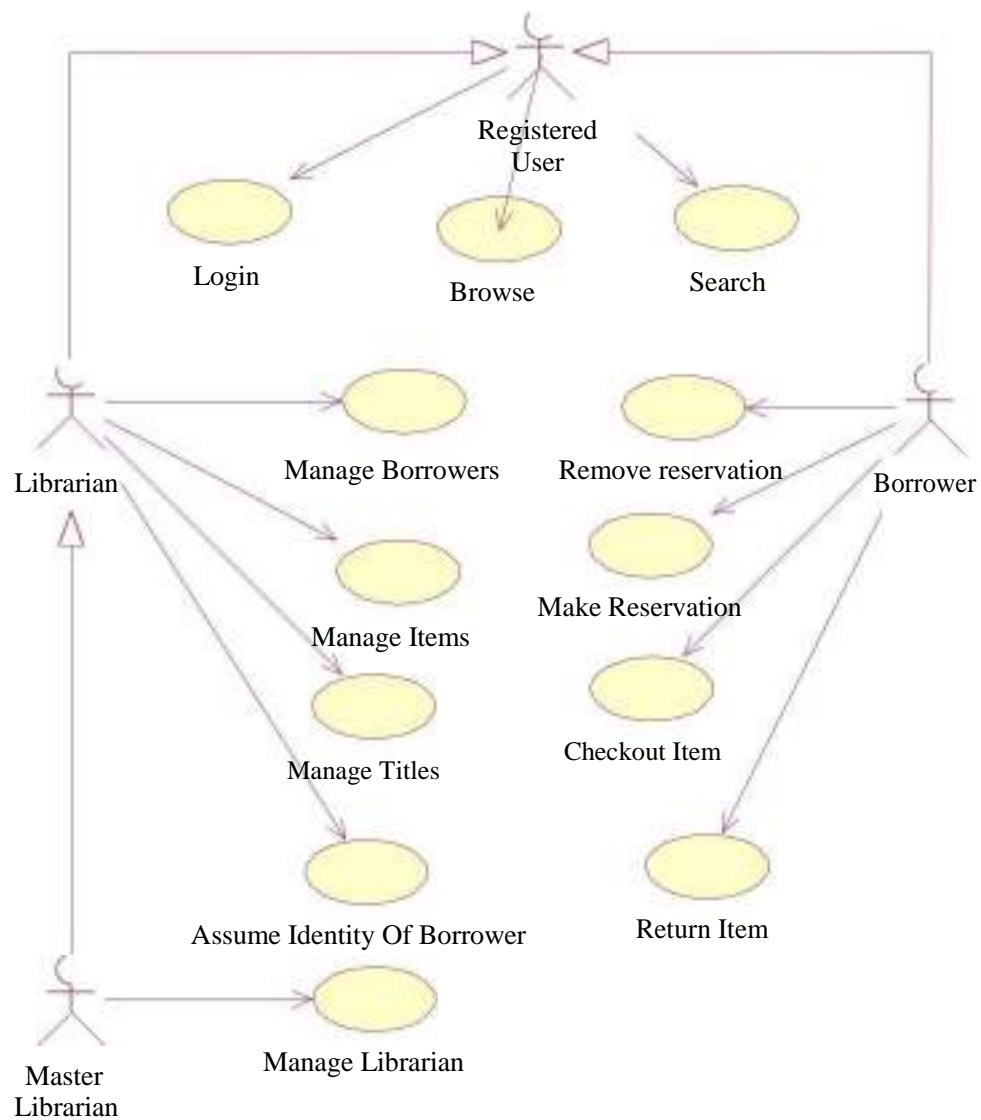
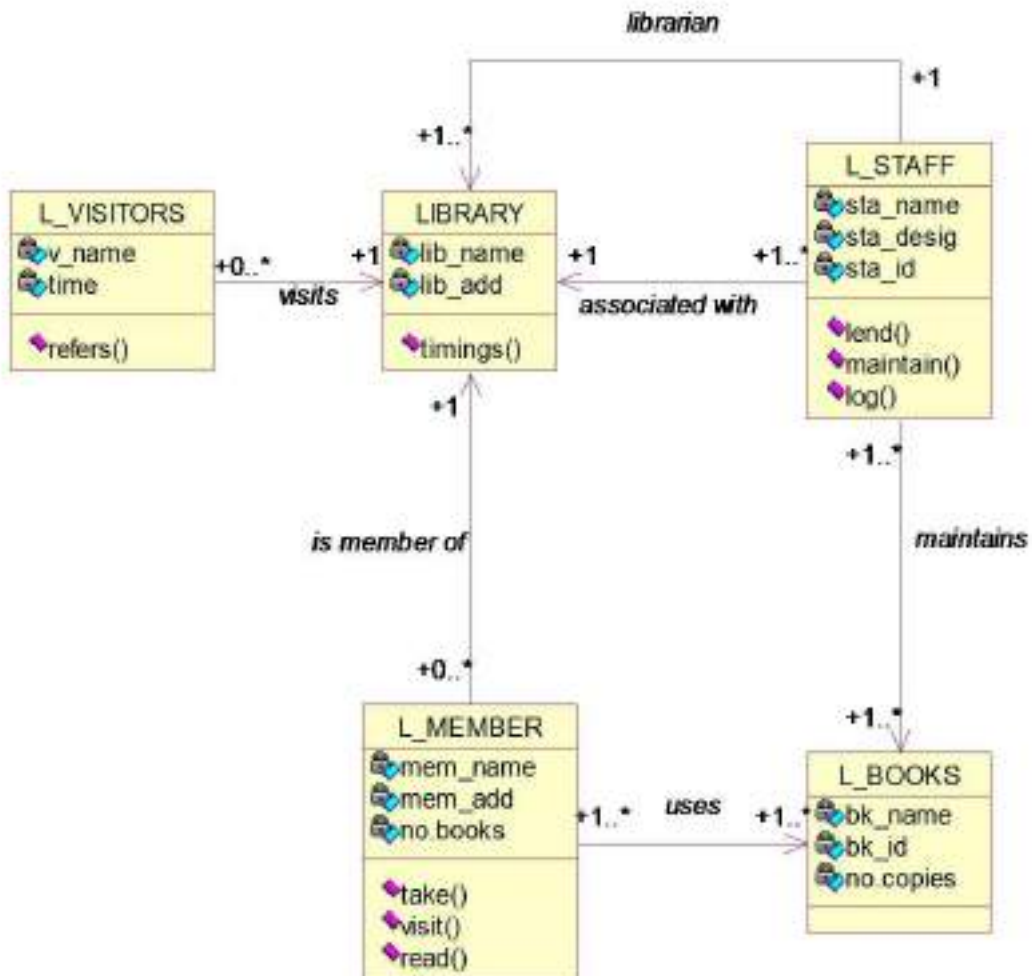
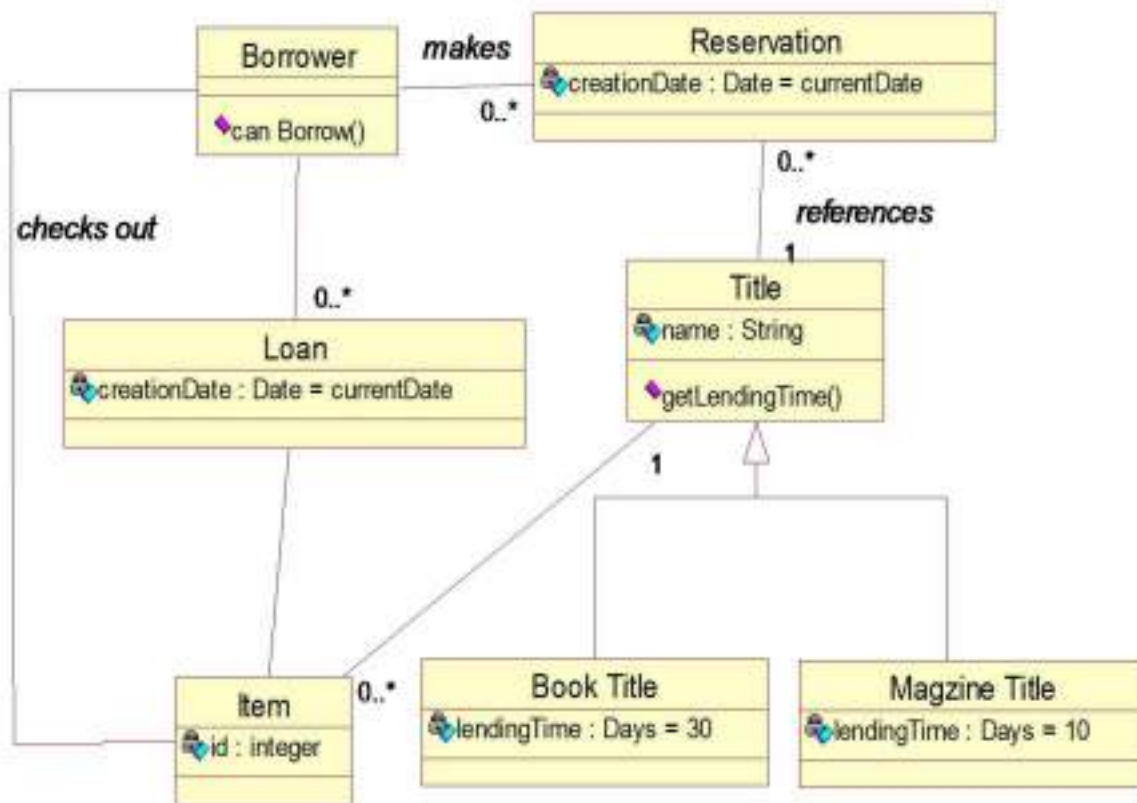


Fig: A use-case diagram for library system

Class diagram for library system:

Fig: classes for the library system





classes for the library system.

Sequence diagram for use case Return Item:

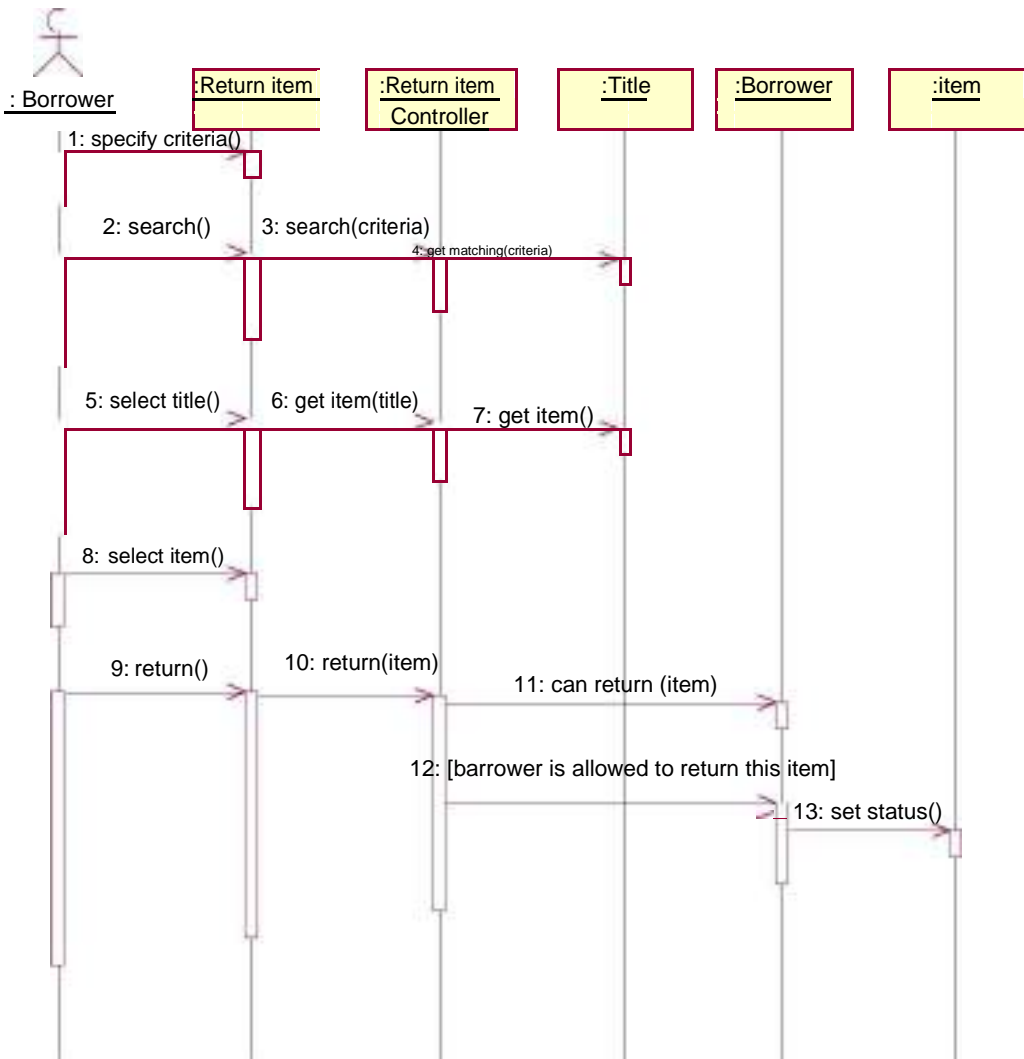
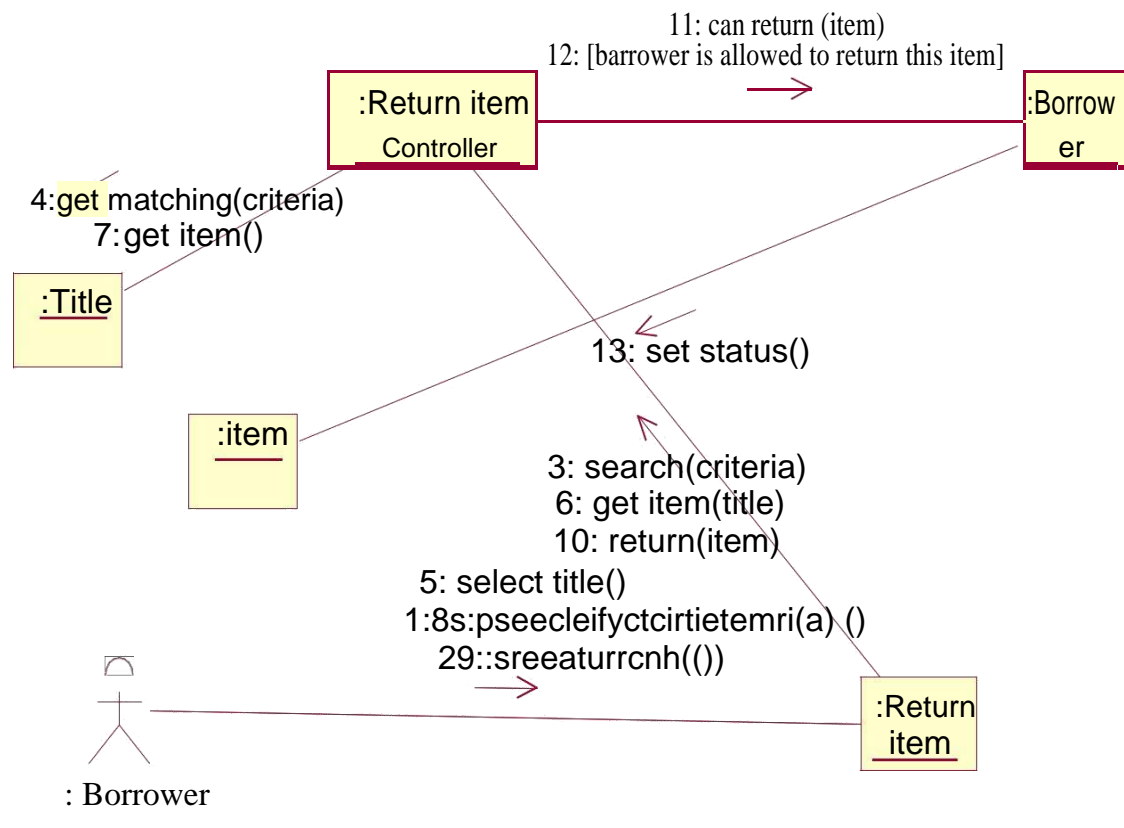
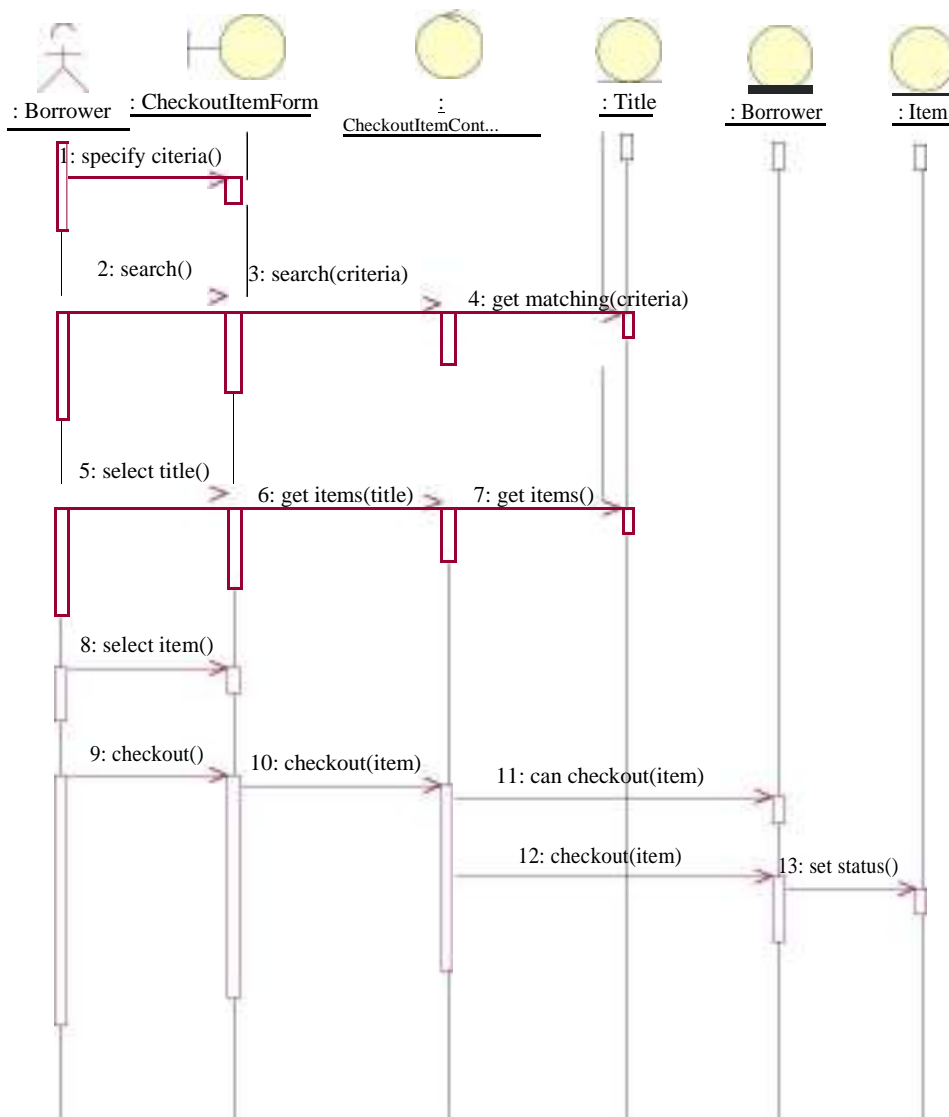


Fig: collaboration diagram for use case Return Item



Sequence diagram for use cases:

Checkout Item:



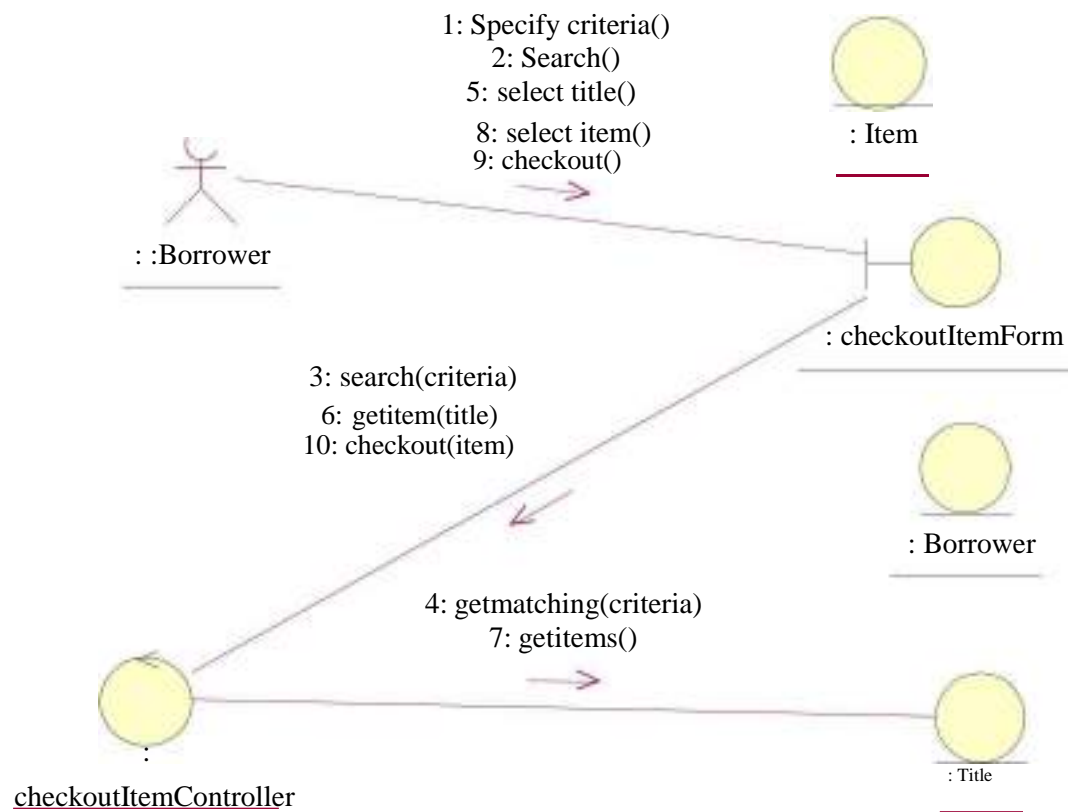
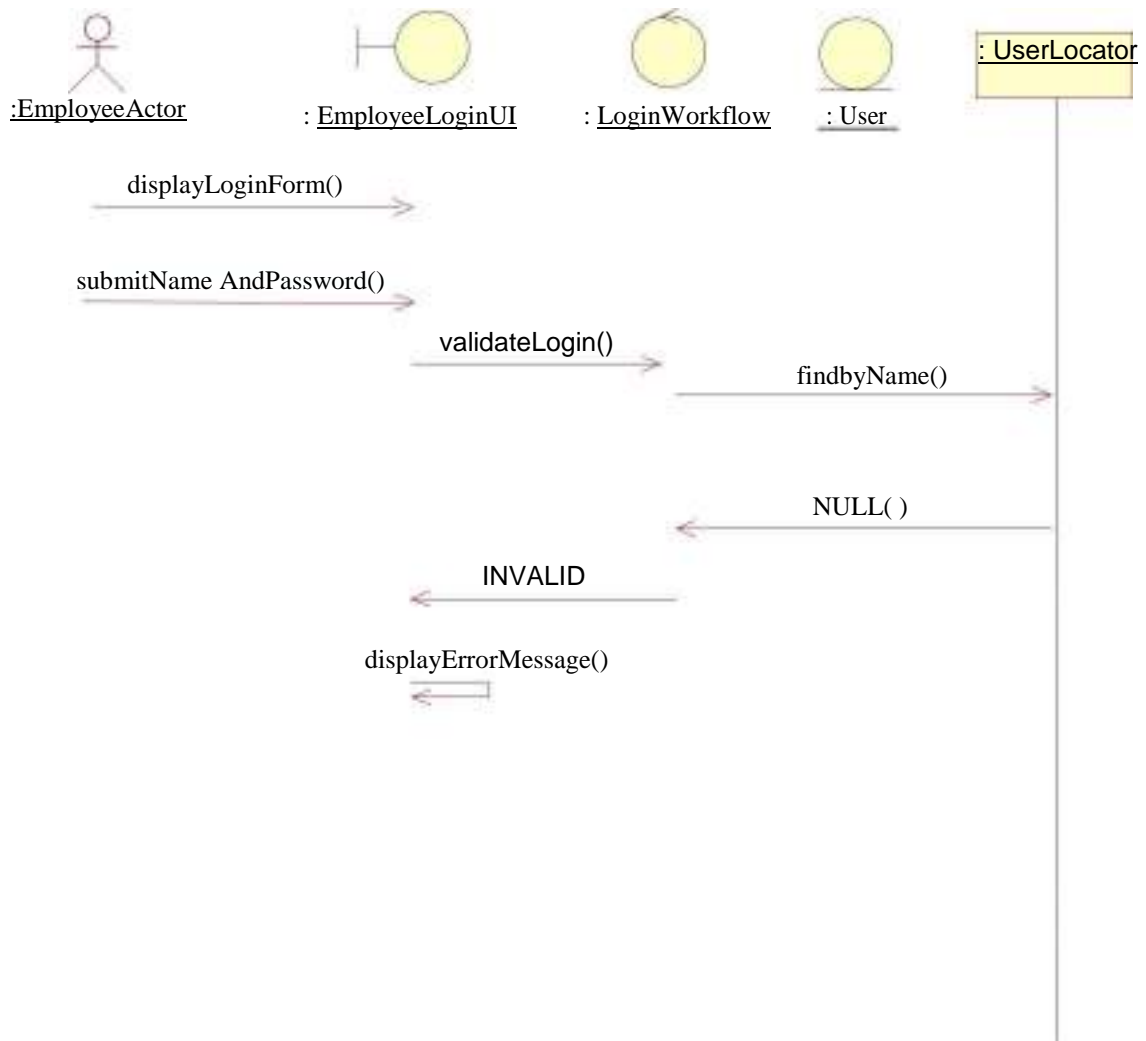


Fig: collaboration diagram for Checkout Item

Sequence diagram for use case login:



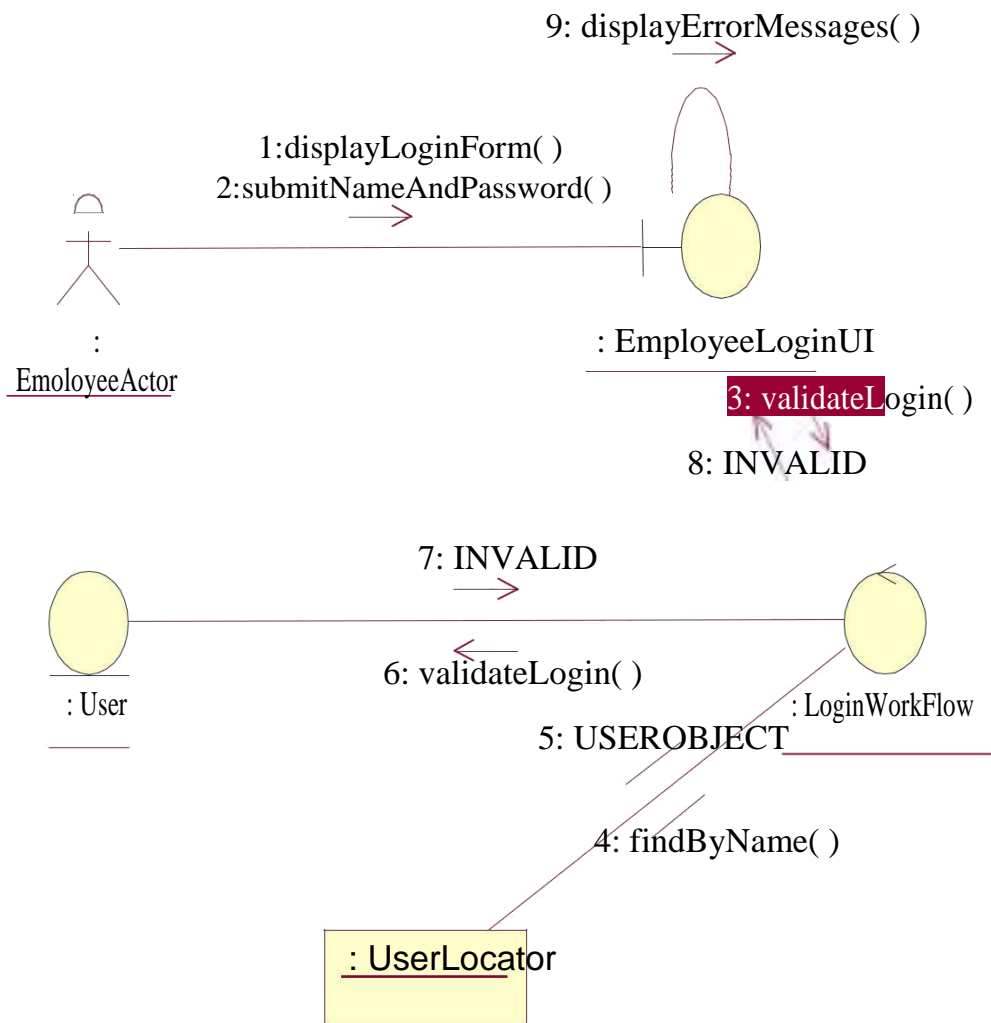
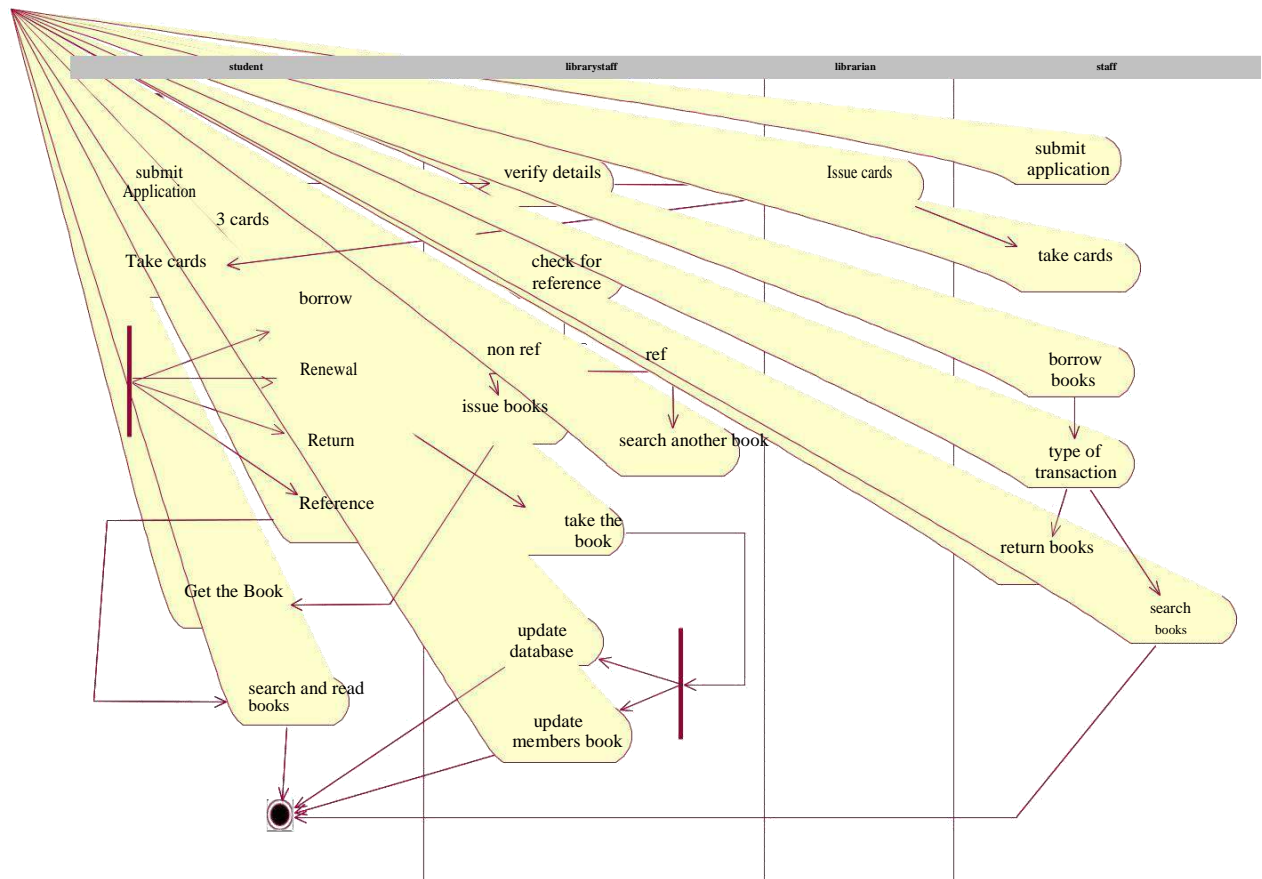
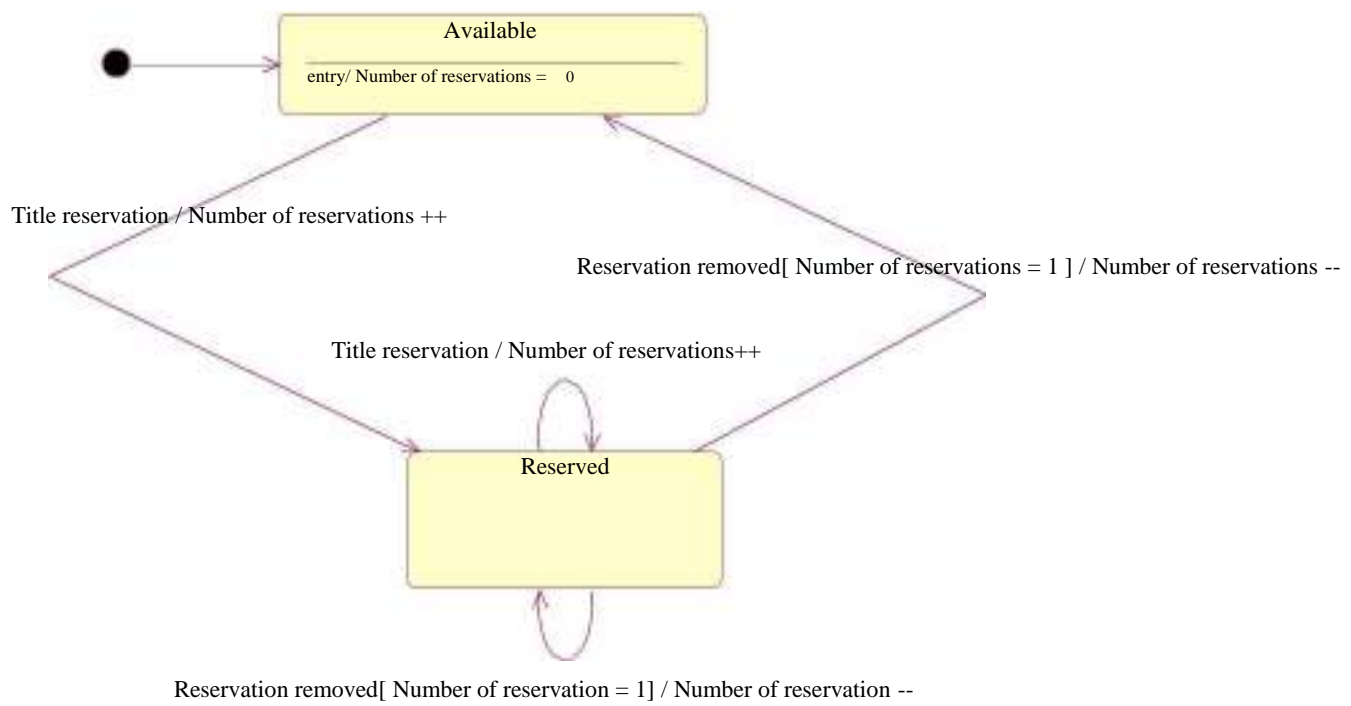


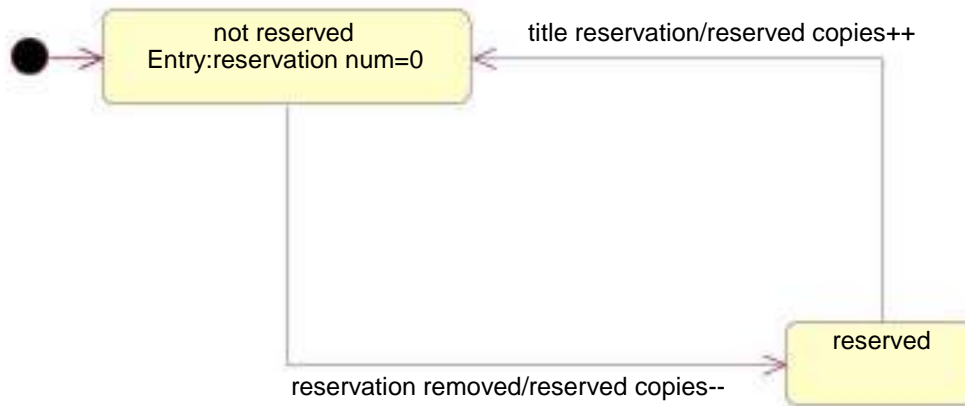
Fig: collaboration diagram for login

Activity diagram for library application:



STATE MACHINE DIAGRAM FOR THE TITLE CLASS:





STATE DIAGRAM FOR LIBRARY SYSTEM

Component diagram :

Component diagram for library application:

Figure1:

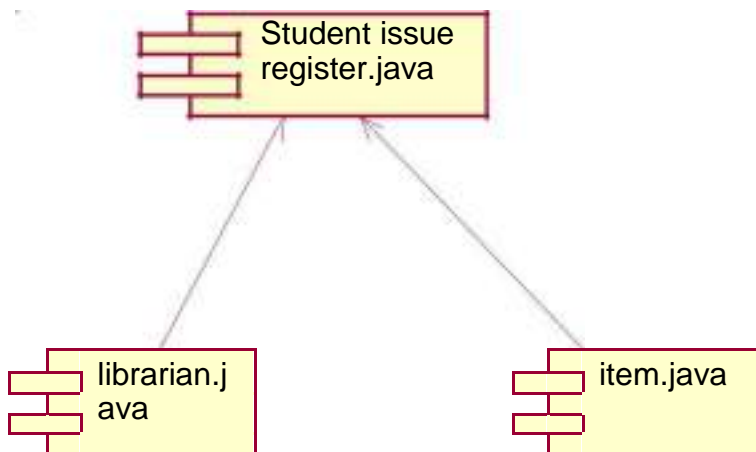
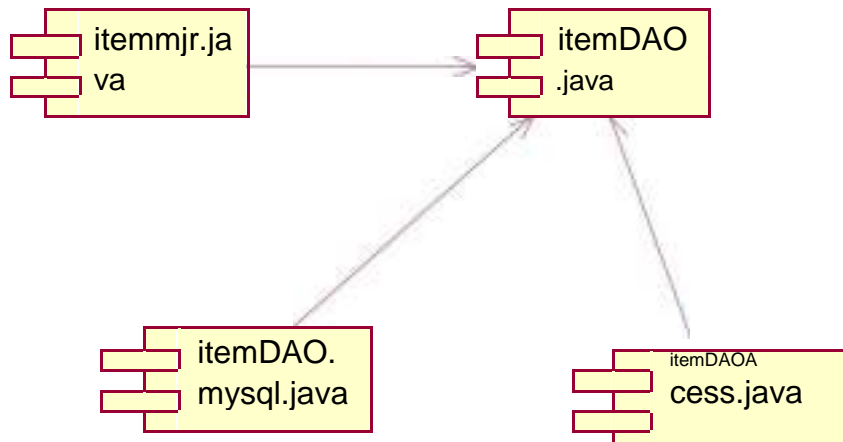


Figure2:



Deployment diagram for library applications

Deployment usually refers to transferring the project to the required end users along with the project documentation. Deployment diagrams are also essential in each application since it narrates packaged scenario of interaction following is a deployment diagram referring to unified library application scenario

Figure 1:

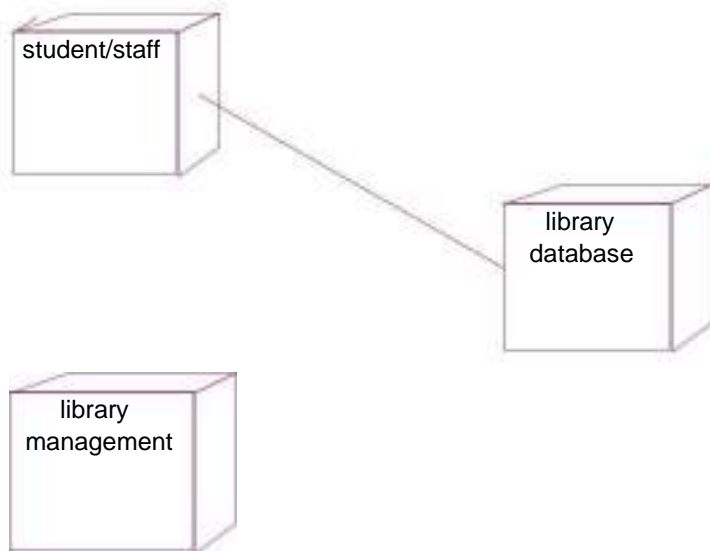
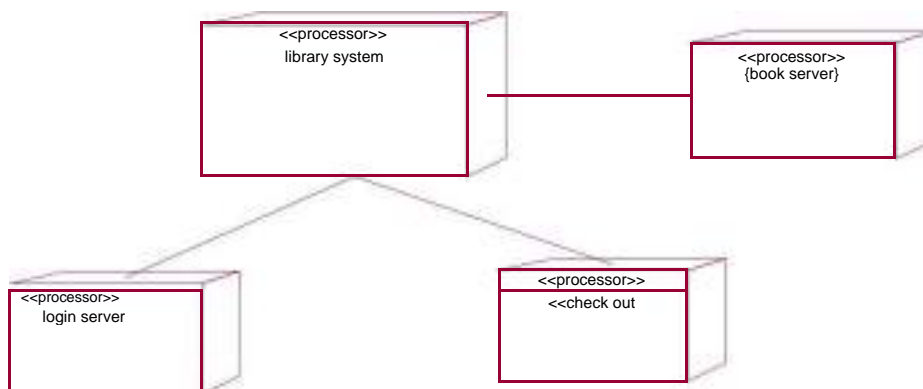


Figure 2:





MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

Permanently Affiliated to JNTUH and Approved by AICTE, New Delhi

LECTURE NOTES

ON

EMBEDDED SYSTEMS (R17A0464)

CSE III B. Tech II semester

Faculty Members

**N.SURESH
Asst.Professor
ECE Dept**

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
III Year B. Tech CSE –IISem **L T/P/D C 3 -/- 3**
(R17A0464)EMBEDDED SYSTEMS

COURSE OBJECTIVES:

For embedded systems, the course will enable the students to:

- 1) To understand the basics of microprocessors and microcontrollers architecture and its functionalities
- 2) Understand the core of an embedded system
- 3) To learn the design process of embedded system applications.
- 4) To understand the RTOS and inter-process communication.

UNIT-I: INTRODUCTION TO MICROPROCESSORS AND MICROCONTROLLERS:

8086 Microprocessor: Architecture of 8086, Register Organization, Programming Model, Memory Segmentation, Signal descriptions of 8086, Addressing modes, Instruction Set.

8051 Microcontroller: 8051 Architecture, I/O Ports, Memory Organization, Instruction set of 8051.

UNIT-II: INTRODUCTION TO EMBEDDED SYSTEMS: History of embedded systems, Classification of embedded systems based on generation and complexity, Purpose of embedded systems, Applications of embedded systems, and characteristics of embedded systems, Operational and Non-operational attributes of embedded systems.

UNIT-III: TYPICAL EMBEDDED SYSTEM: Core of the embedded system, Sensors and actuators, Onboard communication interfaces I2C, SPI, parallel interface; External communication interfaces-RS232, USB, infrared, Bluetooth, Wi-Fi, ZigBee, GPRS.

UNIT-IV: EMBEDDED FIRMWARE DESIGN AND DEVELOPMENT: Embedded firmware design approaches-super loop based approach, operating system based approach; embedded firmware development languages-assembly language based development, high level language based development.

UNIT-V EMBEDDED PROGRAMMING CONCEPTS: Data types, Structures, Modifiers, Loops and Pointers, Macros and Functions, object oriented Programming, Embedded Programming in C++ & JAVA

TEXT BOOKS:

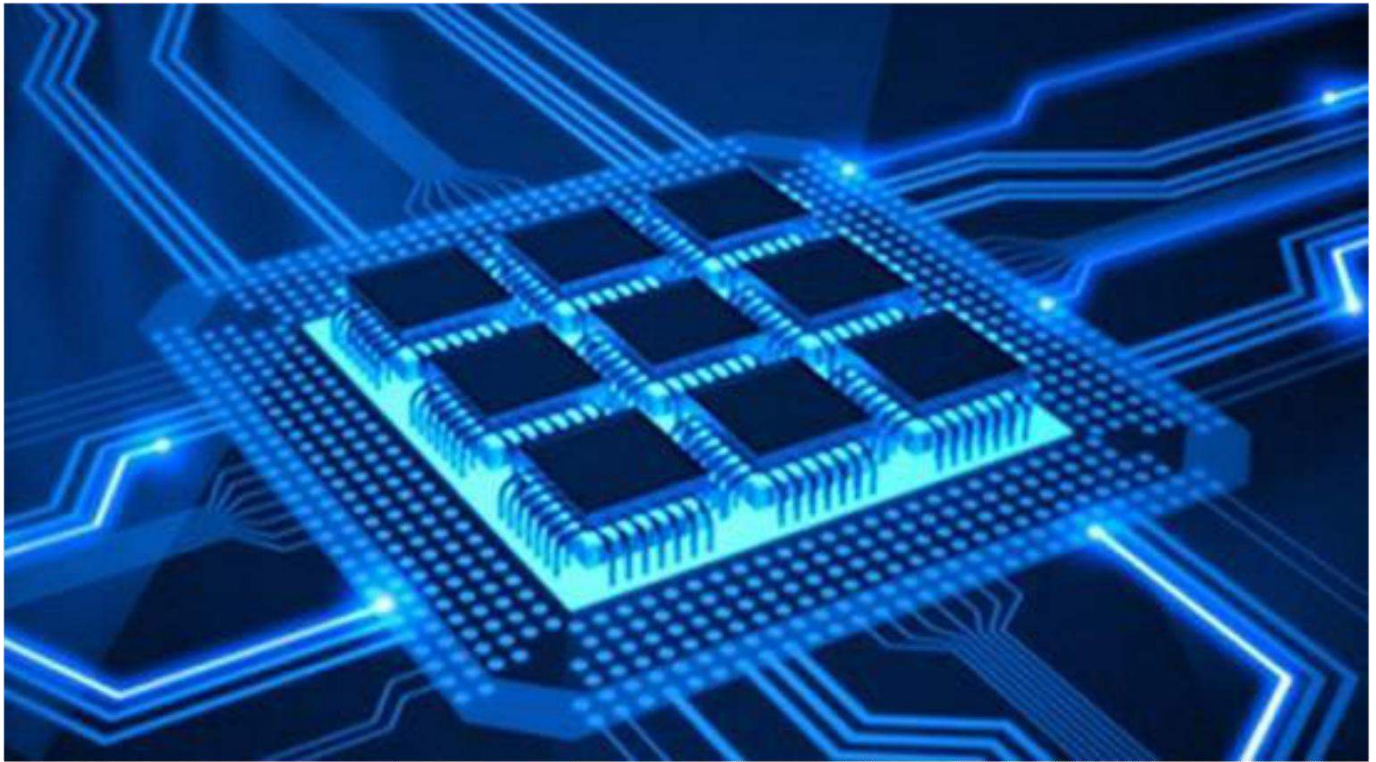
1. Embedded Systems, Raj Kamal, Second Edition TMH.
2. Kenneth. J. Ayala, The 8051 Microcontroller , 3rd Ed., Cengage Learning
3. Introduction to Embedded Systems - shibu k v, Mc Graw Hill Education.

REFERENCE BOOKS:

1. Advanced Microprocessors and Peripherals – A. K. Ray and K.M. Bhurchandi, TMH, 2nd Edition 2006
2. Embedded Systems- An integrated approach - Lyla B Das, Pearson education 2012.

Embedded Systems

III-ESS::



Unit-1

INTRODUCTION TO MICROPROCESSORS & MICROCONTROLLERS

N SURESH

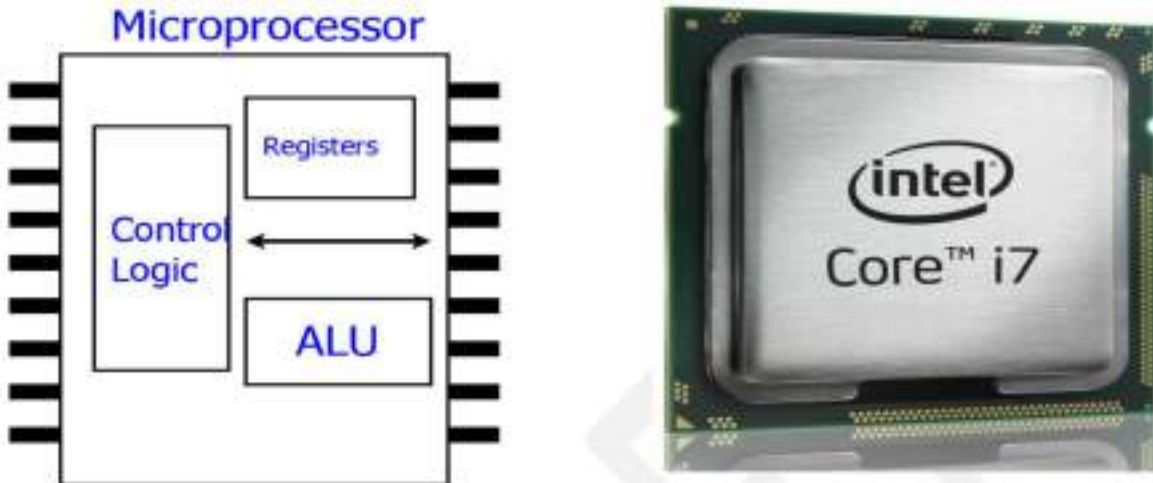
Department of ECE



**MALLA REDDY COLLEGE OF
ENGINEERING & TECHNOLOGY**

Permanently Affiliated to JNTUH and Approved by AICTE, New Delhi

Microprocessor: Microprocessor is a controlling unit of a micro-computer, fabricated on a small chip capable of performing ALU (Arithmetic Logical Unit) operations and communicating with the other devices connected to it.



Microprocessor consists of an ALU, register array, and a control unit. ALU performs arithmetic and logical operations on the data received from the memory or an input device. The control unit controls the flow of data and instructions within the computer.

Block Diagram of a Basic Microcomputer:

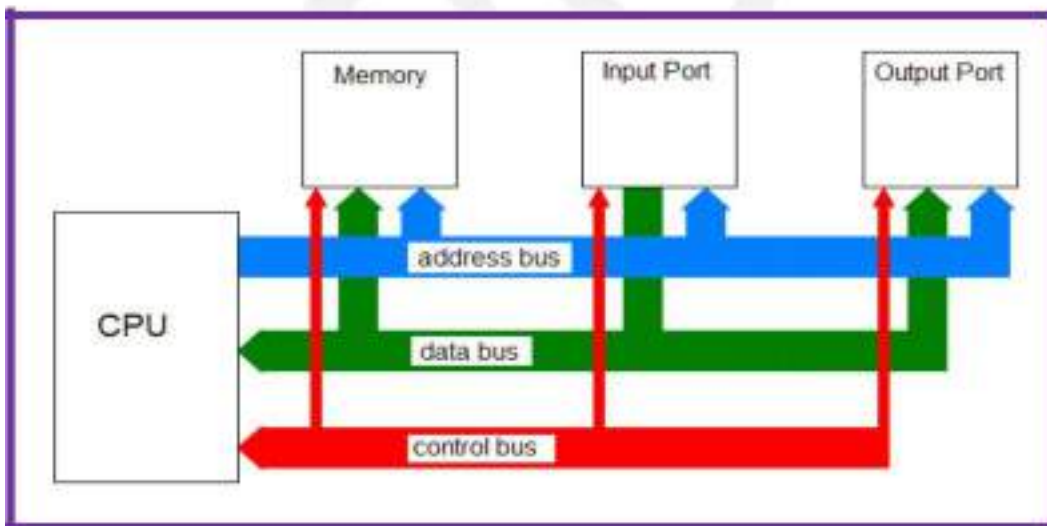


Figure 1 Block diagram of Computer

Figure shows block diagram of a simple microcomputer. The major parts are the central processing unit or CPU, memory and the input and output circuitry or Input/output. Connecting these parts are three sets of parallel line is called buses . In a microcomputer the CPU is a microprocessor and is often referred to as the

microprocessor unit (MPU). Its purpose is to decode the instruction and use them to control the activity within the system. It performs all arithmetic and logical computations.

Memory: Memory section usually consists of a mixture of RAM and ROM. It may also magnetic floppy disks, magnetic hard disks or optical disks, to store the data.

Input/output: The input/output section allows the computer to take in data from the outside world or send data to the outside world. Peripherals such as keyboards, video display terminals. Printers and modem are connected to the input/output section. These allow the user and computer to communicate with each other. The actual physical devices used to interface the computer buses to external systems are often called ports. An input/output port allows data from keyboard, an analog to digital converter (ADC) or some other source to be read into the computer under the control of the CPU. An output port is used to send data from the computer to some peripheral, such as a video display terminal, a printer or a digital to analog converter (DAC).

Central processing Unit (CPU): CPU controls the operation of the computer .In a microcomputer the CPU is a microprocessor. The CPU fetches the binary coded instructions from memory, decodes the instructions into a series of simple action and carries out these actions in sequence of steps.

CPU contains an address counter or instruction pointer register which holds the address of the next instruction or data item to be fetched from memory, general purpose register, which are used for temporary storage or binary data and circuitry, which generates the control bus signals.

Address bus: The address bus consists of 16, 20, 24 or 32 parallel lines. On these lines the CPU sends out the address of the memory locations that are to be written to or read from. The number of memory locations that the CPU can addresses is determined by the number of address lines, then it can directly address 2^n memory location. When the CPU reads data from or writes data to a port, it sends the port address on the address bus.

Ex: CPU has 16 address lines can address 2^{16} or 65536 memory locations.

Data bus: It consists of 8, 16, 32 parallel signal lines. The data bus lines are bidirectional. This means that the CPU can read, data from memory or from a port on these lines, or it can send data out to memory or to port on these lines.

Control bus: The control bus consists of 4 to 10 parallel signals lines. The CPU sends out signals on the control bus enable the outputs of addressed memory devices or port devices. Typical control bus signal are memory read, memory write, I/O read and I/O write.

Hardware, software and Firmware: hardware is the given to the physical devices and circuitry of the computer. Software refers to collection of programs written for the computer. Firmware is the term given programs stored in ROM's or in other devices which permanently keep their stored information.

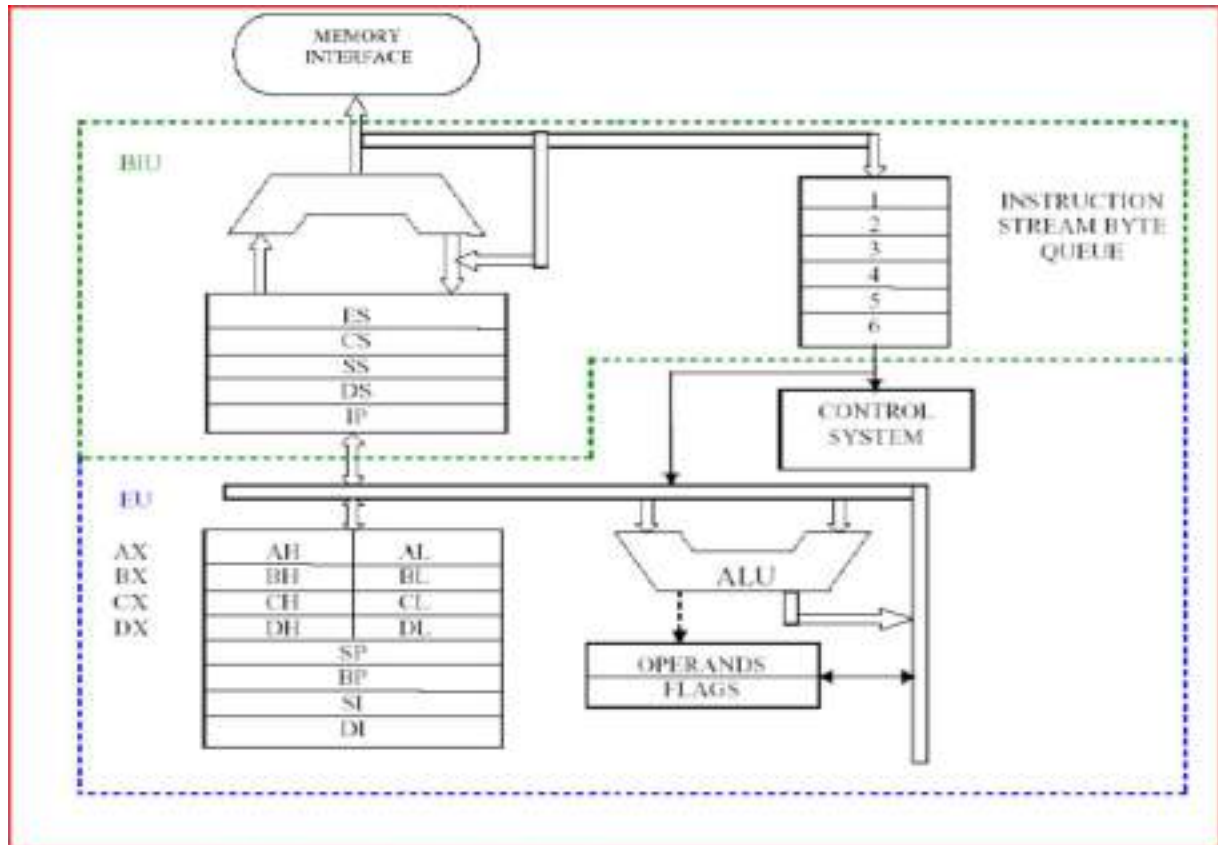
8086 Microprocessor

8086 Microprocessor is an enhanced version of 8085 Microprocessor that was designed by Intel in 1976.

Features of 8086 Microprocessor:

- It is a 16-bit Microprocessor (μp). Its ALU, internal registers work with 16-bit binary word.
- 8086 has a 20-bit address bus can access up to $2^{20} = 1\text{MB}$ memory locations.
- 8086 has a 16-bit data bus. It can read or write data to a memory/port either 16 bits or 8 bits at a time.
- It can support up to 64K I/O ports.
- It provides 14, 16-bit registers.
- Frequency range of 8086 is 6-10 MHz.
- It has multiplexed address and data bus AD0- AD15.
- It requires single phase clock with 33% duty cycle to provide internal timing.
- It can prefetch up to 6 instruction bytes from memory and queues them in order to speed up instruction execution.
- It requires +5V power supply.
- A 40 pin dual in line package.
- It has 256 vectored interrupts.
- It consists of 29,000 transistors.
- 8086 is designed to operate in two modes, Minimum mode and Maximum mode.
 - The minimum mode is selected by applying logic 1 to the $\overline{MN}/\overline{MX}$ input pin.
This is a single microprocessor configuration.
 - The maximum mode is selected by applying logic 0 to the $\overline{MN}/\overline{MX}$ input pin.
This is a multi micro processors configuration.

Architecture or Functional Block diagram of 8086:



8086 has two blocks 1. Bus Interface Unit (BIU) 2. Execution Unit (EU).

- The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating the addresses of the memory operands. The instruction bytes are transferred to the instruction queue.
- EU executes instructions from the instruction byte queue.
- Both units operate asynchronously to give the 8086 an overlapping instruction fetch and execution mechanism which is called as Pipelining. This results in efficient use of the system bus and system performance.
- BIU contains Instruction queue, Segment registers, Instruction pointer and Address adder.
- EU contains Control circuitry, Instruction decoder, ALU, Pointer and Index register, Flag register.

BIU (Bus Interface Unit):

BIU takes care of all data and addresses transfers on the buses for the EU like sending addresses, fetching instructions from the memory, reading data from the ports and the memory as well as writing data to the ports and the memory and computes the 20-bit address. EU has no direct connection with System Buses so this is possible with the BIU. EU and BIU are connected with the Internal Bus. It has the following functional parts:

Instruction queue: BIU contains the instruction queue. BIU gets up to 6 bytes of next instructions and stores them in the instruction queue. When EU executes instructions and is ready for its next instruction, then it simply reads the instruction from this instruction queue resulting in increased execution speed.

Address Adder: The BIU also contains a dedicated adder which is used to generate the 20-bit physical address that is output on the address bus. This address is formed by adding an appended 16-bit segment address and a 16-bit offset address.

Segment register: BIU has 4 segment registers, i.e. CS, DS, SS & ES. It holds the addresses of instructions and data in memory, which are used by the processor to access memory locations. It also contains 1 pointer register IP, which holds the address of the next instruction to be executed by the EU.

Instruction pointer: It is a 16-bit register used to hold the address of the next instruction to be executed.

EU (Execution Unit):

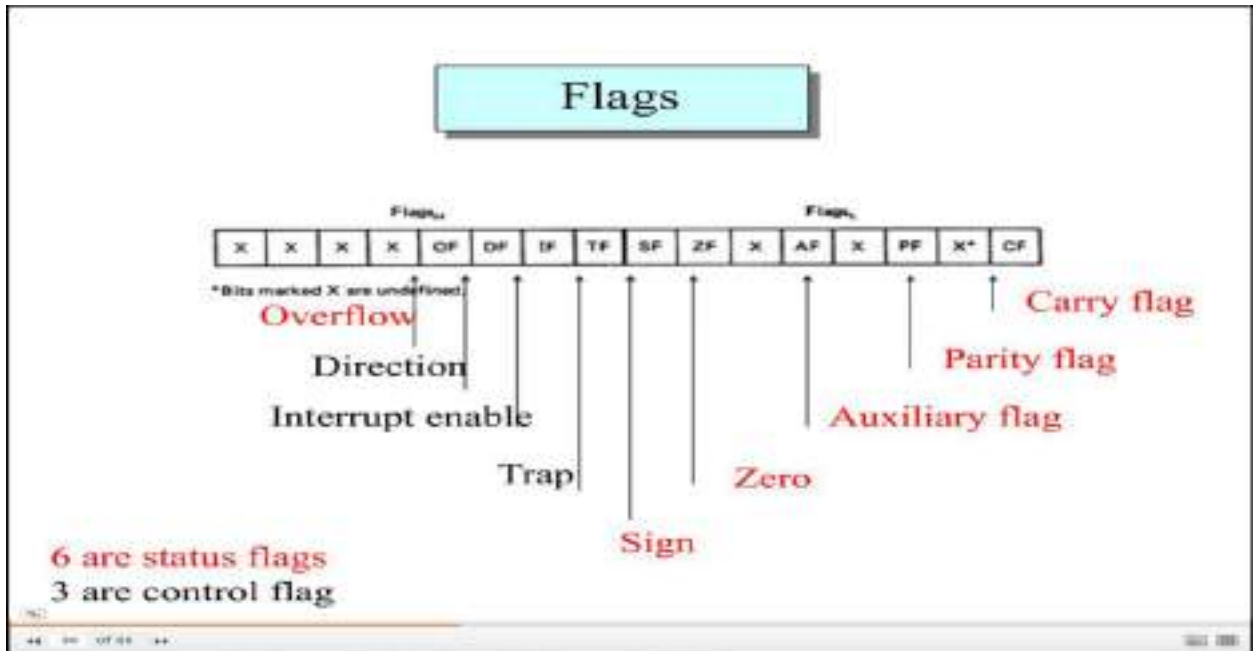
Execution unit gives instructions to BIU stating from where to fetch the data and then decode and execute those instructions. Its function is to control operations on data using the instruction decoder & ALU. EU has no direct connection with system buses as shown in the above figure, it performs operations over data through BIU. Let us now discuss the functional parts of 8086 microprocessors.

ALU: It handles all arithmetic and logical operations, like +, -, ×, /, OR, AND, NOT operations.

Flag Register: It is a 16-bit register that behaves like a flip-flop, i.e. it changes its status according to

the result stored in the accumulator. It has 9 flags and they are divided into 2 groups:

1. Conditional Flags
2. Control Flags.



Conditional Flags:

It represents the result of the last arithmetic or logical instruction executed. Following is the list of conditional flags:

- 1. Carry flag:** This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction.
- 2. Auxiliary flag:** When an operation is performed at ALU, it results in a carry/borrow from lower nibble (i.e. D0 – D3) to upper nibble (i.e. D4 – D7), then this flag is set, i.e. carry given by D3 bit to D4 is AF flag. The processor uses this flag to perform binary to BCD conversion.
- 3. Parity flag:** This flag is used to indicate the parity of the result, i.e. when the lower order 8-bits of the result contains even number of 1's, then the Parity Flag is set. For odd number of 1's, the Parity Flag is reset.

4. Zero flag: This flag is set to 1 when the result of arithmetic or logical operation is zero else it is set to 0.

5. Sign flag: This flag holds the sign of the result, i.e. when the result of the operation is negative, then the sign flag is set to 1 else set to 0.

6. Overflow flag: This flag represents the result when the system capacity is exceeded.

Control Flags:

Control flags controls the operations of the execution unit. Following is the list of control flags:

1. Trap flag: It is used for single step control and allows the user to execute one instruction at a time for debugging. If it is set, then the program can be run in a single step mode.

2. Interrupt flag: It is an interrupt enable/disable flag, i.e. used to allow/prohibit the interruption of a program. It is set to 1 for interrupt enabled condition and set to 0 for interrupt disabled condition.

3. Direction flag: It is used in string operation. As the name suggests when it is set then string bytes are accessed from the higher memory address to the lower memory address and vice-a-versa.

General Purpose Register (GPR):

There are 8 general purpose registers, i.e., AH, AL, BH, BL, CH, CL, DH, and DL. These registers can be used individually to store 8-bit data and can be used in pairs to store 16-bit data. The valid register pairs are AH and AL, BH and BL, CH and CL, and DH and DL. It is referred to the AX, BX, CX, and DX respectively.

AX register: It is also known as accumulator register. It is used to store operands for arithmetic operations.

BX register: It is used as a base register. It is used to store the starting base address of the memory area within the data segment.

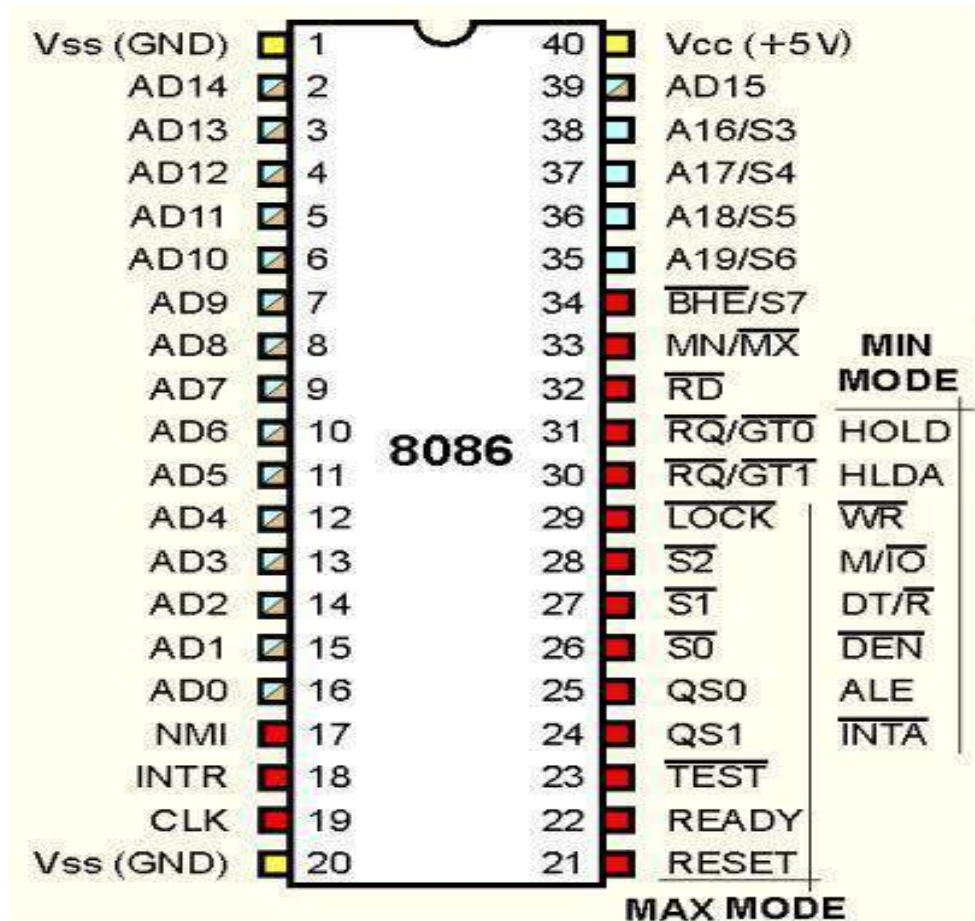
CX register: It is referred to as counter. It is used in loop instruction to store the loop counter.

DX register: This register is used to hold I/O port address for I/O instruction.

Stack pointer register: It is a 16-bit register, which holds the address from the start of the segment to the memory location, (stack top) where a word was most recently stored on the stack.

8086 Pin Diagram:

Here is the pin diagram of 8086 microprocessor:



Power supply & frequency signals:

It uses 5V DC supply at VCC pin 40, and uses ground at VSS pin 1 and 20 for its operation.

Clock signal: Clock signal is provided through Pin-19. It provides timing to the processor for operations. Its frequency is different for different versions, i.e. 5MHz, 8MHz and 10MHz.

Address/data bus: AD0-AD15. These are 16 address/data bus. AD0-AD7 carries low order byte data and AD8-AD15 carries higher order byte data. During the first clock cycle, it carries 16-bit address and after that it carries 16-bit data.

Address/status bus: A16-A19/S3-S6. These are the 4 address/status buses. During the first clock cycle, it carries 4-bit address and later it carries status signals.

S7/BHE: BHE stands for Bus High Enable. It is available at pin 34 and used to indicate the transfer of data using data bus D8-D15. This signal is low during the first clock cycle, thereafter it is active.

Read (\overline{RD}): It is available at pin 32 and is used to read signal for Read operation.

Ready: It is available at pin 32. It is an acknowledgement signal from I/O devices that data is transferred. It is an active high signal. When it is high, it indicates that the device is ready to transfer data. When it is low, it indicates wait state.

RESET: It is available at pin 21 and is used to restart the execution. It causes the processor to immediately terminate its present activity. This signal is active high for the first 4 clock cycles to RESET the microprocessor.

INTR: It is available at pin 18. It is an interrupt request signal, which is sampled during the last clock cycle of each instruction to determine if the processor considered this as an interrupt or not.

NMI: It stands for non-maskable interrupt and is available at pin 17. It is an edge triggered input, which causes an interrupt request to the microprocessor.

\overline{TEST} : This signal is like wait state and is available at pin 23. When this signal is high, then the processor has to wait for IDLE state, else the execution continues.

$\overline{MN}/\overline{MX}$: It stands for Minimum/Maximum and is available at pin 33. It indicates what mode the processor is to operate in; when it is high, it works in the minimum mode and vice-versa.

INTA: It is an interrupt acknowledgement signal and is available at pin 24. When the microprocessor receives this signal, it acknowledges the interrupt.

ALE: It stands for address enable latch and is available at pin 25. A positive pulse is generated each time the processor begins any operation. This signal indicates the availability of a valid address on the address/data lines.

DEN: It stands for Data Enable and is available at pin 26. It is used to enable Transceiver 8286. The transceiver is a device used to separate data from the address/data bus.

DT/R: It stands for Data Transmit/Receive signal and is available at pin 27. It decides the direction of data flow through the transceiver. When it is high, data is transmitted out and vice-a-versa.

M/IO: This signal is used to distinguish between memory and I/O operations. When it is high, it indicates I/O operation and when it is low indicates the memory operation. It is available at pin 28.

WR: It stands for write signal and is available at pin 29. It is used to write the data into the memory or the output device depending on the status of M/IO signal.

HLDA: It stands for Hold Acknowledgement signal and is available at pin 30. This signal acknowledges the HOLD signal.

HOLD: This signal indicates to the processor that external devices are requesting to access the address/data buses. It is available at pin 31.

QS1& QS0: These are queue status signals and are available at pin 24 and 25. These signals provide the status of instruction queue. Their conditions are shown in the following table:

QS ₁	QS ₀	Status
0	0	No operation
0	1	1 st byte of opcode from queue
1	0	Empty queue
1	1	Subsequent byte from queue

$\overline{S_0}, \overline{S_1}, \overline{S_2}$: These are the status signals that provide the status of operation, which is used by the Bus Controller 8288 to generate memory & I/O control signals. These are available at pin 26, 27, and 28. Following is the table showing their status:

$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Status
0	0	0	Interrupt Acknowledge
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt
1	0	0	Opcode Fetch
1	0	1	Memory Read
1	1	0	Memory Write
1	1	1	Passive

LOCK: When this signal is active, it indicates to the other processors not to ask the CPU to leave the system bus. It is activated using the LOCK prefix on any instruction and is available at pin 29.

RQ/GT1& RQ/GT0 : These are the Request/Grant signals used by the other processors requesting the CPU to release the system bus. When the signal is received by CPU, then it sends acknowledgment. RQ/GT0 has a higher priority than RQ/GT1.

Programming model of 8086: The programming model of a processor deals with internal registers, status and control flags, number of address lines, number of data lines and the input/output port addresses which are needed by the programmer to write the programs.

How can a 20-bit address be obtained, if there are only 16-bit registers? However, the largest register is only 16 bits (64k); so physical addresses have to be calculated.

These calculations are done in hardware within the microprocessor. The 16-bit contents of segment register gives the starting/ base address of particular segment. To address a specific memory location within a segment we need an offset address. The offset address is also 16-bit wide and it is provided by one of the associated pointer or index register.

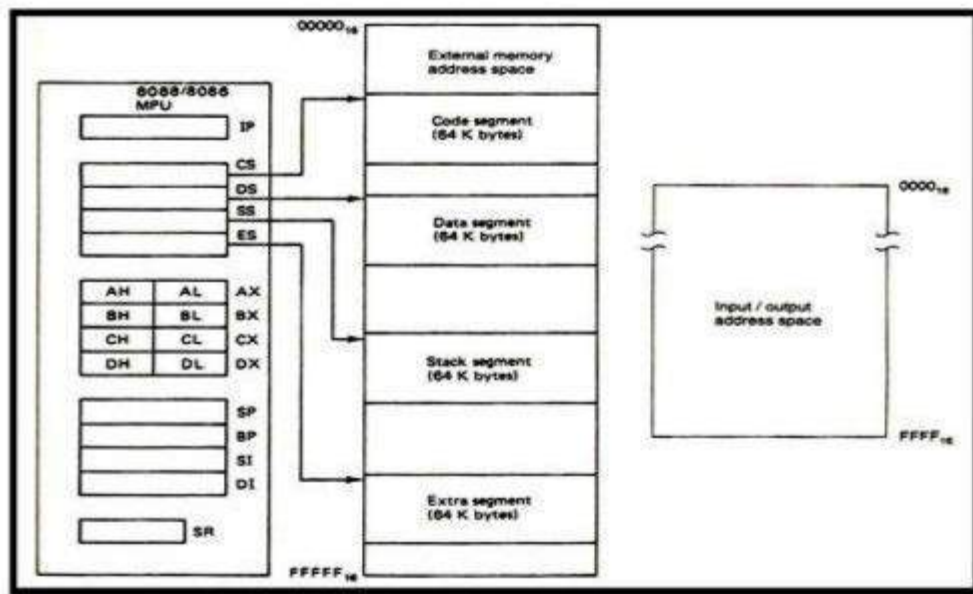
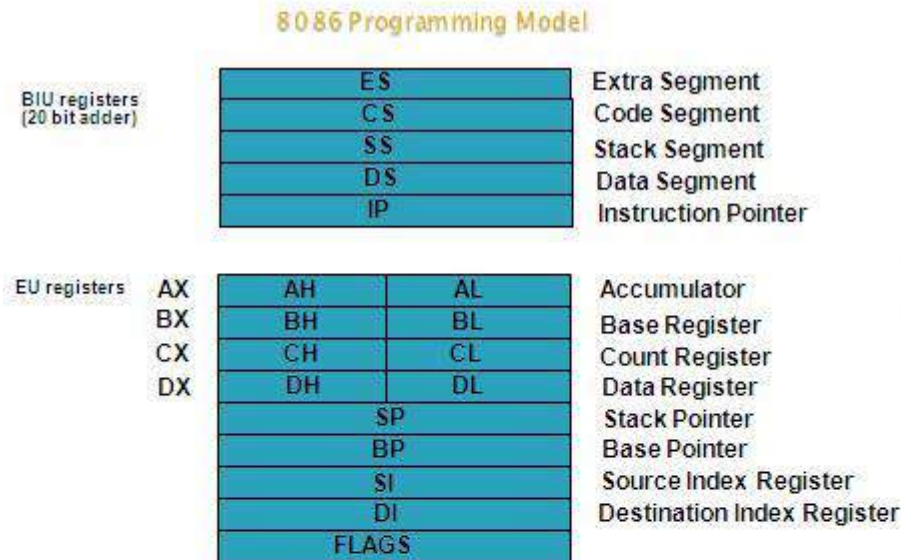


Figure: Software model of 8086 microprocessor

To be able to program a microprocessor, one does not need to know all of its hardware architectural features. What is important to the programmer is being aware of the various registers within the device and to understand their purpose, functions, operating capabilities, and limitations.

The above figure illustrates the software architecture of the 8086 microprocessor. From this diagram, we see that it includes fourteen 16-bit internal registers: the instruction pointer (IP), four data registers (AX, BX, CX, and DX), two pointer registers (BP and SP), two index registers (SI and DI), four segment registers (CS, DS, SS, and ES) and status register (SR), with nine of its bits implemented as status and control flags.



The point to note is that the beginning segment address must begin at an address divisible by 16. Also note that the four segments need not be defined separately. It is allowable for all four segments to completely overlap ($CS = DS = ES = SS$).

REGISTER ORGANISATION:

A register is a very small amount of fast memory that is built in the CPU (or Processor) in order to speed up the operation. Register is very fast and efficient than the other memories like RAM, ROM, external memory etc., That's why the registers occupied the top position in memory hierarchy model.

The 8086 microprocessor has a total of fourteen registers that are accessible to the programmer. All these registers are 16-bit in size. The registers of 8086 are categorized into 5 different groups.

- a) General registers
- b) Index registers
- c) Segment registers
- d) Pointer registers
- e) Status Register

S.No	Type	Register width	Name of the Registers
1	General purpose Registers(4)	16-bit	AX,BX,CX,DX
		8-bit	AL,AH,BL,BH,CL,CH,DL,DH
2	Pointer Registers	16-bit	Stack Pointer(SP) Base Pointer(BP)
3	Index Registers	16-bit	Source Index(SI) Destination Index(DI)
4	Segment Registers	16-bit	Code Segment(CS) Data Segment(DS) Stack Segment(SS) Extra Segment(ES)
5	Flag (PSW)	16-bit	Flag Register

a) General purpose Registers:

All general registers of the 8086 microprocessor can be used for arithmetic and logic operations. These all general registers can be used as either 8-bit or 16-bit registers. The general registers are:

AX (Accumulator): AX is used as 16-bit accumulator. The lower 8-bits of AX are designated to use as AL and higher 8-bits as AH. AL can be used as an 8-bit accumulator for 8-bit operation.

This Accumulator used in arithmetic, logic and data transfer operations. For manipulation and division operations, one of the numbers must be placed in AX or AL.

BX (Base Register): BX is a 16 bit register, but BL indicates the lower 8-bits of BX and BH indicates the higher 8-bits of BX. The register BX is used as address register to form physical address in case of certain addressing modes (ex: indexed and register indirect).

CX (Count Register): The register CX is used default counter in case of string and loop instructions. Count register can also be used as a counter in string manipulation and shift/rotate instruction.

		15	8	7	0	
Accumulator	AX	AH		AL		Multiply, divide, I/O
Base	BX	BH		BL		Pointer to base addresss (data)
Count	CX	CH		CL		Count for loops, shifts
Data	DX	DH		DL		Multiply, divide, I/O

General Purpose Registers

DX (Data Register): DX register is a general purpose register which may be used as an implicit operand or destination in case of a few instructions. Data register can also be used as a port number in I/O operations.

Segment Register:

The 8086 architecture uses the concept of segmented memory. 8086 can able to access a memory capacity of up to 1 megabyte. This 1 megabyte of memory is divided into 16 logical segments. Each segment contains 64 Kbytes of memory.

Code Segment	CS	
Data Segment	DS	
Stack Segment	SS	
Extra Segment	ES	

Segment Registers

This memory segmentation concept will discuss later in this document. There are four segment registers to access this 1 megabyte of memory. The segment registers of 8086 are:

CS(Code Segment): Code segment (CS) is a 16-bit register that is used for addressing memory location in the code segment of the memory (64Kb), where the executable program is stored. CS register cannot be changed directly. The CS register is automatically updated during far jump, far call and far return instructions.

Stack segment (SS): Stack Segment (SS) is a 16-bit register that used for addressing stack segment of the memory (64kb) where stack data is stored. SS register can be changed directly using POP instruction.

Data segment (DS): Data Segment (DS) is a 16-bit register that points the data segment of the memory (64kb) where the program data is stored. DS register can be changed directly using POP and LDS instructions.

Extra segment (ES): Extra Segment (ES) is a 16-bit register that also points the data segment of the memory (64kb) where the program data is stored. ES register can be changed directly using POP and LES instructions.

b) Index Registers:

The index registers can be used for arithmetic operations but their use is usually concerned with the memory addressing modes of the 8086 microprocessor (indexed, base indexed and relative base indexed addressing modes).

The index registers are particularly useful for string manipulation.

SI (Source Index):

SI is a 16-bit register. This register is used to store the offset of source data in data segment. In other words the Source Index Register is used to point the memory locations in the data segment.

DI (Destination Index):

DI is a 16-bit register. This is destination index register performs the same function as SI. There is a class of instructions called string operations that use DI to access the memory locations in Data or Extra Segment.

c) Pointer Registers:

Pointer Registers contains the offset of data(variables, labels) and instructions from their base segments (default segments). 8086 microprocessor contains three pointer registers.



SP (Stack Pointer): Stack Pointer register points the program stack that means SP stores the base address of the Stack Segment.

BP (Base Pointer): Base Pointer register also points the same stack segment. Unlike SP, we can use BP to access data in the other segments also.

IP (Instruction Pointer):

The Instruction Pointer is a register that holds the address of the next instruction to be fetched from memory. It contains the offset of the next word of instruction code instead of its actual address

d) Status Register or Flag Register:

The status register also called as flag register. The 8086 flag register contents indicate the results of computation in the ALU. It also contains some flag bits to control the CPU operations.

It is a 16 bit register which contains six status flags and three control flags. So, only nine bits of the 16 bit register are defined and the remaining seven bits are undefined. Normally this status flag bits indicate the status of the ALU after the arithmetic or logical operations. Each bit of the status register is a flip/flop. The Flag register contains Carry flag, Parity flag, Auxiliary flag Zero flag, Sign flag, Trap flag, Interrupt flag, Direction

flag and overflow flag as shown in the diagram. The CF,PF,AF,ZF,SF,OF are the status flags and the TF,IF and CF are the control flags.

X	X	X	X	OF	DF	IF	TF	SF	ZF	X	AF	X	PF	X	CF
---	---	---	---	----	----	----	----	----	----	---	----	---	----	---	----

CF- Carry Flag: This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

PF - Parity Flag : This flag is set to 1, if the lower byte of the result contains even number of 1's else (for odd number of 1s) set to zero.

AF- Auxilary Carry Flag: This is set, if there is a carry from the lowest nibble, i.e, bit three during addition, or borrow for the lowest nibble, i.e, bit three, during subtraction.

ZF- Zero Flag: This flag is set, if the result of the computation or comparison performed by the previous instruction is zero

SF- Sign Flag : This flag is set, when the result of any computation is negative

TF - Tarp Flag: If this flag is set, the processor enters the single step execution mode.

IF- Interrupt Flag: If this flag is set, the maskable interrupt INTR of 8086 is enabled and if it is zero ,the interrupt is disabled. It can be set by using the STI instruction and can be cleared by executing CLI instruction.

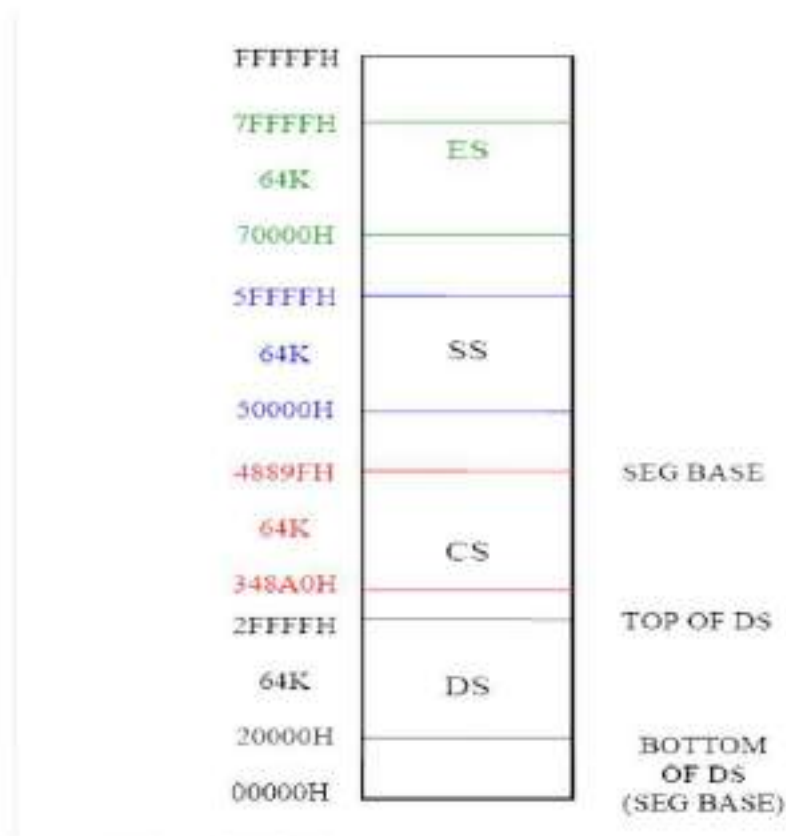
DF- Direction Flag: This is used by string manipulation instructions. If this flag bit is „0“, the string is processed beginning from the lowest address to the highest address, i.e., auto incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto decrementing mode.

OF- Over flow Flag: This flag is set, if an overflow occurs, i.e, if the result of a signed operation is large enough to accommodate in a destination register. The result is of

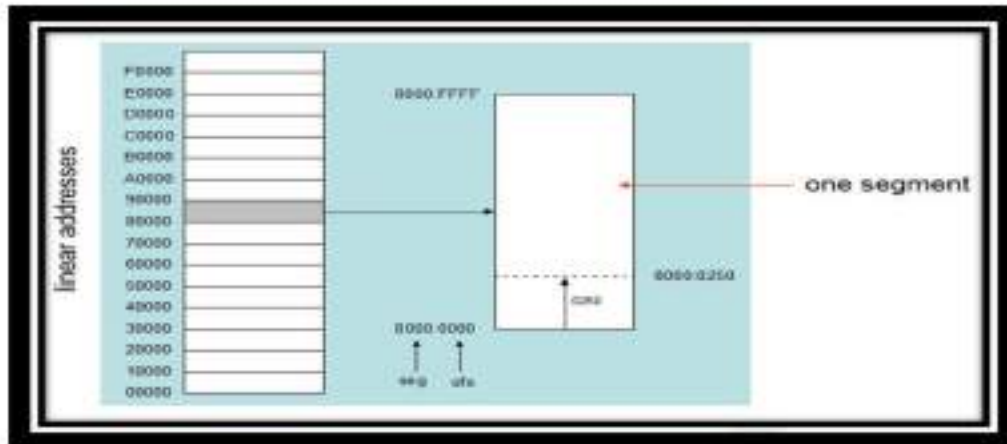
more than 7-bits in size in case of 8-bit signed operation and more than 15-bits in size in case of 16-bit sign operations, then the overflow will be set.

MEMORY SEGMENTATION:

- It is the process in which the main memory of computer is divided into different segments and each segment has its own base address.



- Segmentation is used to increase the execution speed of computer system so that processor can able to fetch and execute the data from memory easily and fastly.
- The size of address bus of 8086 is 20 and is able to address 1 Mbytes of physical memory.
- The complete 1 Mbytes memory can be divided into 16 segments, each of 64 Kbytes size.
- The addresses of the segment may be assigned as 0000H to F000H respectively.
- The offset values are from 0000H to FFFFFH.



Types of Segmentation:

1. Overlapping Segment:

- A segment starts at a particular address and its maximum size can go up to 64 Kbytes. But if another segment starts along this 64 Kbytes location of the first segment, the two segments are said to be overlapping segment.
- The area of memory from the start of the second segment to the possible end of the first segment is called as overlapped segment
- **Non Overlapped Segment:** A segment starts at a particular address and its maximum size can go up to 64 Kbytes. But if another segment starts before this 64 Kbytes location of the first segment, the two segments are said to be Non-overlapping segment.

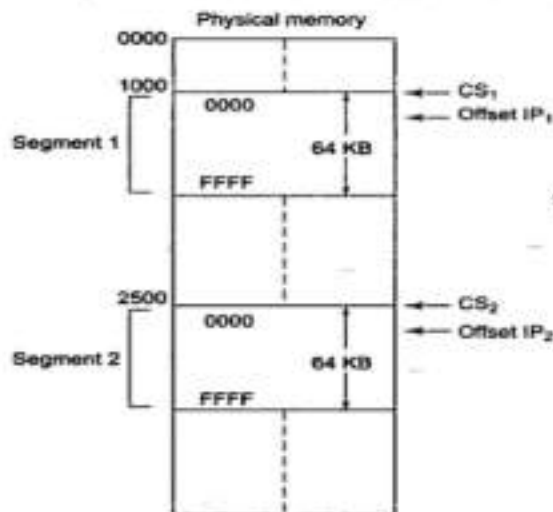


Fig. 1.3(a) Non-overlapping Segments

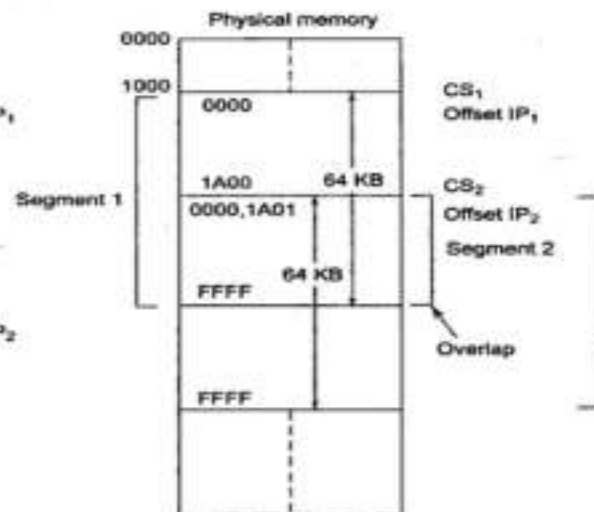


Fig. 1.3(b) Overlapping Segments

Advantages of Segmented memory:

- Allows the memory capacity to be 1MB although the actual addresses to be handled are of 16 bit size.
- Allows the placing of code, data and stack portions of the same program in different parts (segments) of the memory, for data and code protection.
- Permits a program and/or its data to be put into different areas of memory each time program is executed, i.e. provision for relocation may be done.
- The segment registers are used to allow the instruction, data or stack portion of a program to be more than 64Kbytes long. The above can be achieved by using more than one code, data or stack segments.

Addressing Modes of 8086:

Addressing mode indicates a way of locating data or operands. Depending up on the data type used in the instruction and the memory addressing modes, any instruction may belong to one or more addressing modes or same instruction may not belong to any of the addressing modes.

The addressing mode describes the types of operands and the way they are accessed for executing an instruction. According to the flow of instruction execution, the instructions may be categorized as

1. Sequential control flow instructions and**2. Control transfer instructions.**

Sequential control flow instructions are the instructions in which after execution of current instruction, control will be transferred to the next instruction appearing immediately after it (in the sequence) in the program. For example the arithmetic, logic, data transfer and processor control instructions are Sequential control flow instructions.

The control transfer instructions on the other hand transfer control to some predefined address or the address somehow specified in the instruction, after their execution. For example INT, CALL, RET & JUMP instructions fall under this category.

The addressing modes for Sequential and control flow instructions are explained as follows.

1. Immediate addressing mode: In this type of addressing, immediate data is a part of instruction, and appears in the form of successive byte or bytes.

In the above example, 0005H is the immediate data. The immediate data may be 8-bit or 16-bit in size.

2. Direct addressing mode: In the direct addressing mode, a 16-bit memory address (offset) directly specified in the instruction as a part of it.

3. Register addressing mode: In the register addressing mode, the data is stored in a register and it is referred using the particular register. All the registers, except IP, may be used in this mode.

4. Register indirect addressing mode: Sometimes, the address of the memory location which contains data or operands is determined in an indirect way, using the offset registers. The mode of addressing is known as register indirect mode.

In this addressing mode, the offset address of data is in either BX or SI or DI Register. The default segment is either DS or ES.

5. Indexed addressing mode: In this addressing mode, offset of the operand is stored one of the index registers. DS & ES are the default segments for index registers SI & DI respectively.

6. Register relative addressing mode: In this addressing mode, the data is available at an effective address formed by adding an 8-bit or 16-bit displacement with the content of any one of the register BX, BP, SI & DI in the default (either in DS & ES) segment.

7. Based indexed addressing mode: The effective address of data is formed in this addressing mode, by adding content of a base register (any one of BX or BP) to the

content of an index register (any one of SI or DI). The default segment register may be ES or DS.

Example: MOV AX, [BX][SI]

8. Relative based indexed: The effective address is formed by adding an 8 or 16-bit displacement with the sum of contents of any of the base registers (BX or BP) and any one of the index registers, in a default segment.

Example: MOV AX, 50H [BX] [SI]

For the control transfer instructions, the addressing modes depend upon whether the destination location is within the same segment or in a different one. It also depends upon the method of passing the destination address to the processor. Basically, there are two addressing modes for the control transfer instructions, viz. intersegment and intrasegment addressing modes.

If the location to which the control is to be transferred lies in a different segment other than the current one, the mode is called intersegment mode. If the destination location lies in the same segment, the mode is called intrasegment mode.

Addressing Modes for control transfer instructions:

9. Intersegment

- a) Intersegment direct b) Intersegment indirect

10 Intrasegment

- a) Intrasegment direct b) Intrasegment indirect

9. (a) Intersegment direct: In this mode, the address to which the control is to be transferred is in a different segment. This addressing mode provides a means of branching from one code segment to another code segment. Here, the CS and IP of the destination address are specified directly in the instruction.

9. (b) Intersegment indirect: In this mode, the address to which the control is to be transferred lies in a different segment and it is passed to the instruction indirectly, i.e. contents of a memory block containing four bytes, i.e. IP(LSB), IP(MSB), CS(LSB) and

CS(MSB) sequentially. The starting address of the memory block may be referred using any of the addressing modes, except immediate mode. **Example:** JMP [2000H].

Jump to an address in the other segment specified at effective address 2000H in DS.

10.(a) Intrasegment direct mode: In this mode, the address to which the control is to be transferred lies in the same segment in which the control transfers instruction lies and appears directly in the instruction as an immediate displacement value. In this addressing mode, the displacement is computed relative to the content of the instruction pointer.

The effective address to which the control will be transferred is given by the sum of 8 or 16 bit displacement and current content of IP. In case of jump instruction, if the signed displacement (d) is of 8-bits (i.e. $-128 < d < +127$), it is a short jump and if it is of 16 bits (i.e. $-32768 < d < +32767$), it is termed as long jump. **Example:** JMP SHORT LABEL.

10.(b) Intrasegment indirect mode: In this mode, the displacement to which the control is to be transferred is in the same segment in which the control transfer instruction lies, but it is passed to the instruction indirectly. Here, the branch address is found as the content of a register or a memory location. This addressing mode may be used in unconditional branch instructions.

Example: JMP [BX]; Jump to effective address stored in BX.

Instruction set of 8086

The Instruction set of 8086 microprocessor is classified into 8, they are:-

- **Data transfer instructions**
- **Arithmetic instructions**
- **Logical instructions**
- **Shift / rotate instructions**
- **Flag manipulation instructions**
- **Program control transfer instructions**
- **Machine Control Instructions**
- **String instructions**

Data Transfer Instructions:

Data transfer instruction, as the name suggests is for the transfer of data from memory to internal register, from internal register to memory, from one register to another register, from input port to internal register, from internal register to output port etc

1. MOV instruction:

It is a general purpose instruction to transfer byte or word from register to register, memory to register, register to memory or with immediate addressing.

Syntax:

MOV destination, source

Here the source and destination needs to be of the same size, that is both 8 bit and both 16 bit.

MOV instruction does not affect any flags.

Example:-

MOV BX, 00F2H ; load the immediate number 00F2H in BX register

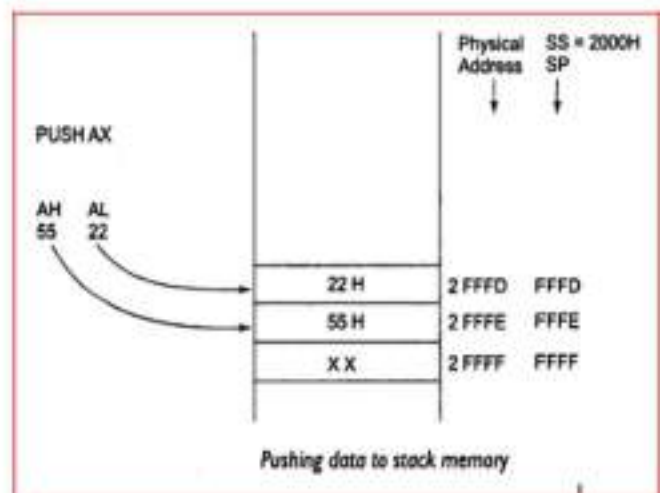
MOV CL, [2000H] ; Copy the 8 bit content of the memory location, at a displacement of 2000H from data segment base to the CL register

MOV [589H], BX ; Copy the 16 bit content of BX register on to the memory location, which at a displacement of 589H from the data segment base. MOV DS, CX ; Move the content of CX to DS

2. PUSH instruction

The PUSH instruction decrements the stack pointer by two and copies the word from source to the location where stack pointer now points. Here the source must of word size data. Source can be a general purpose register, segment register or a memory location.

The PUSH instruction first pushes the most significant byte to sp-1, then the least significant to the sp-2.



Push instruction does not affect any flags.

Example:-

PUSH AX ; Decrements SP by 2, copy content of AX to the stack

(figure shows execution of this instruction)

PUSH DS ; Decrement SP by 2 and copy DS to stack

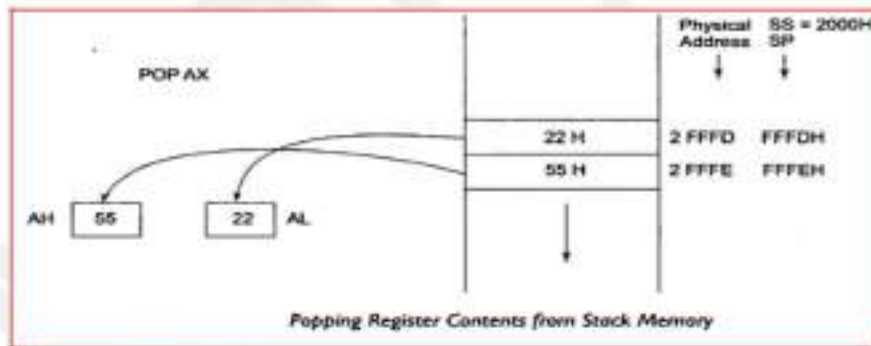
3. POP instruction:

The POP instruction copies a word from the stack location pointed by the stack pointer to the destination. The destination can be a General purpose register, a segment register or a memory location. Here after the content is copied the stack pointer is automatically incremented by two.

The execution pattern is similar to that of the PUSH instruction.

Example:

POP AX ; Copy a word from the top of the stack to CX and increment SP by 2.



4. IN & OUT instructions:

The IN instruction will copy data from a port to the accumulator. If 8 bit is read the data will go to AL and if 16 bit then to AX. Similarly OUT instruction is used to copy data from accumulator to an output port.

Both IN and OUT instructions can be done using direct and indirect addressing modes.

Example:

IN AL, 0F8H ; Copy a byte from the port 0F8H to AL

MOV DX, 30F8H ; Copy port address in DX

IN AL, DX	;	Move 8 bit data from 30F8H port
IN AX, DX	;	Move 16 bit data from 30F8H port
OUT 047H, AL	;	Copy contents of AL to 8 bit port 047H
MOV DX, 30F8H	;	Copy port address in DX
OUT DX, AL	;	Move 8 bit data to the 30F8H port
OUT DX, AX	;	Move 16 bit data to the 30F8H port

5. XCHG instruction

The XCHG instruction exchanges contents of the destination and source. Here destination and source can be register and register or register and memory location, but XCHG cannot interchange the value of 2 memory locations.

Syntax:

XCHG Destination, Source

Example:

XCHG BX, CX ; exchange word in CX with the word in BX

XCHG AL, CL ; exchange byte in CL with the byte in AL

6. LAHF: Load (copy to) AH with the low byte the flag register.

[AH] ← [Flags low
byte] Eg. LAHF

7. SAHF: Store (copy) AH register to low byte of flag register.

[Flags low byte] ← [AH]
Eg. SAHF

8. PUSHF: Copy flag register to top of stack.

[SP] [SP] – 2
[SP] [Flags]
Eg. PUSHF

9. POPF : Copy word at top of stack to flag register.

[Flags] [SP]
[SP] [SP] + 2

Arithmetic Instructions:

The arithmetic and logic logical group of instruction include,

1. ADD instruction

Add instruction is used to add the current contents of destination with that of source and store the result in destination. Here we can use register and/or memory locations. AF, CF, OF, PF, SF, and ZF flags are affected

Syntaxat:

ADD Destination, Source

Example:

- `ADD AL, 0FH` ; Add the immediate content, 0FH to the content of AL and store the result in AL
- `ADD AX, BX` ; $AX \leftarrow AX + BX$
- `ADD AX, 0100H` ; IMMEDIATE
- `ADD AX, BX` ;REGISTER
- `ADD AX, [SI]` ; REGISTER INDIRECT OR INDEXED
- `ADD AX, [5000H]` ;DIRECT

2. ADC: ADD WITH CARRY

This instruction performs the same operation as ADD instruction, but adds the carry flag bit (which may be set as a result of the previous calculation) to the result. All the condition code flags are affected by this instruction. The examples of this instruction along with the modes are as follows:

Example:

- `ADC AX, BX` – REGISTER
- `ADC AX, [SI]` – REGISTER INDIRECT OR INDEXED
- `ADC AX, [5000H]` – DIRECT

3. SUB instruction:

SUB instruction is used to subtract the current contents of destination with that of source and store the result in destination. Here we can use register and/or memory locations. AF, CF, OF, PF, SF, and ZF flags are affected

Syntaxat:

SUB Destination, Source

Example:

SUB AL, 0FH ; subtract the immediate content, 0FH from the content of AL and store the result in AL

SUB AX, BX ; AX <= AX-BX

SUB AX,0100H ; IMMEDIATE (DESTINATION AX)

SUB AX,BX ;REGISTER

SUB AX,[5000H] ;DIRECT

4. SBB: SUBTRACT WITH BORROW

The subtract with borrow instruction subtracts the source operand and the borrow flag (CF) which may reflect the result of the previous calculations, from the destination operand. Subtraction with borrow, here means subtracting 1 from the subtraction obtained by SUB, if carry (borrow) flag is set.

The result is stored in the destination operand. All the flags are affected (condition code) by this instruction. The examples of this instruction are as follows:

Example:

- SBB AX,0100H ;IMMEDIATE (DESTINATION AX)

- SBB AX,BX ;REGISTER

- SBB AX,[5000H] ;DIRECT

5. CMP: COMPARE

The instruction compares the source operand, which may be a register or an immediate data or a memory location, with a destination operand that may be a register or a memory

location. For comparison, it subtracts the source operand from the destination operand but does not store the result anywhere. The flags are affected depending upon the result of the subtraction. If both of the operands are equal, zero flag is set. If the source operand is greater than the destination operand, carry flag is set or else, carry flag is reset. The examples of this instruction are as follows:

Example:

CMP BX,0100H	– IMMEDIATE
CMP AX,0100H	– IMMEDIATE
CMP BX,[SI]	– REGISTER INDIRECT OR INDEXED
CMP BX, CX	– REGISTER

6. INC & DEC instructions

INC and DEC instructions are used to increment and decrement the content of the specified destination by one. AF, CF, OF, PF, SF, and ZF flags are affected.

Example:

- INC AL $AL \leq AL + 1$
- INC AX $AX \leq AX + 1$
- DEC AL $AL \leq AL - 1$
- DEC AX $AX \leq AX - 1$

7. NEG : Negate

The negate instruction forms 2's complement of the specified destination in the instruction. The destination can be a register or a memory location. This instruction can be implemented by inverting each bit and adding 1 to it.

Eg. NEG AL

AL = 0011 0101 35H Replace number in AL with its 2's complement
 complement AL = 1100 1011 = CBH

8. MUL :Unsigned Multiplication Byte or Word

This instruction multiplies an unsigned byte or word by the contents of AL.

Eg. MUL BH ; (AX) (AL) x (BH)
 MUL CX ; (DX)(AX) (AX) x (CX)

9. IMUL :Signed Multiplication

This instruction multiplies a signed byte in source operand by a signed byte in AL or a signed word in source operand by a signed word in AX.

Eg. IMUL BH
IMUL CX
IMUL [SI]

10. CBW : Convert Signed Byte to Word

This instruction copies the sign of a byte in AL to all the bits in AH. AH is then said to be sign extension of AL.

Eg. CBW

AX= 0000 0000 1001 1000 Convert signed byte in AL signed word in AX.
Result in AX = 1111 1111 1001 1000

11.CWD : Convert Signed Word to Double Word

This instruction copies the sign of a word in AX to all the bits in EDI. EDI is then said to be sign extension of AX.

Eg. CWD

Convert signed word in AX to signed double word in EDI :
AX EDI= 1111 1111 1111 1111
Result in EDI = 1111 0000 1100 0001

12. DIV : Unsigned division

This instruction is used to divide an unsigned word by a byte or to divide an unsigned double word by a word.

Eg. DIV CL ; Word in AX / byte in CL
; Quotient in AL, remainder in AH
DIV EDI ; **Double word in EDI and AX / word**
; in EDI, and Quotient in AX,
; remainder in EDI

Logical Instructions:**1. AND instruction**

This instruction logically ANDs each bit of the source byte/word with the corresponding bit in the destination and stores the result in destination. The source can be an immediate number, register or memory location, register can be a register or memory location.

The CF and OF flags are both made zero, PF, ZF, SF are affected by the operation and AF is undefined.

Syntax:**AND Destination, Source****Example:**

AND BL, AL ; suppose BL=1000 0110 and AL = 1100 1010 then after the operation BL would be BL= 1000 0010.

AND CX, AX ; CX <= CX AND AX

AND CL, 08H ; CL<= CL AND (0000 1000)

2. OR instruction:

This instruction logically ORs each bit of the source byte/word with the corresponding bit in the destination and stores the result in destination. The source can be an immediate number, register or memory location, register can be a register or memory location.

The CF and OF flags are both made zero, PF, ZF, SF are affected by the operation and AF is undefined.

Syntax:**OR Destination, Source****Example:**

OR BL, AL ; suppose BL=1000 0110 and AL = 1100 1010 then after the operation BL would be BL= 1100 1110.

OR CX, AX ; CX <= CX AND AX

OR CL, 08H ; CL<= CL AND (0000 1000)

3. NOT instruction:

The NOT instruction complements (inverts) the contents of an operand register or a memory location, bit by bit. The examples are as follows:

Example:

NOT AX (BEFORE AX= (1011)₂= 0BH ; AFTER EXECUTION AX= (0100)₂= 04H.

4. XOR instruction

The XOR operation is again carried out in a similar way to the AND and OR operation. The constraints on the operands are also similar. The XOR operation gives a

high output, when the 2 input bits are dissimilar. Otherwise, the output is zero. The example instructions are as follows:

Example: XOR AX, 0098H

XOR AX, BX

5. TEST : Logical Compare Instruction

The TEST instruction performs a bit by bit logical AND operation on the two operands. The result of this ANDing operation is not available for further use, but flags are affected.

Eg. TEST AX, BX

Shift / Rotate Instructions:

1. ROL – Rotate Left : This instruction rotates all the bits in a specified word or byte to the left some number of bit positions. The data bit rotated out of MSB is circled back into the LSB. It is also copied into CF. In the case of multiple-bit rotate, CF will contain a copy of the bit most recently moved out of the MSB.



Syntax: ROL Destination, Count

Example: ROL AX, 1

2. RCL: Rotate Left through carry: This instruction rotates all the bits in a specified word or byte some number of bit positions to the right. The operation is circular because the LSB of the operand is rotated into the carry flag and the bit in the carry flag is rotated around into MSB of the operand.



Syntax: RCL Destination, Count

Example: RCL AX, 1

3.ROR:Rotate Right: This instruction rotates all the bits in a specified word or byte some number of bit positions to right. The operation is desired as a rotate rather than shift, because the bit moved out of the LSB is rotated around into the MSB. The data bit moved out of the LSB is also copied into CF. In the case of multiple bit rotates, CF will contain a copy of the bit most recently moved out of the LSB.



Syntax: ROR Destination, Count

Example: ROR AX, 1

4.RCR: Rotate Right through: This instruction rotates all the bits in a specified word or byte some number of bit positions to the right. The operation circular because the LSB of the operand is rotated into the carry flag and the bit in the carry flag is rotate around into MSB of the operand.



Syntax: RCR Destination, Count

Example: RCR AX, 1

5. SHL: Shift Logical Left

SAL: Shift arithmetic left

SAL and SHL are two mnemonics for the same instruction. This instruction shifts each bit in the specified destination some number of bit positions to the left. As a bit is shifted out of the LSB operation, a 0 is put in the LSB position. The MSB will be shifted into CF. Bits shifted into CF previously will be lost.



Syntax: SHL/SAL Destination, Count

Example: SHL/SAL AX, 1

6.SAR: Shift arithmetic right: This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a copy of the old MSB is put in the MSB position. In other words, the sign bit is copied into the MSB. The LSB will be shifted into CF. In the case of multiple-bit shift, CF will contain the bit most recently shifted out from the LSB. Bits shifted into CF previously will be lost.



Syntax: SAR Destination, Count

Example: SAR AX, 1

7. SHR: Shift Logical Right: This instruction shifts each bit in the specified destination some number of bit positions to the right. As a bit is shifted out of the MSB position, a 0 is put in its place. The bit shifted out of the LSB position goes to CF. In the case of multi-bit shifts, CF will contain the bit most recently shifted out from the LSB. Bits shifted into CF previously will be lost.



Syntax: SHR Destination, Count

Example: SHR AX, 1

Flag Manipulation Instructions:

The following instructions are used to manipulate the some of the flags

1. STC: SET CARRY FLAG

This instruction sets the carry flag to 1. It does not affect any other flag.

Syntax: STC

2. CLC: CLEAR CARRY FLAG

This instruction resets the carry flag to 0. It does not affect any other flag.

Syntax: CLC

3. CMC: COMPLEMENT CARRY FLAG

This instruction complements the carry flag. It does not affect any other flag.

Syntax: CMC

4. STD: SET DIRECTION FLAG

This instruction sets the direction flag to 1. It does not affect any other flag.

Syntax: STD

5. CLD: CLEAR DIRECTION FLAG

This instruction resets the direction flag to 0. It does not affect any other flag.

Syntax: CLD

6. STI : SET INTERRUPT FLAG

Setting the interrupt flag to a 1 enables the INTR interrupt input of the 8086. The instruction will not take effect until the next instruction after STI. When the INTR input is enabled, an interrupt signal on this input will then cause the 8086 to interrupt program execution, push the return address and flags on the stack, and execute an interrupt service procedure. An IRET instruction at the end of the interrupt service procedure will restore the return address and flags that were pushed onto the stack and return execution to the interrupted program. STI does not affect any other flag.

Syntax: STI

7. CLI : CLEAR INTERRUPT FLAG

This instruction resets the interrupt flag to 0. If the interrupt flag is reset, the 8086 will not respond to an interrupt signal on its INTR input. The CLI instructions, however, has no effect on the non-maskable interrupt input, NMI. It does not affect any other flag.

Syntax: CLI

Machine Control Instructions**1. HLT instruction**

The HLT instruction will cause the 8086 microprocessor stop fetching and executing instructions. The 8086 will enter a halt state. The processor gets out of this Halt signal upon an interrupt signal in INTR pin/NMI pin or a reset signal on RESET input.

Syntax :- HLT

2. WAIT instruction

When this instruction is executed, the 8086 enters into an idle state. This idle state is continued till a high is received on the TEST input pin or a valid interrupt signal is

received. Wait affects no flags. It generally is used to synchronize the 8086 with a peripheral device(s).

Syntax :- WAIT

3. ESC instruction

This instruction is used to pass instruction to a coprocessor like 8087. There is a 6 bit instruction for the coprocessor embedded in the ESC instruction. In most cases the 8086 treats ESC and a NOP, but in some cases the 8086 will access data items in memory for the coprocessor

Syntax :- ESC

4. LOCK instruction

In multiprocessor environments, the different microprocessors share a system bus, which is needed to access external devices like disks. LOCK Instruction is given as prefix in the case when a processor needs exclusive access of the system bus for a particular instruction. It affects no flags.

Example:

LOCK XCHG SEMAPHORE, AL : The XCHG instruction requires two bus accesses. The lock prefix prevents another processor from taking control of the system bus between the 2 accesses

5. NOP instruction:

At the end of NOP instruction, no operation is done other than the fetching and decoding of the instruction. It takes 3 clock cycles. NOP is used to fill in time delays or to provide space for instructions while trouble shooting. NOP affects no flags

Syntax :- NOP

Program control transfer instructions

There are 2 types of such instructions. They are:

- 1. Unconditional transfer instructions – CALL, RET, JMP**
- 2. Conditional transfer instructions – Jump on condition**

1. Unconditional transfer instructions:

(a). CALL instruction: The CALL instruction is used to transfer execution to a subprogram or procedure. There are two types of CALL instructions, near and far.

A **near CALL** is a call to a procedure which is in the same code segment as the CALL instruction. 8086 when encountered a near call, it decrements the SP by 2 and copies the offset of the next instruction after the CALL on the stack. It loads the IP with the offset of the procedure then to start the execution of the procedure.

A **far CALL** is the call to a procedure residing in a different segment. Here value of CS and offset of the next instruction both are backed up in the stack. And then branches to the procedure by changing the content of CS with the segment base containing procedure and IP with the offset of the first instruction of the procedure.

Example:

Near call

CALL PRO ; PRO is the name of the procedure

CALL CX ; Here CX contains the offset of the first instruction of the procedure, that is replaces the content of IP with the content of CX Far call

CALL DWORD PTR[8X] ; New values for CS and IP are fetched from four memory locations in the DS. The new value for CS is fetched from [8X] and [8X+1], the new IP is fetched from [8X+2] and [8X+3].

(b). RET instruction

RET instruction will return execution from a procedure to the next instruction after the CALL instruction in the calling program. If it was a near call, then IP is replaced with the value at the top of the stack, if it had been a far call, then another POP of the stack is required. This second popped data from the stack is put in the CS, thus resuming the execution of the calling program passed.

A RET instruction can be followed by a number, to specify the parameters RET instruction does not affect any flags.

General format: RET**Example:**

```
p1 PROC      ; procedure declaration.  
MOV AX, 1234h ;  
RET          ; return to caller.  
p1 ENDP
```

(c) **JMP instruction:** This is also called as unconditional jump instruction, because the processor jumps to the specified location rather than the instruction after the JMP instruction. Jumps can be **short jumps** when the target address is in the same segment as the JMP instruction or **far jumps** when it is in a different segment.

General Format: JMP <target address>

2. Conditional transfer instructions – Jump on condition: Conditional jumps are always short jumps in 8086. Here jump is done only if the condition specified is true/false. If the condition is not satisfied, then the execution proceeds in the normal way.

Example: There are many conditional jump instructions like

JC : Jump on carry (CF=set)

JNC : Jump on non carry (CF=reset)

JZ : Jump on zero (ZF=set)

JNO : Jump on overflow (OF=set)

Iteration control(LOOP) instructions :

These instructions are used to execute a series of instructions some number of times. The number is specified in the CX register, which will be automatically decremented in course of iteration. But here the destination address for the jump must be in the range of -128 to 127 bytes.

Example: Instructions here are:-

LOOP : loop through the set of instructions until CX is 0

LOOPE/LOOPZ : here the set of instructions are repeated until CX=0 or ZF=0

LOOPNE/LOOPNZ: here repeated until CX=0 or ZF=1

String Instructions

1. MOVS/MOVSMB/MOVSX: These instructions copy a word or byte from a location in the data segment to a location in the extra segment. The offset of the source is in SI and that of destination is in DI. For multiple word/byte transfers the count is stored in the CX register.

When direction flag is 0, SI and DI are incremented and when it is 1, SI and DI are decremented.

MOVS affect no flags. MOVSMB is used for byte sized movements while MOVSX is for word sized.

2. REP/REPE/REP2/REPNE/REPZ

REP is used with string instruction; it repeats an instruction until the specified condition becomes false. **Example:**

REP	=>	CX=0
REPE/REPZ	=>	CX=0 OR ZF=0
REPNE/REPZ	=>	CX=0 OR ZF=1

3. LODS/LODSB/LODSX :

This instruction copies a byte from a string location pointed to by SI to AL or a word from a string location pointed to by SI to AX. LODS does not affect any flags. LODSB copies byte and LODSX copies word.

4. STOS/STOSB/STOSX :

The STOS instruction is used to store a byte/word contained in AL/AX to the offset contained in the DI register. STOS does not affect any flags. After copying the content DI is automatically incremented or decremented, based on the value of direction flag

5. CMPS/CMPSB/CMPSX

CMPS is used to compare the strings, byte wise or word wise. The comparison is affected by subtraction of content pointed by DI from that pointed by SI. The AF, CF, OF, PF, SF and ZF flags are affected by this instruction, but neither operand is affected.

INTEL 8051 MICROCONTROLLER

Introduction: A decade back the process and control operations were totally implemented by the Microprocessors only. But now a days the situation is totally changed and it is occupied by the new devices called Microcontroller. The development is so drastic that we can't find any electronic gadget without the use of a microcontroller. This microcontroller changed the embedded system design so simple and advanced that the embedded market has become one of the most sought after for not only entrepreneurs but for design engineers also.

What is a Microcontroller?

A single chip computer or A CPU with all the peripherals like RAM, ROM, I/O Ports, Timers , ADCs etc... on the same chip. For ex: Motorola's 6811, Intel's 8051, Zilog's Z8 and PIC 16X etc...

MICROPROCESSORS & MICROCONTROLLERS:

Microprocessor: A CPU built into a single VLSI chip is called a microprocessor. It is a general-purpose device and additional external circuitry are added to make it a microcomputer. The microprocessor contains arithmetic and logic unit (ALU), Instruction decoder and control unit, Instruction register, Program counter (PC), clock circuit (internal or external), reset circuit (internal or external) and registers. But the microprocessor has no on chip I/O Ports, Timers , Memory etc.

For example, Intel 8085 is an 8-bit microprocessor and Intel 8086/8088 a 16-bit microprocessor. The block diagram of the Microprocessor is shown in Fig.1

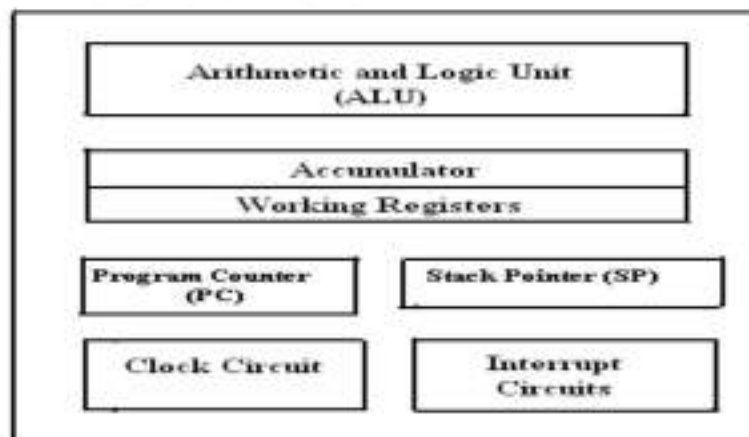


Fig.1 Block diagram of a Microprocessor.

MICROCONTROLLER :

A microcontroller is a highly integrated single chip, which consists of on chip CPU (Central Processing Unit), RAM (Random Access Memory), EPROM/PROM/ROM (Erasable Programmable Read Only Memory), I/O (input/output) – serial and parallel, timers, interrupt controller. For example, Intel 8051 is 8-bit microcontroller and Intel 8096 is 16-bit microcontroller. The block diagram of Microcontroller is shown in Fig.2.

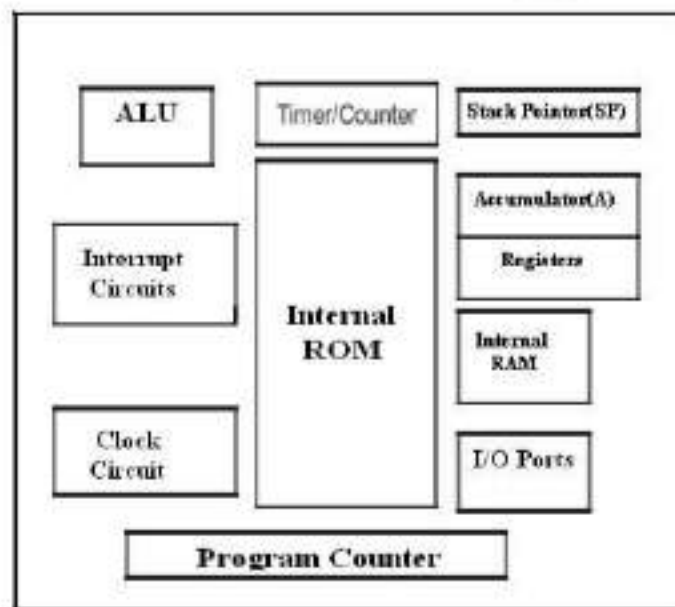


Fig.2. Block Diagram of a Microcontroller

Distinguish between Microprocessor and Microcontroller

S.No	Microprocessor	Microcontroller
1	A microprocessor is a general purpose device which is called a CPU	A microcontroller is a dedicated chip which is also called single chip computer.
2	A microprocessor do not contain onchip I/O Ports, Timers, Memories etc..	A microcontroller includes RAM, ROM, serial and parallel interface, timers, interrupt circuitry (in addition to CPU) in a single chip.
3	Microprocessors are most commonly used as the CPU in microcomputer systems	Microcontrollers are used in small, minimum component designs performing Control-oriented applications.
4	Microprocessor instructions are mainly nibble or byte addressable	Microcontroller instructions are both bit Addressable as well as byte addressable.
5	Microprocessor instruction sets are mainly intended for catering to Large volumes of data.	Microcontrollers have instruction sets catering to the control of inputs and Outputs.
6	Microprocessor based system design is complex and expensive	Microcontroller based system design is rather simple and cost effective
7	The Instruction set of microprocessor is complex with large number of instructions.	The instruction set of a Microcontroller is very simple with less number of instructions. For, ex: PIC microcontrollers have only 35 instructions.
8	A microprocessor has zero status flag	A microcontroller has no zero flag.

INTEL 8051 MICROCONTROLLER:

The 8051 microcontroller is a very popular 8-bit microcontroller introduced by Intel in the year 1981 and it has become almost the academic standard now a days. The 8051 is based on an 8-bit CISC core with Harvard architecture. Its 8-bit architecture is optimized for control applications with extensive Boolean processing. It is available as a 40-pin DIP chip and works at +5 Volts DC. The salient features of 8051 controller are given below.

SALIENT FEATURES: The salient features of 8051 Microcontroller are

1. 4 KB on chip program memory (ROM or EPROM).
2. 128 bytes on chip data memory(RAM).
3. 8-bit data bus
4. 16-bit address bus
5. 32 general purpose registers each of 8 bits
6. Two -16 bit timers T₀ and T₁
7. Five Interrupts (3 internal and 2 external).
8. Four Parallel ports each of 8-bits (PORT0, PORT1, PORT2, PORT3) with a total of 32 I/O lines.
9. One 16-bit program counter and One 16-bit DPTR (data pointer)
10. One 8-bit stack pointer

ARCHITECTURE & BLOCK DIAGRAM OF 8051 MICROCONTROLLER:

The architecture of the 8051 microcontroller can be understood from the block diagram. It has Harward architecture with RISC (Reduced Instruction Set Computer) concept. The blockdiagram of 8051 microcontroller is shown in Fig 3. below1

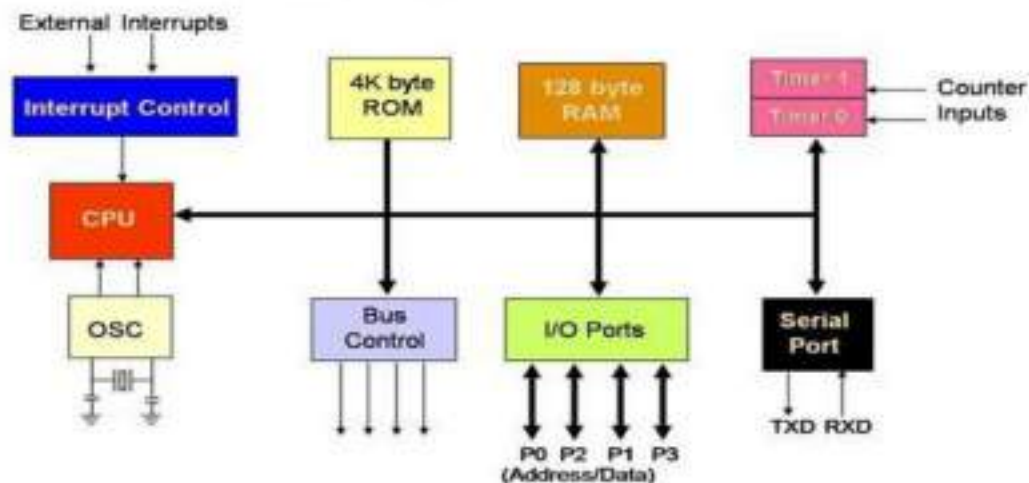
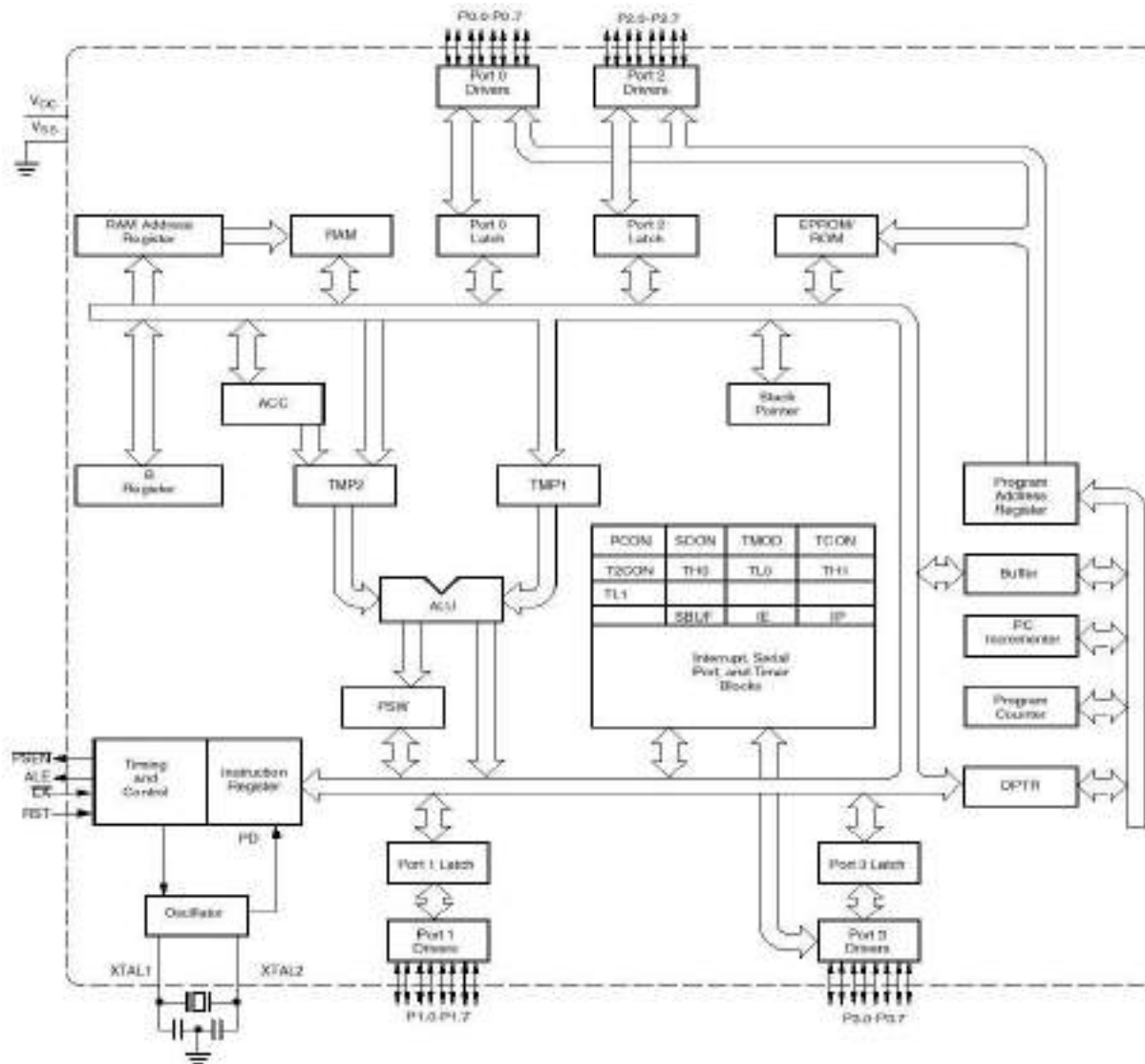


Fig.3. Block Diagram of 8051 Microcontroller.

It consists of an 8-bit ALU, one 8-bit PSW(Program Status Register), A and B registers , one 16-bit Program counter , one 16-bit Data pointer register(DPTR),128 bytes of RAM and 4kB of ROM and four parallel I/O ports each of 8-bit width.

8051 has 8-bit ALU which can perform all the 8-bit arithmetic and logical operations in one machine cycle. The ALU is associated with two registers A & B



A and B Registers : The A and B registers are special function registers which hold the results of many arithmetic and logical operations of 8051. The A register is also called the **Accumulator** and as it's name suggests, is used as a general register to accumulate the

results of a large number of instructions. By default it is used for all mathematical operations and also data transfer operations between CPU and any external memory. The B register is mainly used for multiplication and division operations along with A register.

MUL AB : DIV AB.

It has no other function other than as a location where data may be stored.

The R registers: The "R" registers are a set of eight registers that are named R0, R1, etc. up to and including R7. These registers are used as auxiliary registers in many operations. The "R" registers are also used to temporarily store values.

Program Counter (PC) : 8051 has a 16-bit program counter .The program counter always points to the address of the next instruction to be executed. After execution of one instruction the program counter is incremented to point to the address of the next instruction to be executed. It is the contents of the PC that are placed on the address bus to find and fetch the desired instruction. Since the PC is 16-bit width ,8051 can access program addresses from 0000H to FFFFH ,a total of 6kB of code.

Stack Pointer Register (SP) : It is an 8-bit register which stores the address of the stack top. i.e the Stack Pointer is used to indicate where the next value to be removed from the stack should be taken from. When a value is pushed onto the stack, the 8051 first increments the value of SP and then stores the value at the resulting memory location. Similarly when a value is popped off the stack, the 8051 returns the value from the memory location indicated by SP, and then decrements the value of SP. Since the SP is only 8-bit wide it is incremented or decremented by two. SP is modified directly by the 8051 by six instructions: PUSH, POP, ACALL, LCALL, RET, and RETI. It is also used intrinsically whenever an interrupt is triggered.

STACK in 8051 Microcontroller : The stack is a part of RAM used by the CPU to store information temporarily. This information may be either data or an address .The CPU needs this storage area as there are only limited number of registers. The register used to access the stack is called the Stack pointer which is an 8-bit register..So,it can take values of 00 to FF H. When the 8051 is powered up ,the SP register contains the

value 07.i.e the RAM location value 08 is the first location being used for the stack by the 8051 controller

There are two important instructions to handle this stack. One is the PUSH and the other is the POP. The loading of data from CPU registers to the stack is done by PUSH and the loading of the contents of the stack back into a CPU register is done by POP.

```
EX : MOV R6 ,#35 H
      MOV R1 ,#21 H
      PUSH 6
      PUSH 1
```

In the above instructions the contents of the Registers R6 and R1 are moved to stack and they occupy the 08 and 09 locations of the stack. Now the contents of the SP are incremented by two and it is 0A

Similarly POP 3 instruction pops the contents of stack into R3 register. Now the contents of the SP is decremented by 1

In 8051 the RAM locations 08 to 1F (24 bytes) can be used for the Stack. In any program if we need more than 24 bytes of stack ,we can change the SP point to RAM locations 30-7F H. this can be done with the instruction MOV SP,# XX.

Data Pointer Register (DPTR) : It is a 16-bit register which is the only user-accessible. DPTR, as the name suggests, is used to point to data. It is used by a number of commands which allow the 8051 to access external memory. When the 8051 accesses external memory it will access external memory at the address indicated by DPTR. This DPTR can also be used as two 8-registers DPH and DPL.

Program Status Register (PSW): The 8051 has a 8-bit PSW register which is also known as Flag register. In the 8-bit register only 6-bits are used by 8051. The two unused bits are user definable bits. In the 6-bits four of them are conditional flags .They are Carry –CY, Auxiliary Carry-AC, Parity-P, and Overflow-OV .These flag bits indicate some conditions that resulted after an instruction was executed.



The bits PSW3 and PSW4 are denoted as RS0 and RS1 and these bits are used to select the bank registers of the RAM location. The meaning of various bits of PSW register is shown below.

CY	PSW.7	Carry Flag
AC	PSW.6	Auxiliary Carry Flag
FO	PSW.5	Flag 0 available for general purpose.
RS1	PSW.4	Register Bank select bit 1
RS0	PSW.3	Register bank select bit 0
OV	PSW.2	Overflow flag
---	PSW.1	User definable flag
P	PSW.0	Parity flag .set/cleared by hardware.

The selection of the register Banks and their addresses are given below.

RS1	RS0	Register Bank	Address
0	0	0	00H-07H
0	1	1	08H-0FH
1	0	2	10H-17H
1	1	3	18H-1FH

Memory organization : The 8051 microcontroller has 128 bytes of Internal RAM and 4kB of on chip ROM .The RAM is also known as Data memory and the ROM is known as program memory. The program memory is also known as Code memory .This Code memory holds the actual 8051 program that is to be executed. In 8051 this

memory is limited to 64K. Code memory may be found on-chip, as ROM or EPROM. It may also be stored completely off-chip in an external ROM or, more commonly, an external EPROM. The 8051 has only 128 bytes of Internal RAM but it supports 64kB of external RAM. As the name suggests, external RAM is any random access memory which is off-chip. Since the memory is off-chip it is not as flexible in terms of accessing, and is also slower. For example, to increment an Internal RAM location by 1, it requires only 1 instruction and 1 instruction cycle but to increment a 1-byte value stored in External RAM requires 4 instructions and 7 instruction cycles. So, here the external memory is 7 times slower.

Internal RAM OF 8051 : This Internal RAM is found on-chip on the 8051. So it is the fastest RAM available, and it is also the most flexible in terms of reading, writing, and modifying its contents. Internal RAM is volatile, so when the 8051 is reset this memory is cleared. The 128 bytes of internal RAM is organized as below.

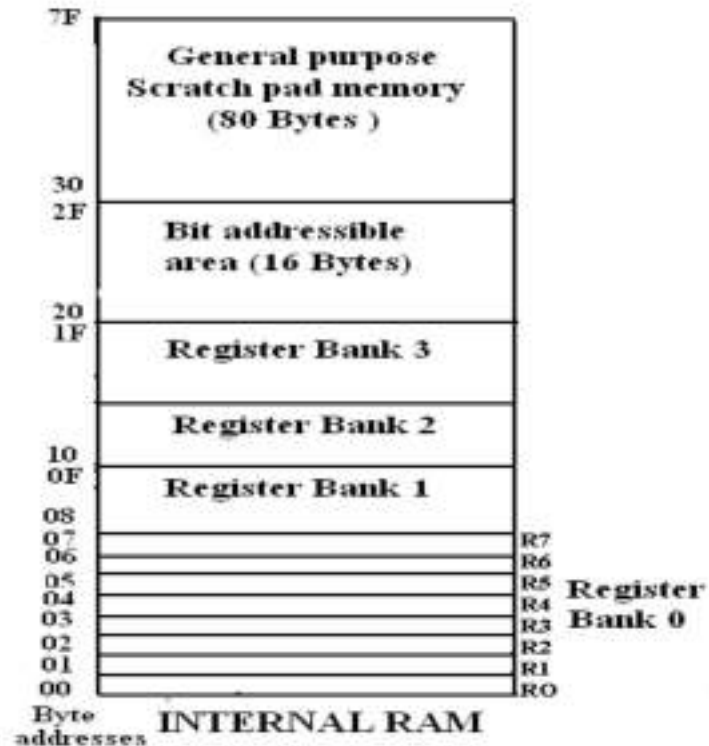
(i) Four register banks (Bank0, Bank1, Bank2 and Bank3) each of 8-bits (total 32 bytes). The default bank register is Bank0. The remaining Banks are selected with the help of RS0 and RS1 bits of PSW Register.

(ii) 16 bytes of bit addressable area and

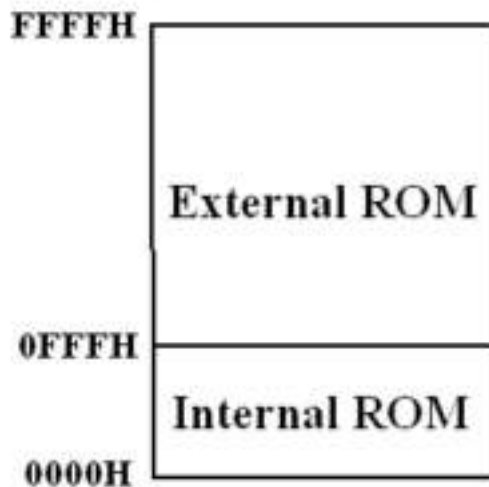
(iii) 80 bytes of general purpose area (Scratch pad memory) as shown in the diagram below. This area is also utilized by the microcontroller as a storage area for the operating stack.

The 32 bytes of RAM from address 00 H to 1FH are used as working registers organized as four banks of eight registers each. The registers are named as R0-R7. Each register can be addressed by its name or by its RAM address.

For EX : MOV A, R7 or MOV R7, #05H



Internal ROM (On –chip ROM): The 8051 microcontroller has 4kB of on chip ROM but it can be extended up to 64kB. This ROM is also called program memory or code memory. The CODE segment is accessed using the program counter (PC) for opcode fetches and by DPTR for data. The external ROM is accessed when the EA(active low) pin is connected to ground or the contents of program counter exceeds 0FFFH. When the Internal ROM address is exceeded the 8051 automatically fetches the code bytes from the external program memory.



PARALLEL I/O PORTS :

The 8051 microcontroller has four parallel I/O ports , each of 8-bits .So, it provides the user 32 I/O lines for connecting the microcontroller to the peripherals. The four ports are P0 (Port 0), P1(Port1) ,P2(Port 2) and P3 (Port3). Upon reset all the ports are output ports. In order to make them input, all the ports must be set i.e a high bit must be sent to all the port pins. This is normally done by the instruction “SETB”.

Ex: MOV A, #0FFH ; A = FF
MOV P0,A ; make P0 an input port

PORT 0: Port 0 is an 8-bit I/O port with dual purpose. If external memory is used, these port pins are used for the lower address byte address/data (AD₀-AD₇), otherwise all bits of the port are either input or output..

Dual role of port 0: Port 0 can also be used as address/data bus(AD₀-AD₇), allowing it to be used for both address and data. When connecting the 8051 to an external memory, port 0 provides both address and data. The 8051 multiplexes address and data through port 0 to save the pins. ALE indicates whether P0 has address or data. When ALE = 0, it provides data D₀-D₇, and when ALE =1 it provides address

Port 1: Port 1 occupies a total of 8 pins (pins 1 through 8). It has no dual application and acts only as input or output port. Upon reset, Port 1 is configured as an output port. To configure it as an input port , port bits must be set i.e a high bit must be sent to all the port pins. This is normally done by the instruction “SETB”.

For Ex :

MOV A, #0FFH ; A=FF HEX
MOV P1,A ; make P1 an input port by writing 1's to all of its pins

Port 2: Port 2 is also an eight bit parallel port. (pins 21- 28). It can be used as input or output port. Upon reset, Port 2 is configured as an output port. If the port is to be used as input port, all the port bits must be made high by sending FF to the port. For ex,

MOV A, #0FFH ; A=FF hex
MOV P2, A ; make P2 an input port by writing all 1's to it

Dual role of port 2 : Port2 lines are also associated with the higher order address lines A8-A15. Port 2 is used along with P0 to provide the 16-bit address for the external memory. Since an 8051 is capable of accessing 64K bytes of external memory, it needs a path for the 16 bits of the address. While P0 provides the lower 8 bits via A0-A7, it is the job of P2 to provide bits A8-A15 of the address. In other words, when 8051 is connected to external memory, Port 2 is used for the upper 8 bits of the 16 bit address, and it cannot be used for I/O operations.

PORT 3 : Port3 is also an 8-bit parallel port with dual function.(pins 10 to 17). The port pins can be used for I/O operations as well as for control operations. The details of these additional operations are given below in the table. Upon reset port 3 is configured as an output port. If the port is to be used as input port, all the port bits must be made high by sending FF to the port. For ex,

MOV A, #0FFH ; A= FF hex

MOV P3, A ; make P3 an input port by writing all 1's to it

Alternate Functions of Port 3 : P3.0 and P3.1 are used for the RxD (Receive Data) and TxD (Transmit Data) serial communications signals. Bits P3.2 and P3.3 are meant for external interrupts. Bits P3.4 and P3.5 are used for Timers 0 and 1 and P3.6 and P3.7 are used to provide the write and read signals of external memories connected in 8031 based systems

Table: PORT 3 alternate functions

S.No	Port 3 bit	Pin No	Function
1	P3.0	10	RxD
2	P3.1	11	TxD
3	P3.2	12	$\overline{\text{INT0}}$
4	P3.3	13	$\overline{\text{INT1}}$
5	P3.4	14	T0
6	P3.5	15	T1
7	P3.6	16	$\overline{\text{WR}}$
8	P3.7	17	$\overline{\text{RD}}$

8051 Instructions: The process of writing program for the microcontroller mainly consists of giving instructions (commands) in the specific order in which they should be executed in order to carry out a specific task. All commands are known as INSTRUCTION SET. All microcontrollers compatible with the 8051 have in total of 255 instructions

These can be grouped into the following categories

1. **Arithmetic Instructions**
2. **Logical Instructions**
3. **Data Transfer instructions**
4. **Boolean Variable Instructions**
5. **Program Branching Instructions**

The following nomenclatures for register, data, address and variables are used while write instructions.

A: Accumulator

B: "B" register

C: Carry bit

Rn: Register R0 - R7 of the currently selected register bank

Direct: 8-bit internal direct address for data. The data could be in lower 128bytes of RAM (00 - 7FH) or it could be in the special function register (80 - FFH).

@Ri: 8-bit external or internal RAM address available in register R0 or R1. This is used for indirect addressing mode.

#data8: Immediate 8-bit data available in the instruction.

#data16: Immediate 16-bit data available in the instruction.

Addr11: 11-bit destination address for short absolute jump. Used by instructions AJMP & ACALL. Jump range is 2 kbyte (one page).

Addr16: 16-bit destination address for long call or long jump.

bit: Directly addressed bit in internal RAM or SFR

The first part of each instruction, called MNEMONIC refers to the operation an instruction performs (copy, addition, logic operation etc.). Mnemonics are abbreviations of the name of operation being executed.

The other part of instruction, called OPERAND is separated from mnemonic by at least one whitespace and defines data being processed by instructions. Some of the instructions have no operand, while some of them have one, two or three. If there is more than one operand in an instruction, they are separated by a comma

1.Arithmetic Instructions: Arithmetic instructions perform several basic arithmetic operations such as addition, subtraction, division, multiplication etc. After execution, the result is stored in the first operand.

For example:

ADD A, R1 - The result of addition (A+R1) will be stored in the accumulator.

Mnemonics	Description
ADD A, Rn	$A \leftarrow A + Rn$
ADD A, direct	$A \leftarrow A + (\text{direct})$
ADD A, @Ri	$A \leftarrow A + @Ri$
ADD A, #data	$A \leftarrow A + \text{data}$
ADDC A, Rn	$A \leftarrow A + Rn + C$
ADDC A, direct	$A \leftarrow A + (\text{direct}) + C$
ADDC A, @Ri	$A \leftarrow A + @Ri + C$
ADDC A, #data	$A \leftarrow A + \text{data} + C$
DA A	Decimal adjust accumulator
DIV AB	Divide A by B $A \leftarrow \text{quotient}$ $B \leftarrow \text{remainder}$
DEC A	$A \leftarrow A - 1$
DEC Rn	$Rn \leftarrow Rn - 1$
DEC direct	$(\text{direct}) \leftarrow (\text{direct}) - 1$
DEC @Ri	$@Ri \leftarrow @Ri - 1$
INC A	$A \leftarrow A + 1$
INC Rn	$Rn \leftarrow Rn + 1$
INC direct	$(\text{direct}) \leftarrow (\text{direct}) + 1$
INC @Ri	$@Ri \leftarrow @Ri + 1$
INC DPTR	$DPTR \leftarrow DPTR + 1$
MUL AB	Multiply A by B $A \leftarrow \text{low byte } (A * B)$ $B \leftarrow \text{high byte } (A * B)$
SUBB A, Rn	$A \leftarrow A - Rn - C$
SUBB A, direct	$A \leftarrow A - (\text{direct}) - C$
SUBB A, @Ri	$A \leftarrow A - @Ri - C$
SUBB A, #data	$A \leftarrow A - \text{data} - C$

2.Logical Instructions: The Logical Instructions are used to perform logical operations like AND, OR, XOR, NOT, Rotate, Clear and Swap. Logical Instruction are performed on Bytes of data on a bit-by-bit basis. Logic instructions perform logic operations upon corresponding bits of two registers. After execution, the result is stored in the first operand.

Mnemonics	Description
ANL A, Rn	$A \leftarrow A \text{ AND } Rn$
ANL A, direct	$A \leftarrow A \text{ AND (direct)}$
ANL A, @Ri	$A \leftarrow A \text{ AND } @Ri$
ANL A, #data	$A \leftarrow A \text{ AND data}$
ANL direct, A	$(\text{direct}) \leftarrow (\text{direct}) \text{ AND } A$
ANL direct, #data	$(\text{direct}) \leftarrow (\text{direct}) \text{ AND data}$
CLR A	$A \leftarrow 00H$
CPL A	$A \leftarrow \neg A$
ORL A, Rn	$A \leftarrow A \text{ OR } Rn$
ORL A, direct	$A \leftarrow A \text{ OR (direct)}$
ORL A, @Ri	$A \leftarrow A \text{ OR } @Ri$
ORL A, #data	$A \leftarrow A \text{ OR data}$
ORL direct, A	$(\text{direct}) \leftarrow (\text{direct}) \text{ OR } A$
ORL direct, #data	$(\text{direct}) \leftarrow (\text{direct}) \text{ OR data}$
RL A	Rotate accumulator left
RLC A	Rotate accumulator left through carry
RR A	Rotate accumulator right
RRC A	Rotate accumulator right through carry
SWAP A	Swap nibbles within Accumulator
XRL A, Rn	$A \leftarrow A \text{ EXOR } Rn$
XRL A, direct	$A \leftarrow A \text{ EXOR (direct)}$
XRL A, @Ri	$A \leftarrow A \text{ EXOR } @Ri$
XRL A, #data	$A \leftarrow A \text{ EXOR data}$
XRL direct, A	$(\text{direct}) \leftarrow (\text{direct}) \text{ EXOR } A$
XRL direct, #data	$(\text{direct}) \leftarrow (\text{direct}) \text{ EXOR data}$

3. Data Transfer Instructions: Data transfer instructions move the content of one register to another. The register the content of which is moved remains unchanged. If they have the suffix “X” (MOVX), the data is exchanged with external memory.

Mnemonics	Description
MOV A, Rn	$A \leftarrow Rn$
MOV A, direct	$A \leftarrow (\text{direct})$
MOV A, @Ri	$A \leftarrow @Ri$
MOV A, #data	$A \leftarrow \text{data}$
MOV Rn, A	$Rn \leftarrow A$
MOV Rn, direct	$Rn \leftarrow (\text{direct})$

MOV Rn, #data	$Rn \leftarrow \text{data}$
MOV direct, A	$(\text{direct}) \leftarrow A$
MOV direct, Rn	$(\text{direct}) \leftarrow Rn$
MOV direct1, direct2	$(\text{direct1}) \leftarrow (\text{direct2})$
MOV direct, @Ri	$(\text{direct}) \leftarrow @Ri$
MOV direct, #data	$(\text{direct}) \leftarrow \text{\#data}$
MOV @Ri, A	$@Ri \leftarrow A$
MOV @Ri, direct	$@Ri \leftarrow (\text{direct})$
MOV @Ri, #data	$@Ri \leftarrow \text{\#data}$
MOV DPTR, #data16	$DPTR \leftarrow \text{data16}$
MOVC A, @A+DPTR	$A \leftarrow \text{Code byte pointed by } A + DPTR$
MOVC A, @A+PC	$A \leftarrow \text{Code byte pointed by } A + PC$
MOVC A, @Ri	$A \leftarrow \text{Code byte pointed by Ri 8-bit address}$
MOVX A, @DPTR	$A \leftarrow \text{External data pointed by DPTR}$
MOVX @Ri, A	$@Ri \leftarrow A \text{ (External data - 8bit address)}$
MOVX @DPTR, A	$@DPTR \leftarrow A \text{ (External data - 16bit address)}$
PUSH direct	$(SP) \leftarrow (\text{direct})$
POP direct	$(\text{direct}) \leftarrow (SP)$
XCH Rn	Exchange A with Rn
XCH direct	Exchange A with direct byte
XCH @Ri	Exchange A with indirect RAM
XCHD A, @Ri	Exchange least significant nibble of A with that of indirect RAM

4.Boolean Variable Instructions: Boolean or Bit Manipulation Instructions will deal with bit variables. Similar to logic instructions, bit-oriented instructions perform logic operations. The difference is that these are performed upon single bits.

Mnemonics	Description
CLR C	$C \leftarrow \text{-bit0}$
CLR bit	$\text{bit} \leftarrow 0$
SET C	$C \leftarrow 1$
SET bit	$\text{bit} \leftarrow 1$
CPL C	$C \leftarrow \overline{C}$
CPL bit	$\text{bit} \leftarrow \overline{\text{bit}}$
ANL C, /bit	$C \leftarrow C \cdot \overline{\text{bit}}$
ANL C, bit	$C \leftarrow C \cdot \text{bit}$
ORL C, /bit	$C \leftarrow C + \overline{\text{bit}}$

ORL C, bit	$C \leftarrow C + \text{bit}$
MOV C, bit	$C \leftarrow \text{bit}$
MOV bit, C	$\text{Bit} \leftarrow C$

5.Program Branching Instructions: There are two kinds of branch instructions:

Unconditional jump instructions: upon their execution a jump to a new location from where the program continues execution is executed.

Conditional jump instructions: Jump to a new program location is executed only if a specified condition is met. Otherwise, the program normally proceeds with the next instruction.

Mnemonics	Description
ACALL addr11	$PC + 2 \rightarrow (SP) ; \text{addr11} \rightarrow PC$
AJMP addr11	$\text{Addr11} \rightarrow PC$
CJNE A, direct, rel	Compare with A, jump (PC + rel) if not equal
CJNE A, #data, rel	Compare with A, jump (PC + rel) if not equal
CJNE Rn, #data, rel	Compare with Rn, jump (PC + rel) if not equal
CJNE @Ri, #data, rel	Compare with @Ri A, jump (PC + rel) if not equal
DJNZ Rn, rel	Decrement Rn, jump if not zero
DJNZ direct, rel	Decrement (direct), jump if not zero
JC rel	Jump (PC + rel) if C bit = 1
JNC rel	Jump (PC + rel) if C bit = 0
JB bit, rel	Jump (PC + rel) if bit = 1
JNB bit, rel	Jump (PC + rel) if bit = 0
JBC bit, rel	Jump (PC + rel) if bit = 1
JMP @A+DPTR	$A+DPTR \rightarrow PC$
JZ rel	If A=0, jump to PC + rel
JNZ rel	If A ≠ 0, jump to PC + rel
LCALL addr16	$PC + 3 \rightarrow (SP), \text{addr16} \rightarrow PC$
LJMP addr16	$\text{Addr16} \rightarrow PC$
NOP	No operation
RET	$(SP) \rightarrow PC$
RETI	$(SP) \rightarrow PC$, Enable Interrupt
SJMP rel	$PC + 2 + \text{rel} \rightarrow PC$
JMP @A+DPTR	$A+DPTR \rightarrow PC$
JZ rel	If A = 0. jump PC+ rel
JNZ rel	If A ≠ 0, jump PC + rel
NOP	No operation

Memory interfacing to 8051: The system designer is not limited by the amount of internal RAM and ROM available on chip. Two separate external memory spaces are made available by the 16-bit PC and DPTR and by different control pins for enabling external ROM and RAM chips.

External RAM, which is accessed by the DPTR, may also be needed when 128 bytes of internal data storage is not sufficient. External RAM, up to 64K bytes, may also be added to any chip in the 8051 family.

Connecting External Memory: Figures shows the connections between an 8051 and an external memory configuration consisting of EPROM and static RAM. The 8051 accesses external RAM whenever certain program instructions are executed. External ROM is accessed whenever the EA (external access) pin is connected to ground or when the PC contains an address higher than the last address in the internal 4K bytes ROM (FFFFh). 8051 designs can thus use internal and external ROM automatically; the 8051, having no internal ROM, must have (EA)' grounded.

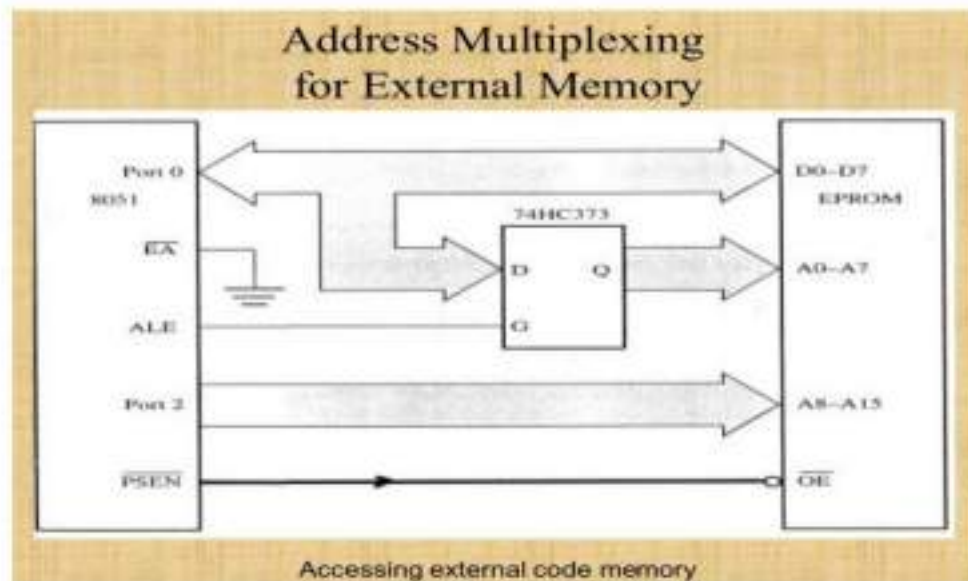


Figure: Interfacing External code memory to 8051

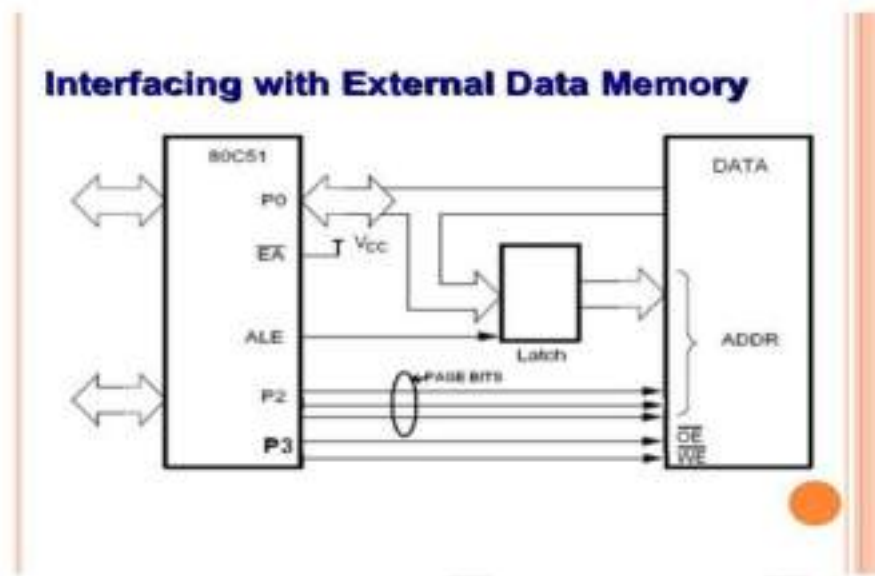


Figure: Interfacing External data memory to 8051

If the memory access is for a byte of program code in the ROM, the (PSEN)'(program store enable) pin will go low to enable the ROM to place a byte of program code on the data bus. If the access is for a RAM byte, the (WR)'(write) or (RD)'(read) pins will go low, enabling data to flow between the RAM and the data bus.

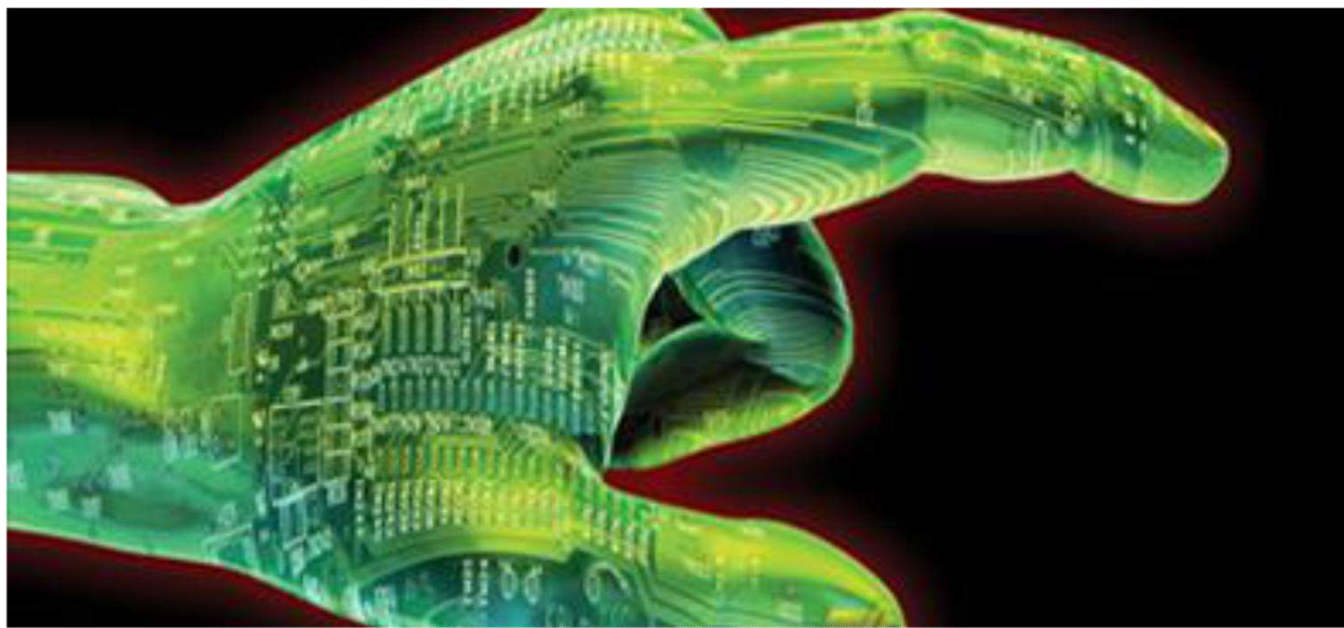
Note that the (WR)' and (RD)' signals are alternate uses for port 3 pins 16 and 17. Also, port 0 is used for the lower address byte and data; port 2 is used for upper address bits. The use of external memory consumes many of the port pins, leaving only port 1 and parts of port 3 for general I/O.

Text Books

1. D. V. Hall, Microprocessors and Interfacing, TMGH, 2nd Edition 2006.
2. Advanced Microprocessors and Peripherals – A. K. Ray and K.M. Bhurchandi, TMH, 2nd Edition 2006
3. Kenneth. J. Ayala, The 8051 Microcontroller , 3rd Ed., Cengage Learning

Embedded Systems

III - ESE ::



UNIT-2

Introduction to Embedded Systems

N SURESH
T Vinay Simha Reddy
Department of ECE



**MALLA REDDY COLLEGE OF
ENGINEERING & TECHNOLOGY**

Permanently Affiliated to JNTUH and Approved by AICTE, New Delhi

1. Introduction to Embedded Systems

What is Embedded System?

(DEC2016, March-2017.)

An Electronic/Electro mechanical system which is designed to perform a specific function and is a combination of both hardware and firmware (Software)

E.g. Electronic Toys, Mobile Handsets, Washing Machines, Air Conditioners, Automotive Control Units, Set Top Box, DVD Player etc...

Embedded Systems are:

- ☐ Unique in character and behavior
- ☐ With specialized hardware and software

Embedded Systems Vs General Computing Systems:

(March-2017)

General Purpose Computing System	Embedded System
A system which is a combination of generic hardware and General Purpose Operating System for executing a variety of applications	A system which is a combination of special purpose hardware and embedded OS for executing a specific set of applications
Contain a General Purpose Operating System (GPOS)	May or may not contain an operating system for functioning
Applications are alterable (programmable) by user (It is possible for the end user to re-install the Operating System, and add or remove user applications)	The firmware of the embedded system is pre-programmed and it is non-alterable by end-user
Performance is the key deciding factor on the selection of the system. Always „Faster is Better“	Application specific requirements (like performance, power requirements, memory usage etc) are the key deciding factors
Less/not at all tailored towards reduced operating power requirements, options for different levels of power management.	Highly tailored to take advantage of the power saving modes supported by hardware and Operating System
Response requirements are not time critical	For certain category of embedded systems like mission critical systems, the response time requirement is highly critical
Need not be deterministic in execution behavior	Execution behavior is deterministic for certain type of embedded systems like „Hard Real Time“ systems

History of Embedded Systems:

- ❑ First Recognized Modern Embedded System: Apollo Guidance Computer (AGC) developed by [Charles Stark Draper](#) at the MIT Instrumentation Laboratory.

- It has two modules
- 1.Command module(CM) 2.Lunar Excursion module(LEM)
- RAM size 256 , 1K ,2K words
- ROM size 4K,10K,36K words
- Clock frequency is 1.024MHz
- 5000 ,3-input RTL NOR gates are used
- User interface is DSKY(display/Keyboard)



- First Mass Produced Embedded System: Autonetics **D-17** Guidance computer for Minuteman-I missile

Classification of Embedded Systems:

(March-2017)

- ❑ Based on Generation
- ❑ Based on Complexity & Performance Requirements
- ❑ Based on deterministic behavior
- ❑ Based on Triggering

1. Embedded Systems - Classification based on Generation

- **First Generation:** The early embedded systems built around 8-bit microprocessors like 8085 and Z80 and 4-bit microcontrollers
EX. stepper motor control units, Digital Telephone Keypads etc.
- **Second Generation:** Embedded Systems built around 16-bit microprocessors and 8 or 16-bit microcontrollers, following the first generation embedded systems
EX.SCADA, Data Acquisition Systems etc.
- **Third Generation:** Embedded Systems built around high performance 16/32 bit Microprocessors/controllers, Application Specific Instruction set processors like Digital Signal Processors (DSPs), and Application Specific Integrated Circuits (ASICs).The instruction set is complex and powerful.
EX. Robotics, industrial process control, networking etc.

- **Fourth Generation:** Embedded Systems built around System on Chips (SoCs), Re-configurable processors and multicore processors. It brings high performance, tight integration and miniaturization into the embedded device market
EX Smart phone devices, MIDs etc.

2. Embedded Systems - Classification based on Complexity & Performance

- **Small Scale:** The embedded systems built around low performance and low cost 8 or 16 bit microprocessors/ microcontrollers. It is suitable for simple applications and where performance is not time critical. It may or may not contain OS.
- **Medium Scale:** Embedded Systems built around medium performance, low cost 16 or 32 bit microprocessors / microcontrollers or DSPs. These are slightly complex in hardware and firmware. It may contain GPOS/RTOS.
- **Large Scale/Complex:** Embedded Systems built around high performance 32 or 64 bit RISC processors/controllers, RSoC or multi-core processors and PLD. It requires complex hardware and software. These system may contain multiple processors/controllers and co-units/hardware accelerators for offloading the processing requirements from the main processor. It contains RTOS for scheduling, prioritization and management.

3. Embedded Systems - Classification Based on deterministic behavior: It is applicable for Real Time systems. The application/task execution behavior for an embedded system can be either deterministic or non-deterministic

These are classified in to two types

- 1. Soft Real time Systems:** Missing a deadline may not be critical and can be tolerated to a certain degree
- 2. Hard Real time systems:** Missing a program/task execution time deadline can have catastrophic consequences (financial, human loss of life, etc.)

4. Embedded Systems - Classification Based on Triggering:

These are classified into two types

- 1. Event Triggered :** Activities within the system (e.g., task run-times) are dynamic and depend upon occurrence of different events .
- 2. Time triggered:** Activities within the system follow a statically computed schedule (i.e., they are allocated time slots during which they can take place) and thus by nature are predictable.

Major Application Areas of Embedded Systems:

- ☐ **Consumer Electronics:** Camcorders, Cameras etc.
- ☐ **Household Appliances:** Television, DVD players, washing machine, Fridge, Microwave Oven etc.
- ☐ **Home Automation and Security Systems:** Air conditioners, sprinklers, Intruder detection alarms, Closed Circuit Television Cameras, Fire alarms etc.
- ☐ **Automotive Industry:** Anti-lock breaking systems (ABS), Engine Control, Ignition Systems, Automatic Navigation Systems etc.
- ☐ **Telecom:** Cellular Telephones, Telephone switches, Handset Multimedia Applications etc.
- ☐ **Computer Peripherals:** Printers, Scanners, Fax machines etc.
- ☐ **Computer Networking Systems:** Network Routers, Switches, Hubs, Firewalls etc.
- ☐ **Health Care:** Different Kinds of Scanners, EEG, ECG Machines etc.
- ☐ **Measurement & Instrumentation:** Digital multi meters, Digital CROs, Logic Analyzers PLC systems etc.
- ☐ **Banking & Retail:** Automatic Teller Machines (ATM) and Currency counters, Point of Sales (POS)
- ☐ **Card Readers:** Barcode, Smart Card Readers, Hand held Devices etc.

Purpose of Embedded Systems:**(DEC2016)**

Each Embedded Systems is designed to serve the purpose of any one or a combination of the following tasks.

- Data Collection/Storage/Representation
- Data Communication
- Data (Signal) Processing
- Monitoring
- Control
- Application Specific User Interface

1. Data Collection/Storage/Representation:-

- ❖ Performs acquisition of data from the external world.
- ❖ The collected data can be either analog or digital
- ❖ Data collection is usually done for storage, analysis, manipulation and transmission
- ❖ The collected data may be stored directly in the system or may be transmitted to some other systems or it may be processed by the system or it may be deleted instantly after giving a meaningful representation



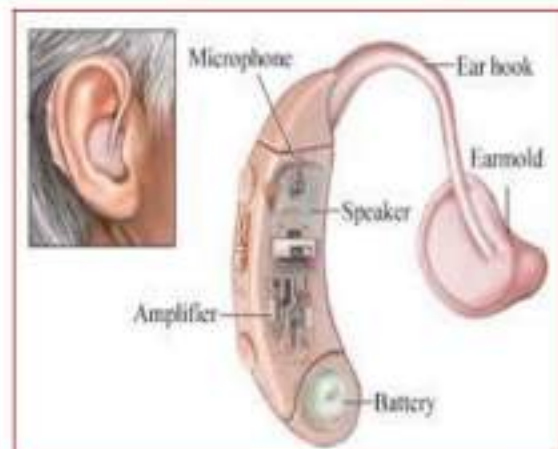
2. Data Communication:-

- Embedded Data communication systems are deployed in applications ranging from complex satellite communication systems to simple home networking systems
- Embedded Data communication systems are dedicated for data communication
- The data communication can happen through a wired interface (like Ethernet, RS-232C/USB/IEEE1394 etc) or wireless interface (like Wi-Fi, GSM/GPRS, Bluetooth, ZigBee etc)
- Network hubs, Routers, switches, Modems etc are typical examples for dedicated data transmission embedded systems



3. Data (Signal) Processing:-

- Embedded systems with Signal processing functionalities are employed in applications demanding signal processing like Speech coding, synthesis, audio video codec, transmission applications etc
- Computational intensive systems
- Employs Digital Signal Processors (DSPs)



4. Monitoring:-

- Embedded systems coming under this category are specifically designed for monitoring purpose
- They are used for determining the state of some variables using input sensors
- They cannot impose control over variables.
- Electro Cardiogram (ECG) machine for monitoring the heart beat of a patient is a typical example for this
- The sensors used in ECG are the different Electrodes connected to the patient's body
- Measuring instruments like Digital CRO, Digital Multi meter, Logic Analyzer etc used in Control & Instrumentation applications are also examples of embedded systems for monitoring purpose



5. Control:-

- Embedded systems with control functionalities are used for imposing control over some variables according to the changes in input variables
- Embedded system with control functionality contains both sensors and actuators
- Sensors are connected to the input port for capturing the changes in environmental variable or measuring variable
- The actuators connected to the output port are controlled according to the changes in input variable to put an impact on the controlling variable to bring the controlled variable to the specified range
- Air conditioner for controlling room temperature is a typical example for embedded system with „Control“ functionality
- Air conditioner contains a room temperature sensing element (sensor) which may be a thermistor and a handheld unit for setting up (feeding) the desired temperature
- The air compressor unit acts as the actuator. The compressor is controlled according to the current room temperature and the desired temperature set by the end user.



6. Application Specific User Interface:-

- Embedded systems which are designed for a specific application
- Contains Application Specific User interface (rather than general standard UI) like key board, Display units etc
- Aimed at a specific target group of users
- Mobile handsets, Control units in industrial applications etc are examples



Characteristics of Embedded systems:

(DEC2016, March-2017)

Embedded systems possess certain specific characteristics and these are unique to each Embedded system.

1. Application and domain specific
2. Reactive and Real Time
3. Operates in harsh environments
4. Distributed
5. Small Size and weight
6. Power concerns
7. Single-functioned
8. Complex functionality
9. Tightly-constrained
10. Safety-critical

1. Application and Domain Specific:-

- Each E.S has certain functions to perform and they are developed in such a manner to do the intended functions only.
- They cannot be used for any other purpose.
- Ex – The embedded control units of the microwave oven cannot be replaced with AC“S embedded control unit because the embedded control units of microwave oven and AC are specifically designed to perform certain specific tasks.

2. Reactive and Real Time:-

- E.S are in constant interaction with the real world through sensors and user-defined input devices which are connected to the input port of the system.
- Any changes in the real world are captured by the sensors or input devices in real time and the control algorithm running inside the unit reacts in a designed manner to bring the controlled output variables to the desired level.
- E.S produce changes in output in response to the changes in the input, so they are referred as reactive systems.
- Real Time system operation means the timing behavior of the system should be deterministic ie the system should respond to requests in a known amount of time.
- Example – E.S which are mission critical like flight control systems, Antilock Brake Systems (**ABS**) etc are Real Time systems.

3. Operates in Harsh Environment :-

- The design of E.S should take care of the operating conditions of the area where the system is going to implement.
- Ex – If the system needs to be deployed in a high temperature zone, then all the components used in the system should be of high temperature grade.
- Also proper shock absorption techniques should be provided to systems which are going to be commissioned in places subject to high shock.

4. Distributed: –

- It means that embedded systems may be a part of a larger system.
- Many numbers of such distributed embedded systems form a single large embedded control unit.
- Ex – Automatic vending machine. It contains a card reader, a vending unit etc. Each of them are independent embedded units but they work together to perform the overall vending function.

5. Small Size and Weight:-

- Product aesthetics (size, weight, shape, style, etc) is an important factor in choosing a product.
- It is convenient to handle a compact device than a bulky product.
- In embedded domain compactness is a significant deciding factor.

6. Power Concerns:-

- Power management is another important factor that needs to be considered in designing embedded systems.
- E.S should be designed in such a way as to minimize the heat dissipation by the system.

7. Single-functioned:- Dedicated to perform a single function**8. Complex functionality: -** We have to run sophisticated algorithms or multiple algorithms in some applications.**9. Tightly-constrained:-**

- Low cost, low power, small, fast, etc

10. Safety-critical:-

- Must not endanger human life and the environment

Quality Attributes of Embedded System: Quality attributes are the non-functional requirements that need to be documented properly in any system design. (DEC16, March-2017)

Quality attributes can be classified as

I. Operational quality attributes**II. Non-operational quality attributes.**

I. Operational Quality Attributes: The operational quality attributes represent the relevant quality attributes related to the embedded system when it is in the operational mode or online mode.

Operational Quality Attributes are:**1. Response :-**

- It is the measure of quickness of the system.
- It tells how fast the system is tracking the changes in input variables.
- Most of the E.S demands fast response which should be almost real time.

Ex – Flight control application.

2. Throughput :-

- It deals with the efficiency of a system.
- It can be defined as the rate of production or operation of a defined process over a stated period of time.
- The rates can be expressed in terms of products, batches produced or any other meaningful measurements.
- Ex – In case of card reader throughput means how many transactions the reader can perform in a minute or in an hour or in a day.
- Throughput is generally measured in terms of “Benchmark”.
- A Benchmark is a reference point by which something can be measured

3. Reliability :-

- It is a measure of how much we can rely upon the proper functioning of the system.
- Mean Time Between Failure (MTBF) and Mean Time To Repair (MTTR) are the terms used in determining system reliability.
- MTBF gives the frequency of failures in hours/weeks/months.
- MTTR specifies how long the system is allowed to be out of order following a failure.
- For embedded system with critical application need, it should be of the order of minutes.

4. Maintainability:-

- It deals with support and maintenance to the end user or client in case of technical issues and product failure or on the basis of a routine system checkup.
- Reliability and maintainability are complementary to each other.
- A more reliable system means a system with less corrective maintainability requirements and vice versa.
- Maintainability can be broadly classified into two categories
 1. Scheduled or Periodic maintenance (Preventive maintenance)
 2. Corrective maintenance to unexpected failures

5. Security:-

- Confidentiality, Integrity and availability are the three major measures of information security.
- Confidentiality deals with protection of data and application from unauthorized disclosure.
- Integrity deals with the protection of data and application from unauthorized modification.
- Availability deals with protection of data and application from unauthorized users.

6. Safety :-

- Safety deals with the possible damages that can happen to the operator, public and the environment due to the breakdown of an Embedded System.
- The breakdown of an embedded system may occur due to a hardware failure or a firmware failure.
- Safety analysis is a must in product engineering to evaluate the anticipated damages and determine the best course of action to bring down the consequences of damage to an acceptable level.

II. Non-Operational Quality Attributes: The quality attributes that needs to be addressed for the product not on the basis of operational aspects are grouped under this category.

1. Testability and Debug-ability:-

- Testability deals with how easily one can test the design, application and by which means it can be done.
- For an E.S testability is applicable to both the embedded hardware and firmware.
- Embedded hardware testing ensures that the peripherals and total hardware functions in the desired manner, whereas firmware testing ensures that the firmware is functioning in the expected way.
- Debug-ability is a means of debugging the product from unexpected behavior in the system
- Debug-ability is two level process
- 1.Hardware level 2.software level
- **1. Hardware level:** It is used for finding the issues created by hardware problems.
- **2. Software level:** It is employed for finding the errors created by the flaws in the software.

2. Evolvability :-

- It is a term which is closely related to Biology.
- It is referred as the non-heritable variation.
- For an embedded system evolvability refers to the ease with which the embedded product can be modified to take advantage of new firmware or hardware technologies.

3. Portability:-

- It is the measure of system independence.
- An embedded product is said to be portable if the product is capable of functioning in various environments, target processors and embedded operating systems.
- „Porting“ represents the migration of embedded firmware written for one target processor to a different target processor.

4. Time-to-Prototype and Market:-

- It is the time elapsed between the conceptualization of a product and the time at which the product is ready for selling.
- The commercial embedded product market is highly competitive and time to market the product is critical factor in the success of commercial embedded product.
- There may be multiple players in embedded industry who develop products of the same category (like mobile phone).

5. Per Unit Cost and Revenue:-

- Cost is a factor which is closely monitored by both end user and product manufacturer.
- Cost is highly sensitive factor for commercial products
- Any failure to position the cost of a commercial product at a nominal rate may lead to the failure of the product in the market.
- Proper market study and cost benefit analysis should be carried out before taking a decision on the per-unit cost of the embedded product.
- The ultimate aim of the product is to generate marginal profit so the budget and total cost should be properly balanced to provide a marginal profit.

SUMMARY

1. An embedded system is an electronic/electromechanical system designed to perform a specific function and is a combination of both hardware and firmware (software).
2. A general purpose computing system is a combination of generic hardware and general purpose operating system for executing a variety of applications, whereas an embedded system is a combination of special purpose hardware and embedded OS/firmware for executing a specific set of applications.
3. Apollo Guidance Computer (AGC) is the first recognized modern embedded system and Autonetics D-17, the guidance computer for the Minuteman-I missile, was the first mass produced embedded system.
4. Based on the complexity and performance requirements, embedded systems are classified into small-scale, medium-scale and large-scale/complex.
5. The presences of embedded system vary from simple electronic system toys to complex flight and missile control systems.
6. Embedded systems are designed to serve the purpose of any one or combination of data collection/storage/representation, data processing, monitoring, control or application specific user interface.
7. Wearable devices refer to embedded systems which are incorporated into accessories and apparels. It envisions the bonding of embedded technology in our day to day lives.

OBJECTIVE QUESTIONS

1. Embedded systems are
 - (a) General Purpose
 - (b) Special Purpose
2. Embedded system is
 - (a) An electronic system
 - (b) A pure mechanical system
 - (c) An electro-mechanical system
 - (d) (a) or (c)
3. Which of the following is not true about embedded systems?
 - (a) Built around specialized hardware
 - (b) Always contain an operating system
 - (c) Execution behavior may be deterministic
 - (d) All of these
 - (e) none of these
4. Which of the following is not an example of small scale embedded system?
 - (a) Electronic Barbie doll
 - (b) Simple calculator
 - (c) Cell Phone
 - (d) Electronic toy car

5. The first recognized modern embedded system is
- (a) Apple computer
 - (b) Apollo Guidance Computer
 - (c) Calculator
 - (d) Radio navigation system
6. The first mass produced embedded system is
- (a) Minuteman-I
 - (b) Minuteman-II
 - (c) Autonetics D17
 - (d) Apollo Guidance Computer
7. Which of the following is (are) an intended purpose of embedded systems?
- (a) Data collection
 - (b) Data processing
 - (c) Data communication
 - (d) All of these
 - (e) None of these
8. Which of the following is an example of an embedded system for data communication?
- (a) USB mass storage device
 - (b) Network router
 - (c) Digital camera
 - (d) Music player
 - (e) All of these
 - (f) None of these
9. A digital multimeter is an example of embedded system for
- (a) Data communication
 - (b) Monitoring
 - (c) Control
 - (d) All of these
 - (e) None of these
10. Which of the following is an example of an embedded system for signal processing?
- (a) Apple iPOD
 - (b) Sandisk USB mass storage device
 - (c) both a and b
 - (d) None of these

Reference Text Books:-

1. Introduction to Embedded Systems – Shibu K.V Mc Graw Hill

2. Embedded System Design-Raj Kamal TMH

Embedded Systems

III - ESE ::



UNIT-3

The Typical Embedded System

N.SURESH
Department of ECE

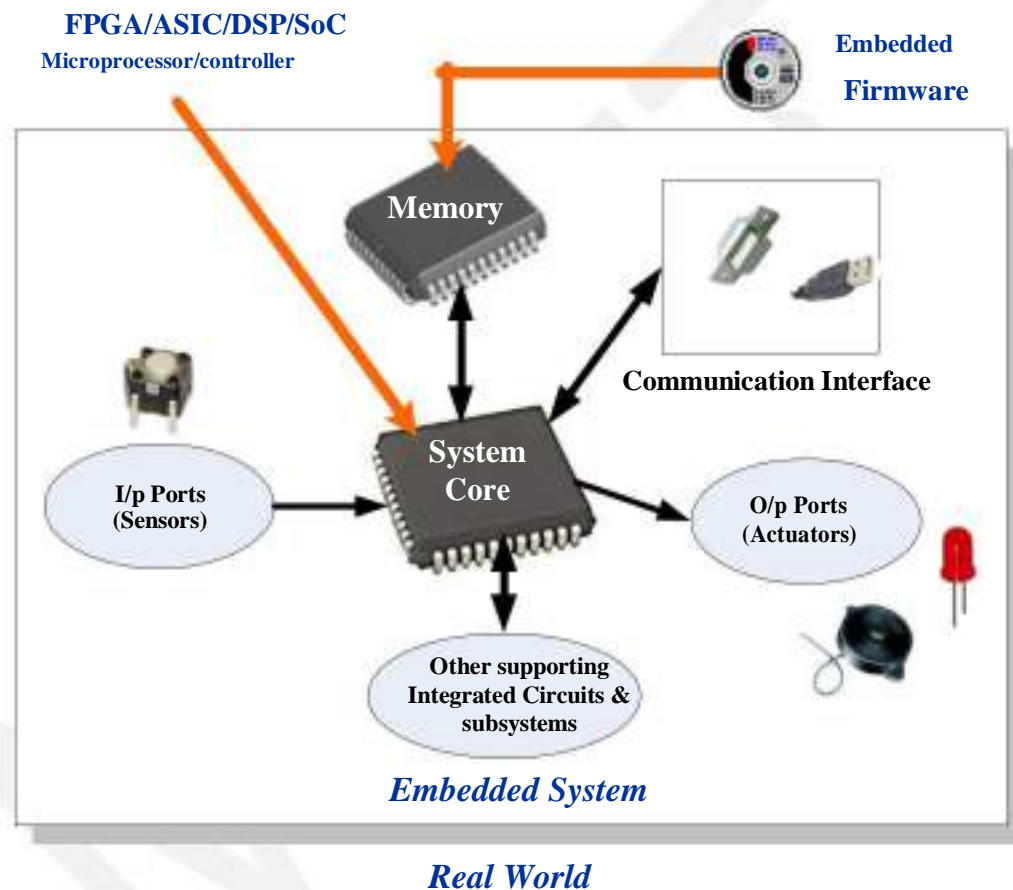


**MALLA REDDY COLLEGE OF
ENGINEERING & TECHNOLOGY**

Permanently Affiliated to JNTUH and Approved by AICTE, New Delhi

ELEMENTS OF EMBEDDED SYSTEMS:

An embedded system is a combination of 3 things, Hardware Software Mechanical Components and it is supposed to do one specific task only. A typical embedded system contains a single chip controller which acts as the master brain of the system. Diagrammatically an embedded system can be represented as follows:



Embedded systems are basically designed to regulate a physical variable (such as Microwave Oven) or to manipulate the state of some devices by sending some signals to the actuators or devices connected to the output port system (such as temperature in Air Conditioner), in response to the input signal provided by the end users or sensors which are connected to the input ports. Hence the embedded systems can be viewed as a reactive

system. The control is achieved by processing the information coming from the sensors and user interfaces and controlling some actuators that regulate the physical variable.

Keyboards, push button, switches, etc. are Examples of common user interface input devices and LEDs, LCDs, Piezoelectric buzzers, etc examples for common user interface output devices for a typical embedded system. The requirement of type of user interface changes from application to application based on domain.

Some embedded systems do not require any manual intervention for their operation. They automatically sense the input parameters from real world through sensors which are connected at input port. The sensor information is passed to the processor after signal conditioning and digitization. The core of the system performs some predefined operations on input data with the help of embedded firmware in the system and sends some actuating signals to the actuator connect connected to the output port of the system.

The memory of the system is responsible for holding the code (control algorithm and other important configuration details). There are two types of memories are used in any embedded system. Fixed memory (ROM) is used for storing code or program. The user cannot change the firmware in this type of memory. The most common types of memories used in embedded systems for control algorithm storage are OTP,PROM,UVEPROM,EEPROM and FLASH

An embedded system without code (i.e. the control algorithm) implemented memory has all the peripherals but is not capable of making decisions depending on the situational as well as real world changes.

Memory for implementing the code may be present on the processor or may be implemented as a separate chip interfacing the processor

In a controller based embedded system, the controller may contain internal memory for storing code Such controllers are called Micro-controllers with on-chip ROM, eg. Atmel AT89C51.

The Core of the Embedded Systems: The core of the embedded system falls into any one of the following categories.

- ❑ **General Purpose and Domain Specific Processors**
 - Microprocessors
 - Microcontrollers
 - Digital Signal Processors
- ❑ **Programmable Logic Devices (PLDs)**
- ❑ **Application Specific Integrated Circuits (ASICs)**
- ❑ **Commercial off the shelf Components (COTS)**

GENERAL PURPOSE AND DOMAIN SPECIFIC PROCESSOR:

- Almost 80% of the embedded systems are processor/ controller based.
- The processor may be microprocessor or a microcontroller or digital signal processor, depending on the domain and application.

Microprocessor:

- A silicon chip representing a Central Processing Unit (CPU), which is capable of performing arithmetic as well as logical operations according to a pre-defined set of Instructions, which is specific to the manufacturer
- In general the CPU contains the Arithmetic and Logic Unit (ALU), Control Unit and Working registers
- Microprocessor is a dependant unit and it requires the combination of other hardware like Memory, Timer Unit, and Interrupt Controller etc for proper functioning.
- Intel claims the credit for developing the first Microprocessor unit Intel 4004, a 4 bit processor which was released in Nov 1971
- Developers of microprocessors.
 - Intel – Intel 4004 – November 1971(4-bit)
 - Intel – Intel 4040.
 - Intel – Intel 8008 – April 1972.
 - Intel – Intel 8080 – April 1974(8-bit).
 - Motorola – Motorola 6800.
 - Intel – Intel 8085 – 1976.
 - Zilog - Z80 – July 1976

Microcontroller:

- ❖ A highly integrated silicon chip containing a CPU, scratch pad RAM, Special and General purpose Register Arrays, On Chip ROM/FLASH memory for program storage, Timer and Interrupt control units and dedicated I/O ports
- ❖ Microcontrollers can be considered as a super set of Microprocessors
- ❖ Microcontroller can be general purpose (like Intel 8051, designed for generic applications and domains) or application specific (Like Automotive AVR from Atmel Corporation. Designed specifically for automotive applications)
- ❖ Since a microcontroller contains all the necessary functional blocks for independent working, they found greater place in the embedded domain in place of microprocessors
- ❖ Microcontrollers are cheap, cost effective and are readily available in the market
- ❖ Texas Instruments TMS 1000 is considered as the world's first microcontroller

Microprocessor Vs Microcontroller:

Microprocessor	Microcontroller
A silicon chip representing a Central Processing Unit (CPU), which is capable of performing arithmetic as well as logical operations according to a pre-defined set of Instructions	A microcontroller is a highly integrated chip that contains a CPU, scratch pad RAM, Special and General purpose Register Arrays, On Chip ROM/FLASH memory for program storage, Timer and Interrupt control units and dedicated I/O ports
It is a dependent unit. It requires the combination of other chips like Timers, Program and data memory chips, Interrupt controllers etc for functioning	It is a self contained unit and it doesn't require external Interrupt Controller, Timer, UART etc for its functioning
Most of the time general purpose in design and operation	Mostly application oriented or domain specific
Doesn't contain a built in I/O port. The I/O Port functionality needs to be implemented with the help of external Programmable Peripheral Interface Chips like 8255	Most of the processors contain multiple built-in I/O ports which can be operated as a single 8 or 16 or 32 bit Port or as individual port pins
Targeted for high end market where performance is important	Targeted for embedded market where performance is not so critical (At present this demarcation is invalid)
Limited power saving options compared to microcontrollers	Includes lot of power saving features

General Purpose Processor (GPP) Vs Application Specific Instruction Set Processor (ASIP)

- ❖ General Purpose Processor or GPP is a processor designed for general computational tasks
- ❖ GPPs are produced in large volumes and targeting the general market. Due to the high volume production, the per unit cost for a chip is low compared to ASIC or other specific ICs
- ❖ A typical general purpose processor contains an Arithmetic and Logic Unit (ALU) and Control Unit (CU)
- ❖ Application Specific Instruction Set processors (ASIPs) are processors with architecture and instruction set optimized to specific domain/application requirements like Network processing, Automotive, Telecom, media applications, digital signal processing, control applications etc.
- ❖ ASIPs fill the architectural spectrum between General Purpose Processors and Application Specific Integrated Circuits (ASICs)
- ❖ The need for an ASIP arises when the traditional general purpose processor are unable to meet the increasing application needs
- ❖ Some Microcontrollers (like Automotive AVR, USB AVR from Atmel), System on Chips, Digital Signal Processors etc are examples of Application Specific Instruction Set Processors (ASIPs)
- ❖ ASIPs incorporate a processor and on-chip peripherals, demanded by the application requirement, program and data memory

Digital Signal Processors (DSPs):

- Powerful special purpose 8/16/32 bit microprocessors designed specifically to meet the computational demands and power constraints of today's embedded audio, video, and communications applications
- Digital Signal Processors are 2 to 3 times faster than the general purpose microprocessors in signal processing applications
- DSPs implement algorithms in hardware which speeds up the execution whereas general purpose processors implement the algorithm in firmware and the speed of execution depends primarily on the clock for the processors
- DSP can be viewed as a microchip designed for performing high speed computational operations for „addition“, „subtraction“, „multiplication“ and „division“

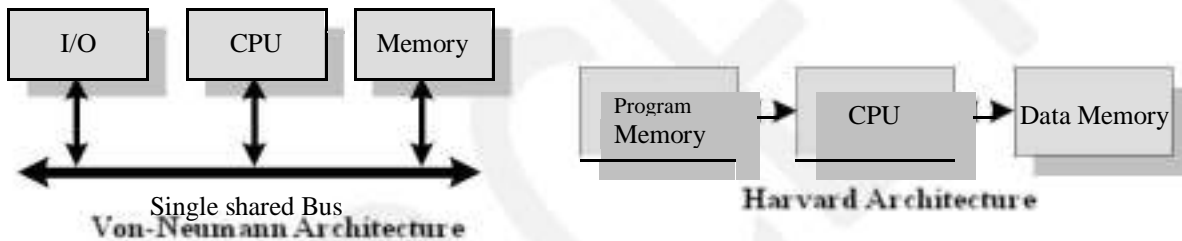
- A typical Digital Signal Processor incorporates the following key units
 - ❖ Program Memory
 - ❖ Data Memory
 - ❖ Computational Engine
 - ❖ I/O Unit
- Audio video signal processing, telecommunication and multimedia applications are typical examples where DSP is employed

RISC V/s CISC Processors/Controllers:

RISC	CISC
Lesser no. of instructions	Greater no. of Instructions
Instruction Pipelining and increased execution speed	Generally no instruction pipelining feature
Orthogonal Instruction Set (Allows each instruction to operate on any register and use any addressing mode)	Non Orthogonal Instruction Set (All instructions are not allowed to operate on any register and use any addressing mode. It is instruction specific)
Operations are performed on registers only, the only memory operations are load and store	Operations are performed on registers or memory depending on the instruction
Large number of registers are available	Limited no. of general purpose registers
Programmer needs to write more code to execute a task since the instructions are simpler ones	. A programmer can achieve the desired functionality with a single instruction which in turn provides the effect of using more simpler single instructions in RISC
Single, Fixed length Instructions	Variable length Instructions
Less Silicon usage and pin count	More silicon usage since more additional decoder logic is required to implement the complex instruction decoding.
With Harvard Architecture	Can be Harvard or Von-Neumann Architecture

Harvard V/s Von-Neumann Processor/Controller Architecture

- The terms Harvard and Von-Neumann refers to the processor architecture design.
- Microprocessors/controllers based on the **Von-Neumann** architecture shares a single common bus for fetching both instructions and data. Program instructions and data are stored in a common main memory
- Microprocessors/controllers based on the **Harvard** architecture will have separate data bus and instruction bus. This allows the data transfer and program fetching to occur simultaneously on both buses
- With Harvard architecture, the data memory can be read and written while the program memory is being accessed. These separated data memory and code memory buses allow one instruction to execute while the next instruction is fetched (“Pre-fetching”)

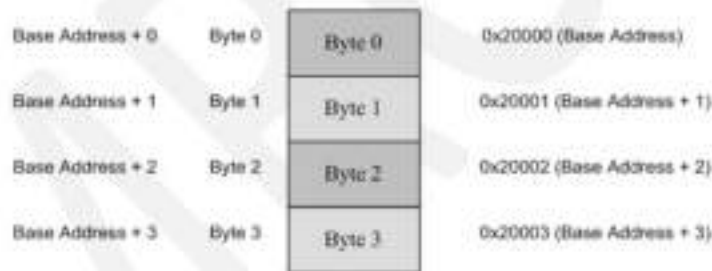
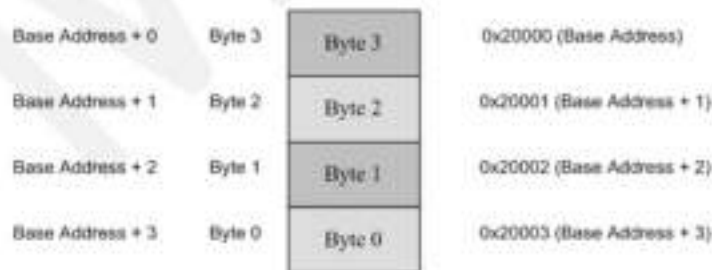


Harvard V/s Von-Neumann Processor/Controller Architecture:

Harvard Architecture	Von-Neumann Architecture
Separate buses for Instruction and Data fetching	Single shared bus for Instruction and Data fetching
Easier to Pipeline, so high performance can be achieved	Low performance Compared to Harvard Architecture
Comparatively high cost	Cheaper
No memory alignment problems	Allows self modifying codes [†]
Since data memory and program memory are stored physically in different locations, no chances for accidental corruption of program memory	Since data memory and program memory are stored physically in same chip, chances for accidental corruption of program memory

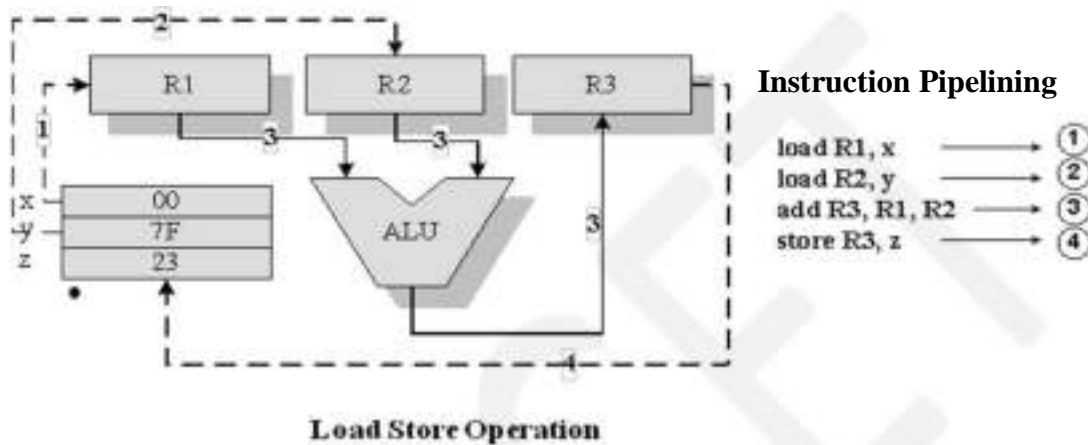
Big-endian V/s Little-endian processors:

- ✓ Endianness specifies the order in which the data is stored in the memory by processor operations in a multi byte system (Processors whose word size is greater than one byte). Suppose the word length is two byte then data can be stored in memory in two different ways
 - Higher order of data byte at the higher memory and lower order of data byte at location just below the higher memory
 - Lower order of data byte at the higher memory and higher order of data byte at location just below the higher memory
- ✓ *Little-endian* means the lower-order byte of the data is stored in memory at the lowest address, and the higher-order byte at the highest address. (The little end comes first)
- ✓ *Big-endian* means the higher-order byte of the data is stored in memory at the lowest address, and the lower-order byte at the highest address. (The big end comes first.)

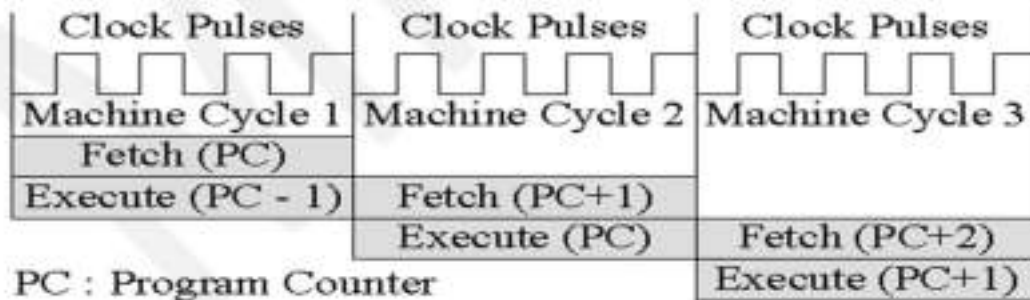
Big-endian V/s Little-endian processors**Little-endian Operation****Big-endian Operation**

Load Store Operation & Instruction Pipelining:

The RISC processor instruction set is orthogonal and it operates on registers. The memory access related operations are performed by the special instructions *load* and *store*. If the operand is specified as memory location, the content of it is loaded to a register using the *load* instruction. The instruction *store* stores data from a specified register to a specified memory location



- The conventional instruction execution by the processor follows the fetch-decode-execute sequence
- The „fetch“ part fetches the instruction from program memory or code memory and the decode part decodes the instruction to generate the necessary control signals



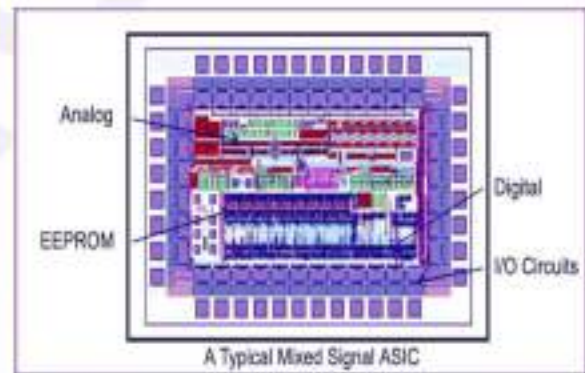
The Single stage pipelining concept

- The execute stage reads the operands, perform ALU operations and stores the result. In conventional program execution, the fetch and decode operations are performed in sequence

- During the decode operation the memory address bus is available and if it possible to effectively utilize it for an instruction fetch, the processing speed can be increased
- In its simplest form instruction pipelining refers to the overlapped execution of instructions

Application Specific Integrated Circuit (ASIC):

- A microchip designed to perform a specific or unique application. It is used as replacement to conventional general purpose logic chips.
- ASIC integrates several functions into a single chip and thereby reduces the system development cost
- Most of the ASICs are proprietary products. As a single chip, ASIC consumes very small area in the total system and thereby helps in the design of smaller systems with high capabilities/functionalities.
- ASICs can be pre-fabricated for a special application or it can be custom fabricated by using the components from a re-usable “*building block*” library of components for a particular customer application



- Fabrication of ASICs requires a non-refundable initial investment (Non Recurring Engineering (NRE) charges) for the process technology and configuration expenses
- If the Non-Recurring Engineering Charges (NRE) is born by a third party and the Application Specific Integrated Circuit (ASIC) is made openly available in the market, the ASIC is referred as Application Specific Standard Product (ASSP)
- The ASSP is marketed to multiple customers just as a general-purpose product , but to a smaller number of customers since it is for a specific application.

- Some ASICs are proprietary products , the developers are not interested in revealing the internal details.

Programmable Logic Devices (PLDs):

- ❖ Logic devices provide specific functions, including device-to-device interfacing, data communication, signal processing, data display, timing and control operations, and almost every other function a system must perform.
- ❖ Logic devices can be classified into two broad categories - Fixed and Programmable. The circuits in a fixed logic device are permanent, they perform one function or set of functions - once manufactured, they cannot be changed
- ❖ Programmable logic devices (PLDs) offer customers a wide range of logic capacity, features, speed, and voltage characteristics - and these devices can be re-configured to perform any number of functions at any time
- ❖ Designers can use inexpensive software tools to quickly develop, simulate, and test their logic designs in PLD based design. The design can be quickly programmed into a device, and immediately tested in a live circuit
- ❖ PLDs are based on re-writable memory technology and the device is reprogrammed to change the design

Programmable Logic Devices (PLDs) – CPLDs and FPGA

- Field Programmable Gate Arrays (FPGAs) and Complex Programmable Logic Devices (CPLDs) are the two major types of programmable logic devices

FPGA:

- FPGA is an IC designed to be configured by a designer after manufacturing.
- FPGAs offer the highest amount of logic density, the most features, and the highest performance.
- Logic gate is Medium to high density ranging from **1K to 500K** system gates

- These advanced FPGA devices also offer features such as built-in hardwired processors (such as the IBM Power PC), substantial amounts of memory, clock management systems, and support for many of the latest, very fast device-to-device signaling technologies

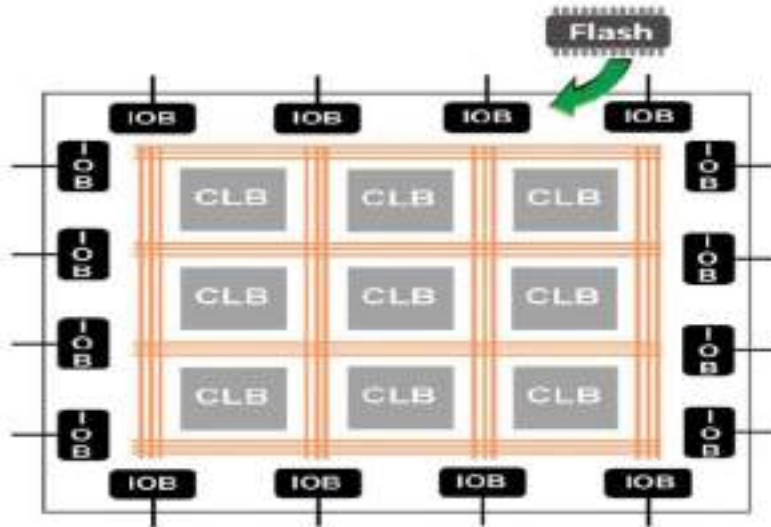


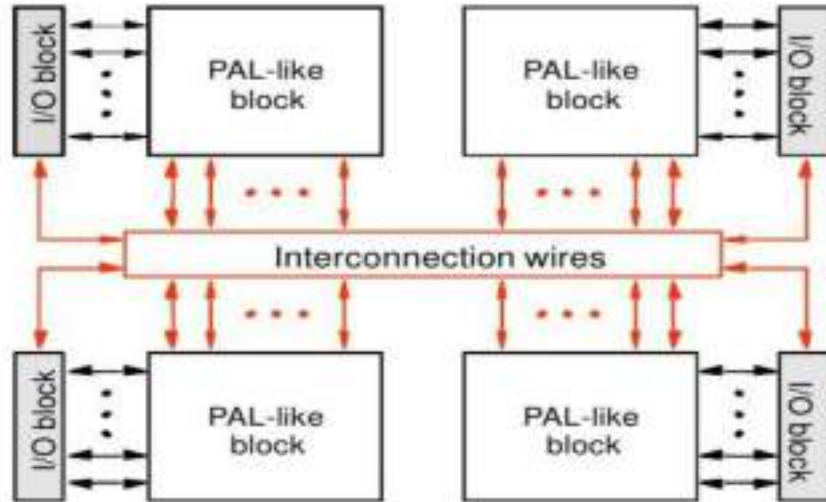
Figure: FPGA Architecture

- These advanced FPGA devices also offer features such as built-in hardwired processors, substantial amounts of memory, clock management systems, and support for many of the latest, very fast device-to-device signaling technologies.
- FPGAs are used in a wide variety of applications ranging from data processing and storage, to instrumentation, telecommunications, and digital signal processing

CPLD:

- A **complex programmable logic device (CPLD)** is a programmable logic device with complexity between that of PALs and FPGAs, and architectural features of both.
- CPLDs, by contrast, offer much smaller amounts of logic - up to about 10,000 gates.
- CPLDs offer very predictable timing characteristics and are therefore ideal for critical control applications.

► Structure of a CPLD



- CPLDs such as the Xilinx **CoolRunner** series also require extremely low amounts of power and are very inexpensive, making them ideal for cost-sensitive, battery-operated, portable applications such as mobile phones and digital handheld assistants.

ADVANTAGES OF PLDs:

- PLDs offer customer much more flexibility during design cycle
- PLDSs do not require long lead times for prototype or production-the PLDs are already on a distributor's self and ready for shipment
- PLDs do not require customers to pay for large NRE costs and purchase expensive mask sets
- PLDs allow customers to order just the number of parts required when they need them. allowing them to control inventory.
- PLDs are reprogrammable even after a piece of equipment is shipped to a customer.
- The manufacturers able to add new features or upgrade the PLD based products that are in the field by uploading new programming file

Commercial off the Shelf Component (COTS):

- A Commercial off-the-shelf (COTS) product is one which is used „as-is“
- COTS products are designed in such a way to provide easy integration and interoperability with existing system components

- Typical examples for the COTS hardware unit are Remote Controlled Toy Car control unit including the RF Circuitry part, High performance, high frequency microwave electronics (2 to 200 GHz), High bandwidth analog-to-digital converters, Devices and components for operation at very high temperatures, Electro-optic IR imaging arrays, UV/IR Detectors etc



- A COTS component in turn contains a General Purpose Processor (GPP) or Application Specific Instruction Set Processor (ASIP) or Application Specific Integrated Chip (ASIC)/Application Specific Standard Product (ASSP) or Programmable Logic Device (PLD)



- The major advantage of using COTS is that they are readily available in the market, cheap and a developer can cut down his/her development time to a great extent.
- There is no need to design the module yourself and write the firmware .
- Everything will be readily supplied by the COTs manufacturer.

- The major problem faced by the end-user is that there are no operational and manufacturing standards.
- The major drawback of using COTs component in embedded design is that the manufacturer may withdraw the product or discontinue the production of the COTs at any time if rapid change in technology

This problem adversely affect a commercial manufacturer of the embedded system which makes use of the specific COTs

Sensors & Actuators:

- Embedded system is in constant interaction with the real world
- Controlling/monitoring functions executed by the embedded system is achieved in accordance with the changes happening to the Real World.
- The changes in the system environment or variables are detected by the sensors connected to the input port of the embedded system.
- If the embedded system is designed for any controlling purpose, the system will produce some changes in controlling variable to bring the controlled variable to the desired value.
- It is achieved through an actuator connected to the out port of the embedded system.

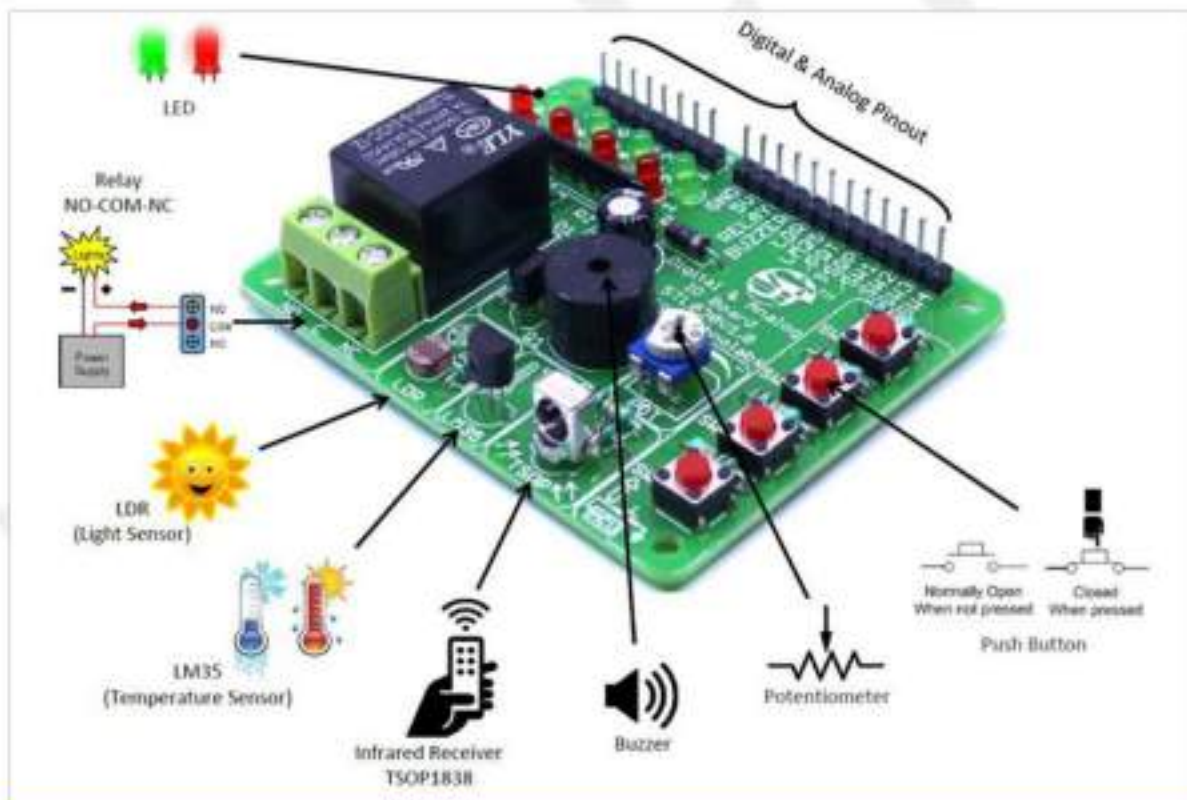
Sensor:

- A transducer device which converts energy from one form to another for any measurement or control purpose. Sensors acts as input device
- Eg. Hall Effect Sensor which measures the distance between the cushion and magnet in the Smart Running shoes from adidas
- **Example: IR, humidity , PIR(passive infra red) , ultrasonic , piezoelectric , smoke sensors**



Actuator:

- A form of transducer device (mechanical or electrical) which converts signals to corresponding physical action (motion). Actuator acts as an output device
- Eg. Micro motor actuator which adjusts the position of the cushioning element in the Smart Running shoes from adidas



Silicon TechnoLabs Digital Analog Arduino Starter kit

Communication Interface :

- Communication interface is essential for communicating with various subsystems of the embedded system and with the external world
- The communication interface can be viewed in two different perspectives; namely;

1. Device/board level communication interface (Onboard Communication Interface)

2. Product level communication interface (External Communication Interface)

1. Device/board level communication interface (Onboard Communication Interface):

The communication channel which interconnects the various components within an embedded product is referred as Device/board level communication interface (Onboard Communication Interface)

Examples: Serial interfaces like I2C, SPI, UART, 1-Wire etc and Parallel bus interface

2. Product level communication interface (External Communication Interface):

- The „Product level communication interface“ (External Communication Interface) is responsible for data transfer between the embedded system and other devices or modules
- The external communication interface can be either wired media or wireless media and it can be a serial or parallel interface.

Examples for wireless communication interface: Infrared (IR), Bluetooth (BT), Wireless LAN (Wi-Fi), Radio Frequency waves (RF), GPRS etc.

Examples for wired interfaces: RS-232C/RS-422/RS 485, USB, Ethernet (TCP-IP), IEEE 1394 port, Parallel port etc.

1. Device/board level or On board communication interface:

The communication channel which interconnects the various components within an embedded product is referred as Device/board level communication interface (Onboard Communication Interface)

These are classified into

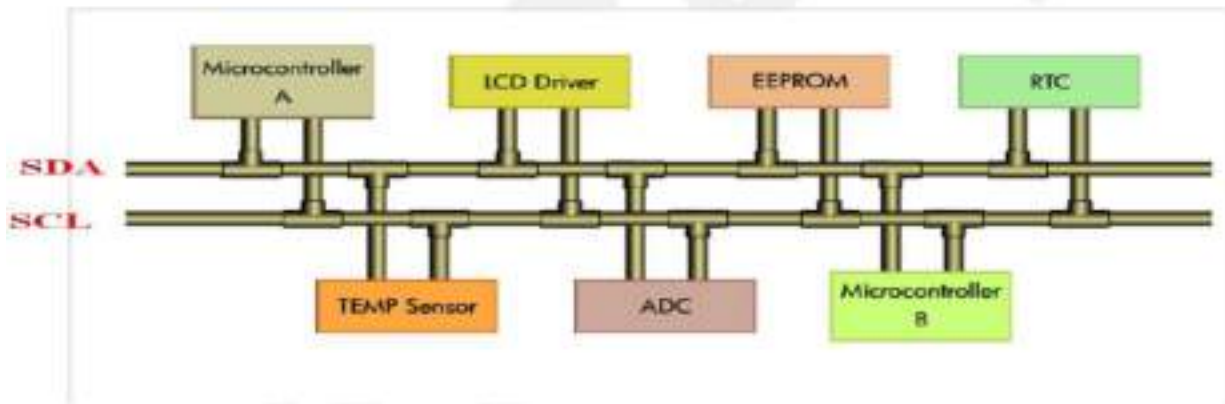
1.1 I2C (Inter Integrated Circuit) Bus

1.2 SPI(Serial Peripheral Interface) Bus

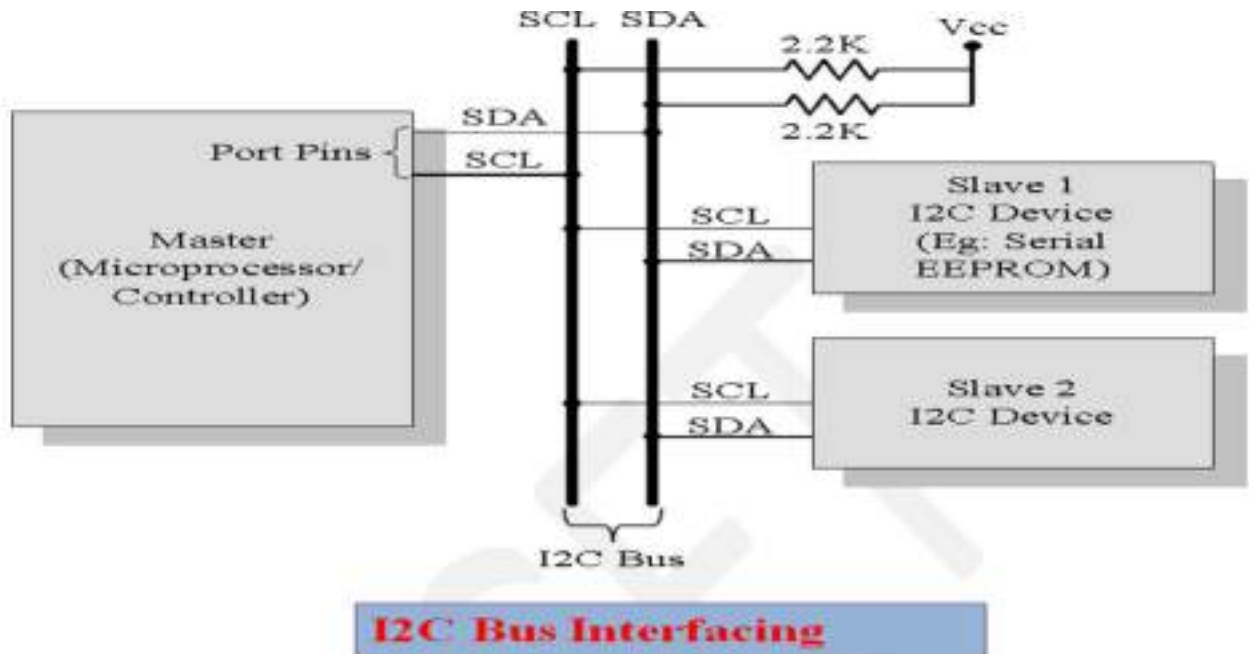
1.3 Parallel Interface

1.1 I2C (Inter Integrated Circuit)Bus :

- Inter Integrated Circuit Bus (I2C - Pronounced „I square C“) is a synchronous bi-directional half duplex (one-directional communication at a given point of time) two wire serial interface bus.
- The concept of I2C bus was developed by „Philips Semiconductors“ in the early 1980“s.
- The original intention of I2C was to provide an easy way of connection between a microprocessor/microcontroller system and the peripheral chips in Television sets.
- The I2C bus is comprised of two bus lines, namely; Serial Clock – SCL and Serial Data – SDA.



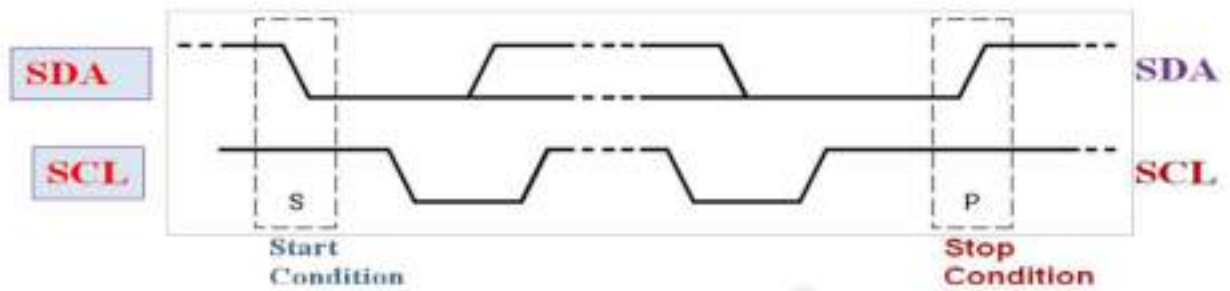
- SCL line is responsible for generating synchronization clock pulses and SDA is responsible for transmitting the serial data across devices.
- I2C bus is a shared bus system to which many number of I2C devices can be connected.
- Devices connected to the I2C bus can act as either „Master“ device or „Slave“ device



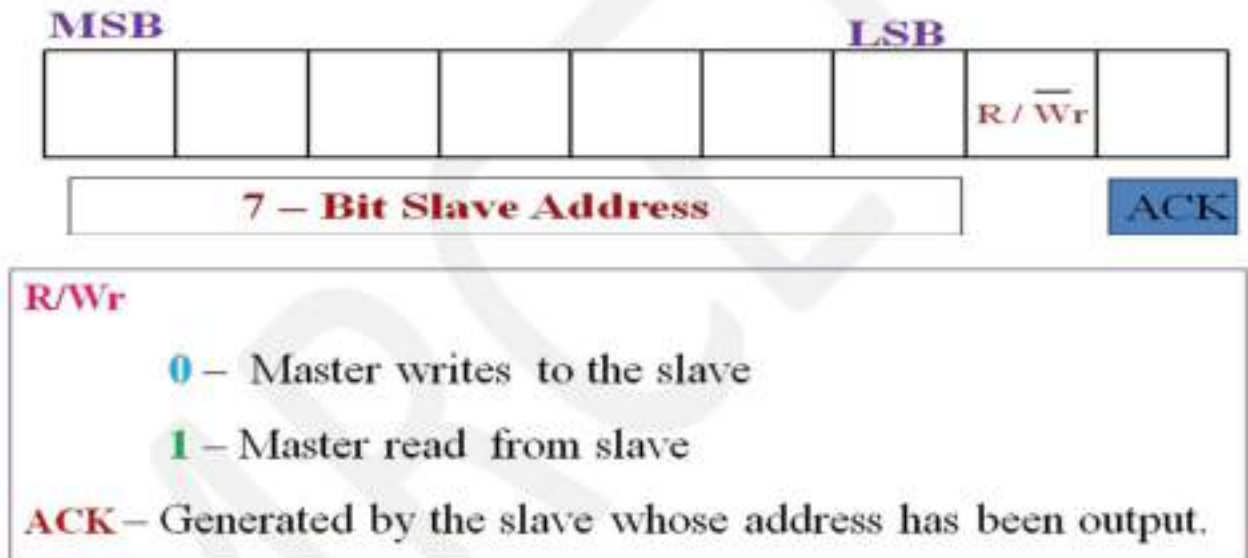
- The „Master“ device is responsible for controlling the communication by initiating/terminating data transfer, sending data and generating necessary synchronization clock pulses.
- „Slave“ devices wait for the commands from the master and respond upon receiving the commands.
- „Master“ and „Slave“ devices can act as either transmitter or receiver.
- Regardless whether a master is acting as transmitter or receiver, the synchronization clock signal is generated by the „Master“ device only.
- I2C supports multi masters on the same bus.

The sequence of operation for communicating with an I2C slave device is:

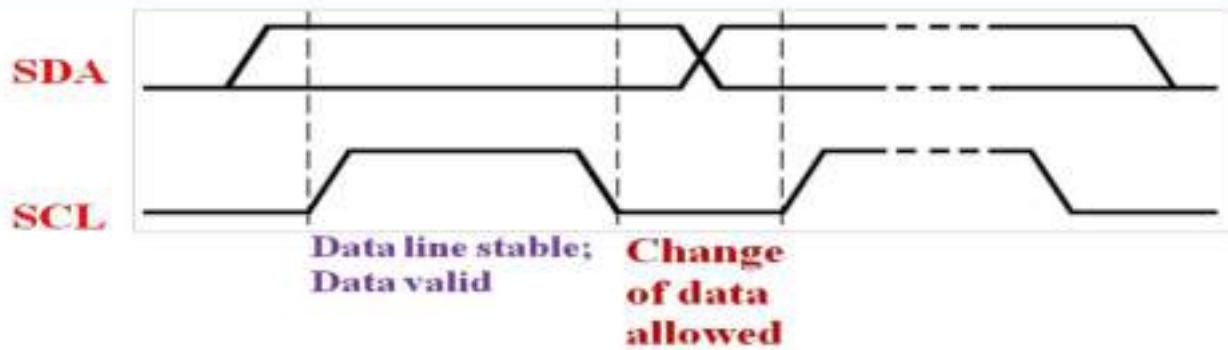
1. Master device pulls the clock line (SCL) of the bus to „HIGH“
2. Master device pulls the data line (SDA) „LOW“, when the SCL line is at logic „HIGH“ (This is the „Start“ condition for data transfer)



- Master sends the address (7 bit or 10 bit wide) of the „Slave“ device to which it wants to communicate, over the SDA line.



- Clock pulses are generated at the SCL line for synchronizing the bit reception by the slave device.
- The MSB of the data is always transmitted first.
- The data in the bus is valid during the „HIGH“ period of the clock signal
- In normal data transfer, the data line only changes state when the clock is low .



8. Master waits for the acknowledgement bit from the slave device whose address is sent on the bus along with the Read/Write operation command.
9. Slave devices connected to the bus compares the address received with the address assigned to them
10. The Slave device with the address requested by the master device responds by sending an acknowledge bit (Bit value =1) over the SDA line
11. Upon receiving the acknowledge bit, master sends the 8bit data to the slave device over SDA line, if the requested operation is „Write to device“.

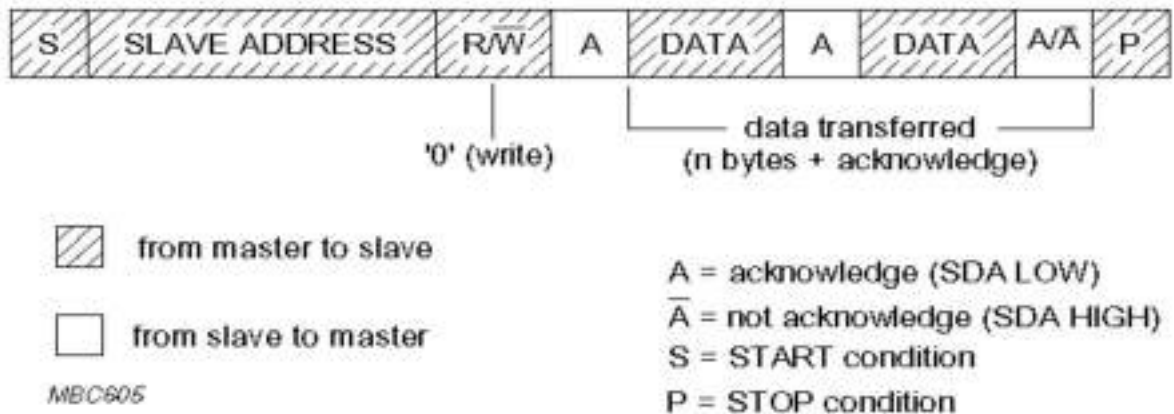


Figure: Master writing to a Slave.

12. If the requested operation is „Read from device“, the slave device sends data to the master over the SDA line.

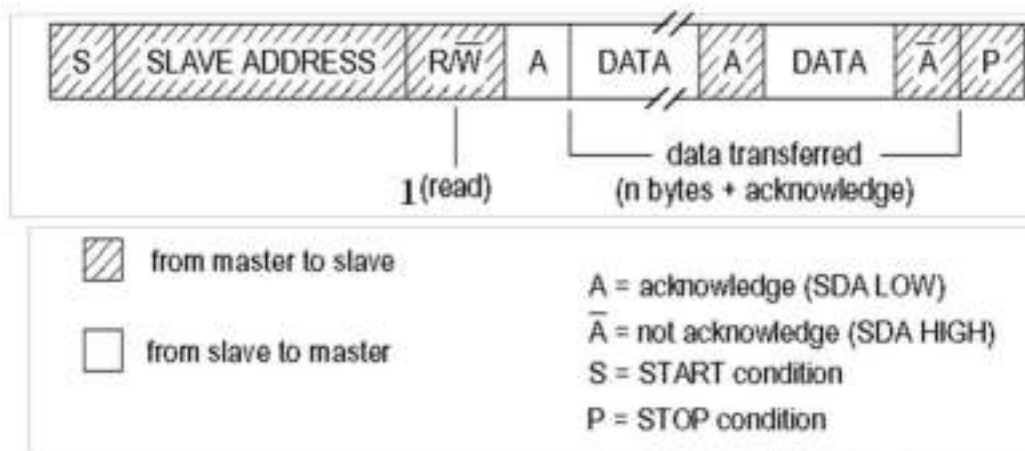


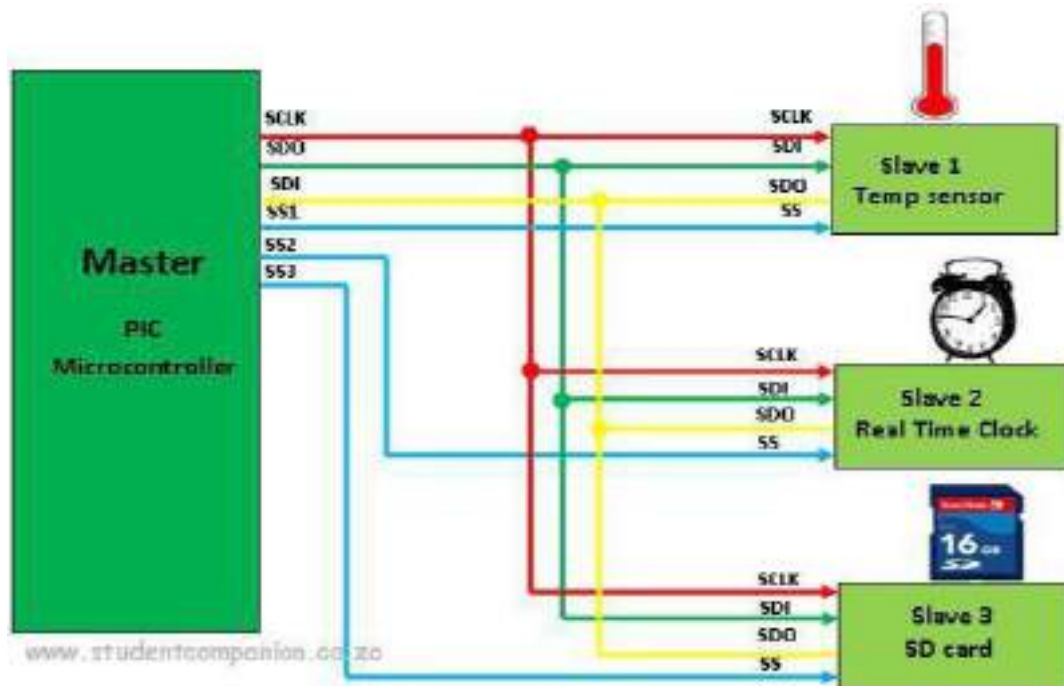
Figure: Master reading from a Slave

13. Master waits for the acknowledgement bit from the device upon byte transfer complete for a write operation and sends an acknowledge bit to the slave device for a read operation
14. Master terminates the transfer by pulling the SDA line „HIGH“ when the clock line SCL is at logic „HIGH“ (Indicating the „STOP“ condition).

1.2 Serial Peripheral Interface (SPI) Bus:

- The Serial Peripheral Interface Bus (SPI) is a synchronous bi-directional full duplex four wire serial interface bus.
- The concept of SPI is introduced by Motorola.
- SPI is a single master multi-slave system.
- It is possible to have a system where more than one SPI device can be master, provided the condition only one master device is active at any given point of time, is satisfied.

- SPI is used to send data between Microcontrollers and small peripherals such as shift registers, sensors, and SD cards.



- SPI requires four signal lines for communication. They are:

Master Out Slave In (MOSI): Signal line carrying the data from master to slave device.

It is also known as Slave Input/Slave Data In (SI/SDI)

Master In Slave Out (MISO): Signal line carrying the data from slave to master device.

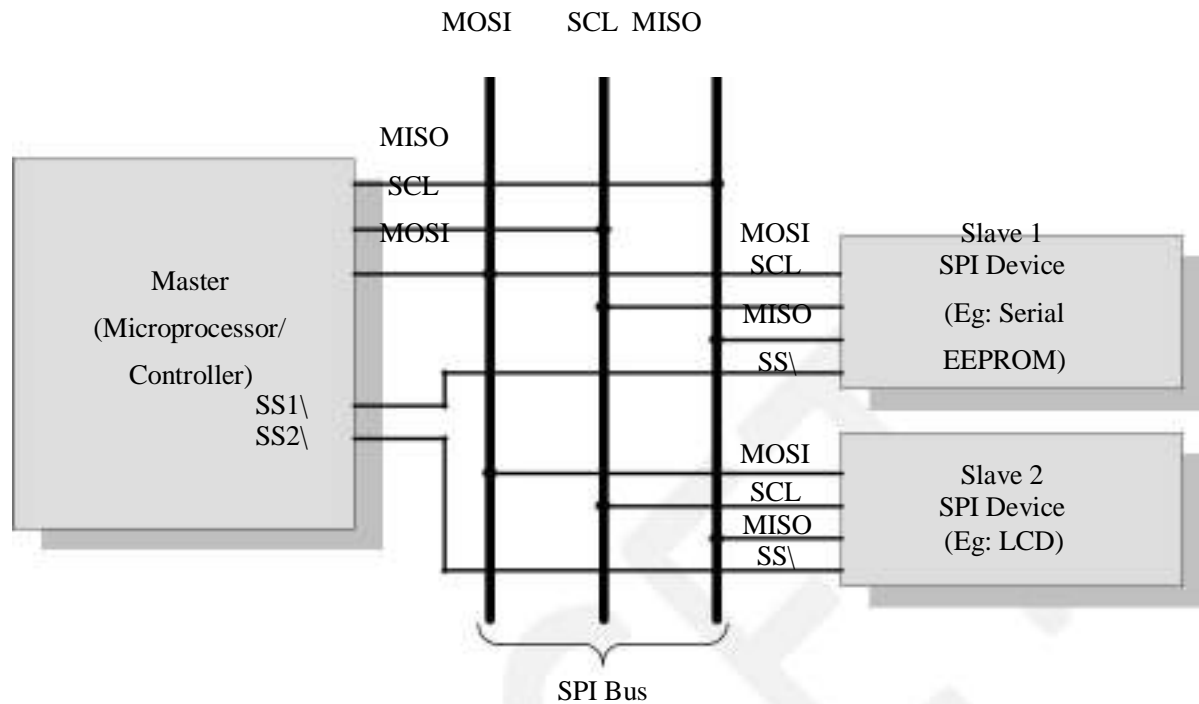
It is also known as Slave Output (SO/SDO)

Serial Clock (SCLK): Signal line carrying the clock signals

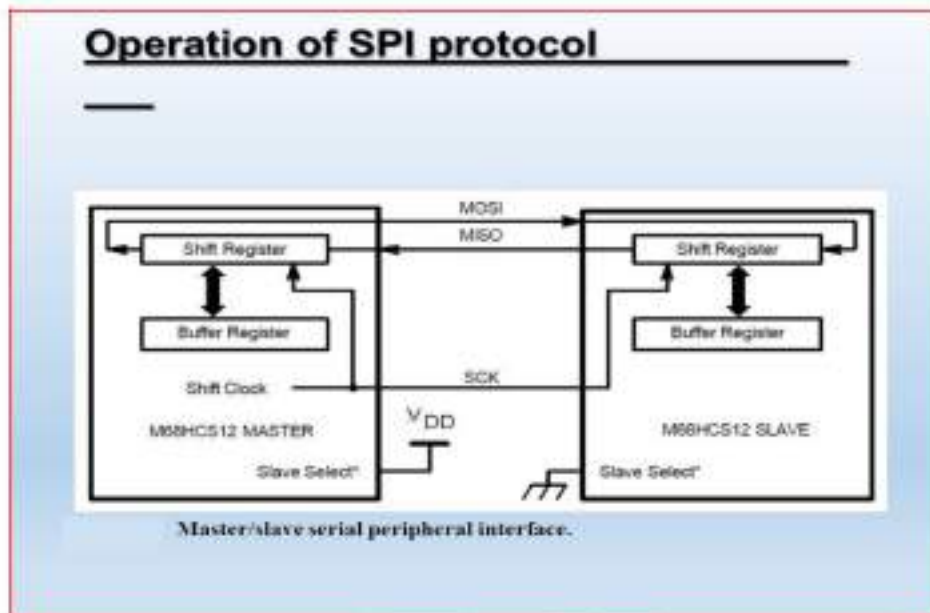
Slave Select (SS): Signal line for slave device select. It is an active low signal.

- The master device is responsible for generating the clock signal.
- Master device selects the required slave device by asserting the corresponding slave device's slave select signal „LOW“.

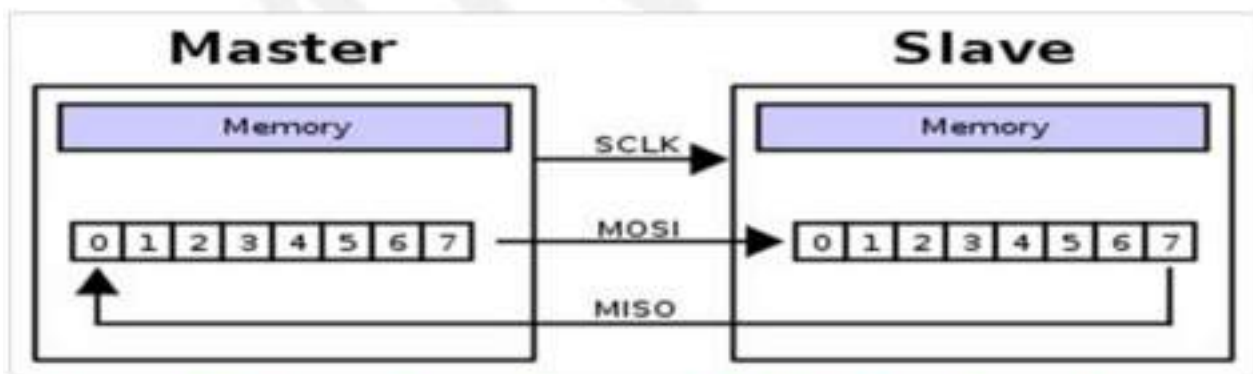
SPI interfacing



- The data out line (MISO) of all the slave devices when not selected floats at high impedance state
- The serial data transmission through SPI Bus is fully configurable.
- SPI devices contain certain set of registers for holding these configurations.
- The Serial Peripheral Control Register holds the various configuration parameters like master/slave selection for the device, baudrate selection for communication, clock signal control etc.
- The status register holds the status of various conditions for transmission and reception.
- SPI works on the principle of „Shift Register“.
- The master and slave devices contain a special shift register for the data to transmit or receive.
- The size of the shift register is device dependent.
- Normally it is a multiple of 8.



- During transmission from the master to slave, the data in the master's shift register is shifted out to the MOSI pin and it enters the shift register of the slave device through the MOSI pin of the slave device.
- At the same time the shifted out data bit from the slave device's shift register enters the shift register of the master device through MISO pin



Master shifts out data to Slave, and shift in data from Slave

2. Product level communication interface (External Communication Interface):

The Product level communication interface (External Communication Interface) is responsible for data transfer between the embedded system and other devices or modules

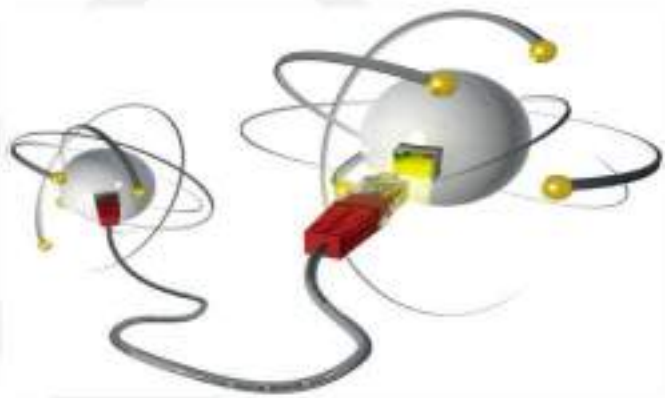
It is classified into two types

1. Wired communication interface
2. Wireless communication interface:

1. Wired communication interface: Wired communication interface is an interface used to transfer information over a wired network.

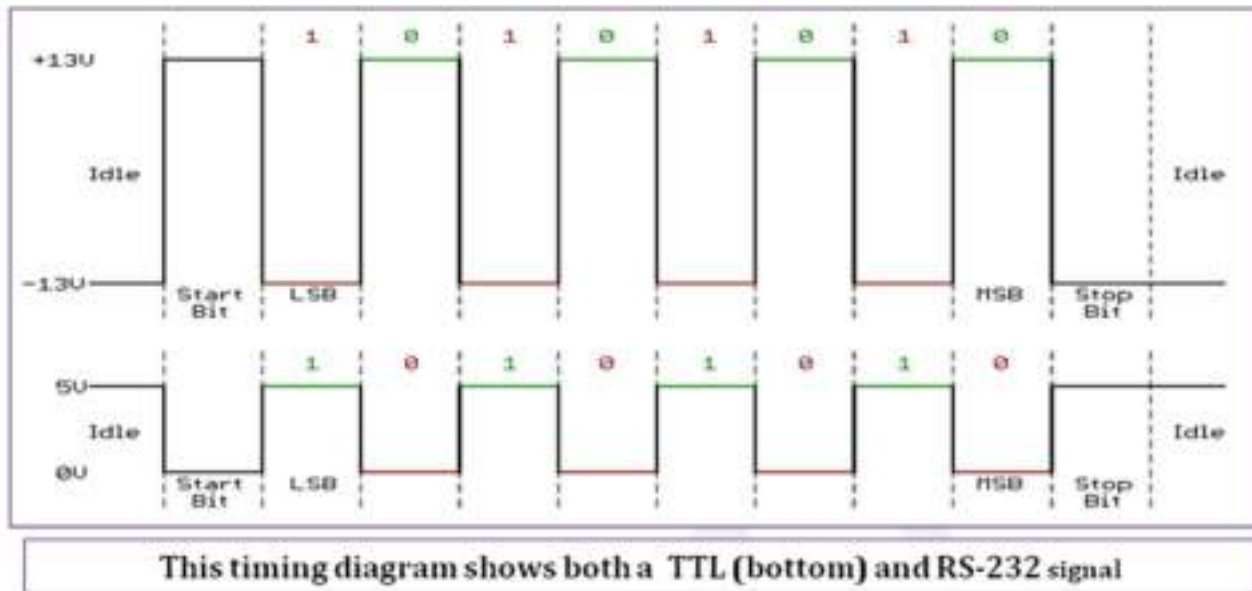
It is classified into following types.

1. RS-232C/RS-422/RS 485
2. USB
3. IEEE 1394 port

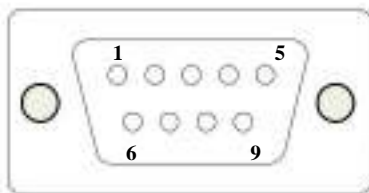


1. RS-232C:

- RS-232 C (Recommended Standard number 232, revision C from the Electronic Industry Association) is a legacy, full duplex, wired, asynchronous serial communication interface
- RS-232 extends the UART communication signals for external data communication.
- UART uses the standard TTL/CMOS logic (Logic „High“ corresponds to bit value 1 and Logic „LOW“ corresponds to bit value 0) for bit transmission whereas RS232 use the EIA standard for bit transmission.
- As per EIA standard, a logic „0“ is represented with voltage between +3 and +25V and a logic „1“ is represented with voltage between -3 and -25V.
- In EIA standard, logic „0“ is known as „Space“ and logic „1“ as „Mark“.
- The RS232 interface define various handshaking and control signals for communication apart from the „Transmit“ and „Receive“ signal lines for data communication



RS-232 supports two different types of connectors, namely; DB-9: 9-Pin connector and DB-25: 25-Pin connector.



DB-9

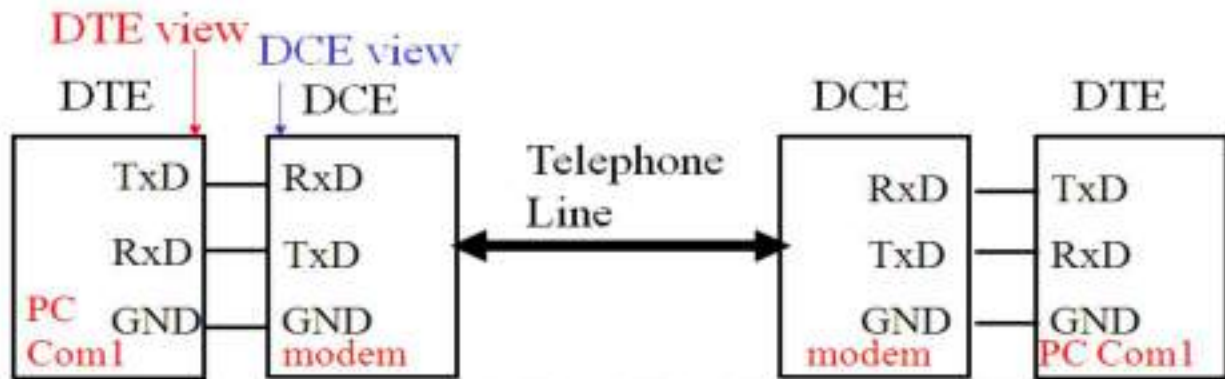


DB-25

Pin Name	Pin No:(For DB-9)	Description
TXD	3	Transmit Pin. Used for Transmitting Serial Data
RXD	2	Receive Pin. Used for Receiving serial Data
RTS	7	Request to send.
CTS	8	Clear To Send
DSR	6	Data Set ready
GND	5	Signal Ground
DCD	1	Data Carrier Detect
DTR	4	Data Terminal Ready
RI	9	Ring Indicator

- RS-232 is a point-to-point communication interface and the devices involved in RS-232 communication are called „Data Terminal Equipment (DTE)“ and „Data Communication Equipment (DCE)“

- If no data flow control is required, only TXD and RXD signal lines and ground line (GND) are required for data transmission and reception.
- The RXD pin of DCE should be connected to the TXD pin of DTE and vice versa for proper data transmission.



- If hardware data flow control is required for serial transmission, various control signal lines of the RS-232 connection are used appropriately.
- The control signals are implemented mainly for modem communication and some of them may be irrelevant for other type of devices
- The Request To Send (RTS) and Clear To Send (CTS) signals co-ordinate the communication between DTE and DCE.
- Whenever the DTE has a data to send, it activates the RTS line and if the DCE is ready to accept the data, it activates the CTS line
- The Data Terminal Ready (DTR) signal is activated by DTE when it is ready to accept data.
- The Data Set Ready (DSR) is activated by DCE when it is ready for establishing a communication link.
- DTR should be in the activated state before the activation of DSR
- The Data Carrier Detect (DCD) is used by the DCE to indicate the DTE that a good signal is being received
- Ring Indicator (RI) is a modem specific signal line for indicating an incoming call on the telephone line.
- As per the EIA standard RS-232 C supports baudrates up to 20Kbps (Upper limit 19.2Kbps)

- The commonly used baudrates by devices are 300bps, 1200bps, 2400bps, 9600bps, 11.52Kbps and 19.2Kbps
- The maximum operating distance supported in RS-232 communication is 50 feet at the highest supported baudrate.
- Embedded devices contain a UART for serial communication and they generate signal levels conforming to TTL/CMOS logic.
- A level translator IC like MAX 232 from Maxim Dallas semiconductor is used for converting the signal lines from the UART to RS-232 signal lines for communication.
- On the receiving side the received data is converted back to digital logic level by a converter IC.
- Converter chips contain converters for both transmitter and receiver
- RS-232 uses single ended data transfer and supports only point-to-point communication and not suitable for multi-drop communication

2. Wireless communication interface :

Wireless communication interface is an interface used to transmission of information over a distance without help of wires, cables or any other forms of electrical conductors.

They are basically classified into following types

1. IrDA

2. Bluetooth

3. Wi-Fi

4. Zigbee

5. GPRS

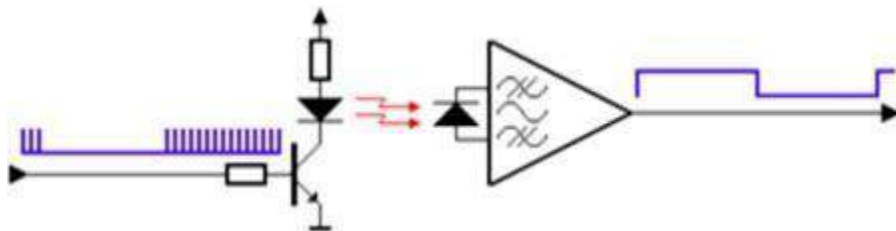
|

INFRARED:

- Infrared is a certain region in the light spectrum
- Ranges from $.7\mu$ to 1000μ or $.1\text{mm}$
- Broken into near, mid, and far infrared
- One step up on the light spectrum from visible light
- Measure of heat



Most of the thermal radiation emitted by objects near room temperature is infrared. Infrared radiation is used in industrial, scientific, and medical applications. Night-vision devices using active near-infrared illumination allow people or animals to be observed without the observer being detected.

**IR transmission:**

The transmitter of an IR LED inside its circuit, which emits infrared light for every electric pulse given to it. This pulse is generated as a button on the remote is pressed, thus completing the circuit, providing bias to the LED.

The LED on being biased emits light of the wavelength of 940nm as a series of pulses, corresponding to the button pressed. However since along with the IR LED many other sources of infrared light such as us human beings, light bulbs, sun, etc, the transmitted information can be interfered. A solution to this problem is by modulation.

The transmitted signal is modulated using a carrier frequency of 38 KHz (or any other frequency between 36 to 46 KHz). The IR LED is made to oscillate at this frequency for the time duration of the pulse. The information or the light signals are pulse width modulated and are contained in the 38 KHz frequency.

IR supports data rates ranging from 9600bits/second to 16Mbps

Serial infrared: 9600bps to 115.2 kbps

Medium infrared: 0.576Mbps to 1.152 Mbps

Fast infrared: 4Mbps

BLUETOOTH:

Bluetooth is a wireless technology standard for short distances (using short-wavelength UHF band from 2.4 to 2.485 GHz) for exchanging data over radio waves in the ISM and mobile devices, and building personal area networks (PANs). Invented by telecom vendor Ericsson in 1994, it was originally conceived as a wireless alternative to RS-232 data cables.

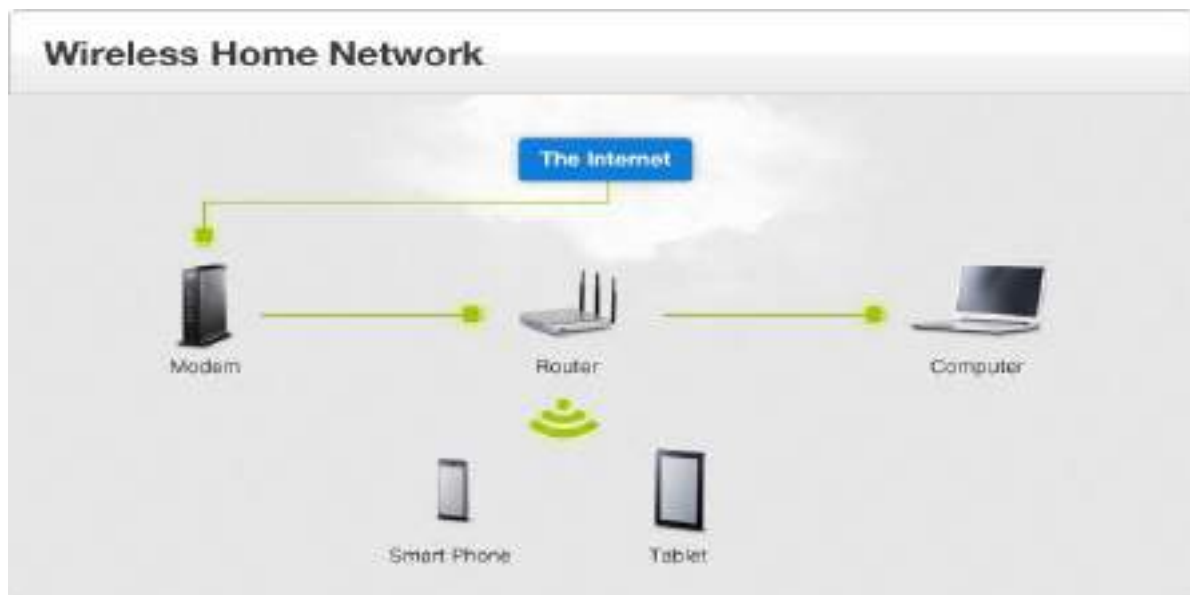
Bluetooth uses a radio technology called frequency-hopping spread spectrum. Bluetooth divides transmitted data into packets, and transmits each packet on one of 79 designated Bluetooth channels. Each channel has a bandwidth of 1 MHz. It usually performs 800 hops per second, with Adaptive Frequency-Hopping (AFH) enabled

Originally, Gaussian frequency-shift keying (GFSK) modulation was the only modulation scheme available. Since the introduction of Bluetooth 2.0+EDR, $\pi/4$ -DQPSK (Differential Quadrature Phase Shift Keying) and 8DPSK modulation may also be used between compatible devices. Bluetooth is a packet-based protocol with a master-slave structure. One master may communicate with up to seven slaves in a piconet. All devices share the master's clock. Packet exchange is based on the basic clock, defined by the master, which ticks at 312.5 μ s intervals.

A master BR/EDR Bluetooth device can communicate with a maximum of seven devices in a piconet (an ad-hoc computer network using Bluetooth technology), though not all devices reach this maximum. The devices can switch roles, by agreement, and the slave can become the master (for example, a headset initiating a connection to a phone necessarily begins as master—as initiator of the connection—but may subsequently operate as slave).

Wi-Fi:

- Wi-Fi is the name of a popular wireless networking technology that uses radio waves to provide wireless high-speed Internet and network connections
- Wi-Fi follows the IEEE 802.11 standard
- Wi-Fi is intended for network communication and it supports Internet Protocol (IP) based communication
- Wi-Fi based communications require an intermediate agent called Wi-Fi router/Wireless Access point to manage the communications.
- The Wi-Fi router is responsible for restricting the access to a network, assigning IP address to devices on the network, routing data packets to the intended devices on the network.



- Wi-Fi enabled devices contain a wireless adaptor for transmitting and receiving data in the form of radio signals through an antenna.

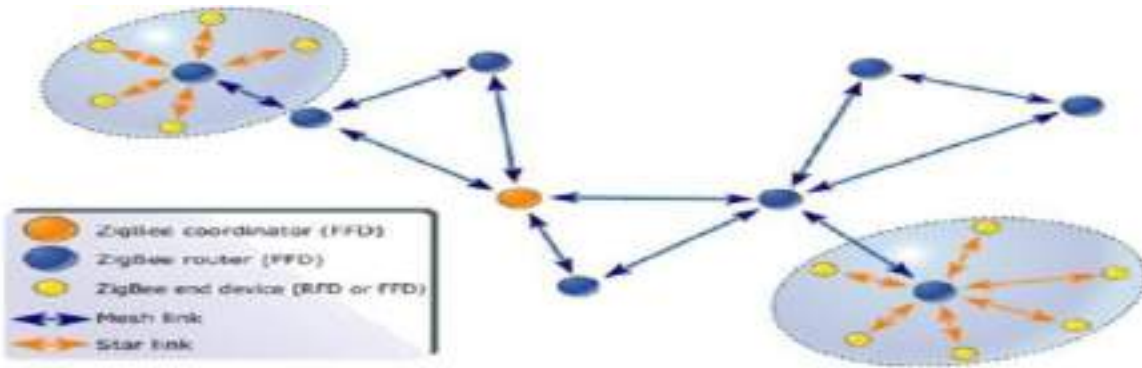
- Wi-Fi operates at 2.4GHZ or 5GHZ of radio spectrum and they co-exist with other ISM band devices like Bluetooth.
- A Wi-Fi network is identified with a Service Set Identifier (SSID). A Wi-Fi device can connect to a network by selecting the SSID of the network and by providing the credentials if the network is security enabled
- Wi-Fi networks implements different security mechanisms for authentication and data transfer.
- Wireless Equivalency Protocol (WEP), Wireless Protected Access (WPA) etc are some of the security mechanisms supported by Wi-Fi networks in data communication.

ZIGBEE:

Zigbee is an IEEE 802.15.4-based specification for a suite of high- level communication protocols used to create personal area networks with small, low-power digital radios, such as for home automation, medical device data collection, and other low-power low-bandwidth needs, designed for small scale projects which need wireless connection. Hence, zigbee is a low-power, low data rate, and close proximity (i.e., personal area) wireless ad hoc network.

The technology defined by the zigbee specification is intended to be simpler and less expensive than other wireless personal area networks (WPANs), such as Bluetooth or Wi-Fi . Applications include wireless light switches, electrical meters with in-home-displays, traffic management systems, and other consumer and industrial equipment that require short-range low-rate wireless data transfer.

Its low power consumption limits transmission distances to 10– 100 meters line-of-sight, depending on power output and environmental characteristics. Zigbee devices can transmit data over long distances by passing data through a mesh network of intermediate devices to reach more distant ones.



Zigbee Coordinator: The zigbee coordinator acts as the root of the zigbee network. The ZC is responsible for initiating the Zigbee network and it has the capability to store information about the network.

Zigbee Router: Responsible for passing information from device to another device or to another ZR.

Zigbee end device: End device containing zigbee functionality for data communication. It can talk only with a ZR or ZC and doesn't have the capability to act as a mediator for transferring data from one device to another.

Zigbee supports an operating distance of up to 100 metres at a data rate of 20 to 250 Kbps.

General Packet Radio Service(GPRS):

General Packet Radio Service (GPRS) is a packet oriented mobile data service on the 2G and 3G cellular communication system's global system for mobile communications (GSM). GPRS was originally standardized by European Telecommunications Standards Institute (ETSI) GPRS usage is typically charged based on volume of data transferred, contrasting with circuit switched data, which is usually billed per minute of connection time. Sometimes billing time is broken down to every third of a minute. Usage above the bundle cap is charged per megabyte, speed limited, or disallowed.

Services offered:

- GPRS extends the GSM Packet circuit switched data capabilities and makes the following services possible:
- SMS messaging and broadcasting
- "Always on" internet access
- Multimedia messaging service (MMS)
- Push-to-talk over cellular (PoC)
- Instant messaging and presence-wireless village Internet applications for smart devices through wireless application protocol (WAP).
- Point-to-point (P2P) service: inter-networking with the Internet (IP).
- Point-to-multipoint (P2M) service]: point-to- multipoint multicast and point-to-multipoint group calls.

Text Book:-**1. Introduction to Embedded Systems – Shibu K.V Mc Graw Hill**

Embedded Systems



UNIT-4

EMBEDDED FIRMWARE DESIGN & DEVELOPMENT

N.SURESH
Department of ECE



**MALLA REDDY COLLEGE OF
ENGINEERING & TECHNOLOGY**

Permanently Affiliated to JNTUH and Approved by AICTE, New Delhi

Embedded Firmware

Introduction:

- The control algorithm (Program instructions) and or the configuration settings that an embedded system developer dumps into the code (Program) memory of the embedded system
- It is an un-avoidable part of an embedded system.
- The embedded firmware can be developed in various methods like
 - Write the program in high level languages like Embedded C/C++ using an Integrated Development Environment (The IDE will contain an editor, compiler, linker, debugger, simulator etc. IDEs are different for different family of processors/controllers.
 - Write the program in Assembly Language using the Instructions Supported by your application's target processor/controller

Embedded Firmware Design & Development:

- The embedded firmware is responsible for controlling the various peripherals of the embedded hardware and generating response in accordance with the functional requirements of the product.
- The embedded firmware is the master brain of the embedded system.
- The embedded firmware imparts intelligence to an Embedded system.
- It is a onetime process and it can happen at any stage.
- The product starts functioning properly once the intelligence imparted to the product by embedding the firmware in the hardware.
- The product will continue serving the assigned task till hardware breakdown occurs or a corruption in embedded firmware.
- In case of hardware breakdown , the damaged component may need to be replaced and for firmware corruptions the firmware should be re-loaded, to bring back the embedded product to the normal functioning.

- The embedded firmware is usually stored in a permanent memory (ROM) and it is non alterable by end users.
- Designing Embedded firmware requires understanding of the particular embedded product hardware, like various component interfacing, memory map details, I/O port details, configuration and register details of various hardware chips used and some programming language (either low level Assembly Language or High level language like C/C++ or a combination of the two)
- The embedded firmware development process starts with the conversion of the firmware requirements into a program model using various modeling tools.
- The firmware design approaches for embedded product is purely dependent on the complexity of the functions to be performed and speed of operation required.
- There exist two basic approaches for the design and implementation of embedded firmware, namely;
 - **The Super loop based approach**
 - **The Embedded Operating System based approach**
- The decision on which approach needs to be adopted for firmware development is purely dependent on the complexity and system requirements

1. Embedded firmware Design Approaches – The Super loop:

- The Super loop based firmware development approach is Suitable for applications that are not time critical and where the response time is not so important (Embedded systems where missing deadlines are acceptable).

- It is very similar to a conventional procedural programming where the code is executed task by task
- The tasks are executed in a never ending loop.
- The task listed on top on the program code is executed first and the tasks just below the top are executed after completing the first task
- A typical super loop implementation will look like:
 1. Configure the common parameters and perform initialization for various hardware components memory, registers etc.
 2. Start the first task and execute it
 3. Execute the second task
 4. Execute the next task
 5. :
 6. :
 7. Execute the last defined task
 8. Jump back to the first task and follow the same flow.

The 'C' program code for the super loop is given below

```
void main ()
{
    Configurations ();
    Initializations ();

    while (1)
    {
        Task 1 ();
        Task 2 ();
        :
    }
}
```

```

:
Task n ();
}
}

```

Pros:

- Doesn't require an Operating System for task scheduling and monitoring and free from OS related overheads
- Simple and straight forward design
- Reduced memory footprint

Cons:

- Non Real time in execution behavior (As the number of tasks increases the frequency at which a task gets CPU time for execution also increases)
- Any issues in any task execution may affect the functioning of the product (This can be effectively tackled by using Watch Dog Timers for task execution monitoring)

Enhancements:

- Combine Super loop based technique with interrupts
- Execute the tasks (like keyboard handling) which require Real time attention as Interrupt Service routines.

2. Embedded firmware Design Approaches – Embedded OS based Approach:

- The embedded device contains an Embedded Operating System which can be one of:
 - A Real Time Operating System (RTOS)
 - A Customized General Purpose Operating System (GPOS)

- The Embedded OS is responsible for scheduling the execution of user tasks and the allocation of system resources among multiple tasks
- It Involves lot of OS related overheads apart from managing and executing user defined tasks
- Microsoft® Windows XP Embedded is an example of GPOS for embedded devices
- Point of Sale (PoS) terminals, Gaming Stations, Tablet PCs etc are examples of embedded devices running on embedded GPOSs
- ‘Windows CE’, ‘Windows Mobile’, ‘QNX’, ‘VxWorks’, ‘ThreadX’, ‘MicroC/OS-II’, ‘Embedded Linux’, ‘Symbian’ etc are examples of RTOSs employed in Embedded Product development
- Mobile Phones, PDAs, Flight Control Systems etc are examples of embedded devices that runs on RTOSs

Embedded firmware Development Languages/Options

- **Assembly Language**
- **High Level Language**
 - Subset of C (Embedded C)
 - Subset of C++ (Embedded C++)
 - Any other high level language with supported Cross-compiler
- **Mix of Assembly & High level Language**
 - Mixing High Level Language (Like C) with Assembly Code
 - Mixing Assembly code with High Level Language (Like C)
 - Inline Assembly

1. Embedded firmware Development Languages/Options – Assembly Language

- ‘*Assembly Language*’ is the human readable notation of ‘*machine language*’
- ‘*Machine language*’ is a processor understandable language
- Machine language is a binary representation and it consists of 1s and 0s
- Assembly language and machine languages are processor/controller dependent
- An Assembly language program written for one processor/controller family will not work with others
- Assembly language programming is the process of writing processor specific machine code in mnemonic form, converting the mnemonics into actual processor instructions (machine language) and associated data using an assembler
- The general format of an assembly language instruction is an Opcode followed by Operands
- The Opcode tells the processor/controller what to do and the Operands provide the data and information required to perform the action specified by the opcode
- It is not necessary that all opcode should have Operands following them. Some of the Opcode implicitly contains the operand and in such situation no operand is required. The operand may be a single operand, dual operand or more

The 8051 Assembly Instruction

MOV A, #30

Moves decimal value 30 to the 8051 Accumulator register. Here *MOV A* is the Opcode and 30 is the operand (single operand). The same instruction when written in machine language will look like

01110100 00011110

The first 8 bit binary value 01110100 represents the opcode *MOV A* and the second 8 bit binary value 00011110 represents the operand 30.

- Assembly language instructions are written one per line
- A machine code program consists of a sequence of assembly language instructions, where each statement contains a mnemonic (Opcode + Operand)
- Each line of an assembly language program is split into four fields as:

LABEL OPCODE OPERAND COMMENTS

- LABEL is an optional field. A 'LABEL' is an identifier used extensively in programs to reduce the reliance on programmers for remembering where data or code is located. LABEL is commonly used for representing
 - ❖ A memory location, address of a program, sub-routine, code portion etc.
 - ❖ The maximum length of a label differs between assemblers. Assemblers insist strict formats for labeling. Labels are always suffixed by a colon and begin with a valid character. Labels can contain number from 0 to 9 and special character _ (underscore).

```
#####
; SUBROUTINE FOR GENERATING DELAY
; DELAY PARAMETR PASSED THROUGH REGISTER R1
; RETURN VALUE NONE,REGISTERS USED: R0, R1
#####
#####  DELAY:  MOV R0, #255    ; Load Register R0 with 255

                DJNZ R1, DELAY; Decrement R1 and loop till    R1= 0

                RET              ; Return to calling program
```

- The symbol ; represents the start of a comment. Assembler ignores the text in a line after the ; symbol while assembling the program
- DELAY is a label for representing the start address of the memory location where the piece of code is located in code memory
- The above piece of code can be executed by giving the label DELAY as part of the instruction. E.g. LCALL DELAY; LMP DELAY

2.Assembly Language – Source File to Hex File Translation:

- The Assembly language program written in assembly code is saved as .asm (Assembly file) file or a .src (source) file or a format supported by the assembler
- Similar to 'C' and other high level language programming, it is possible to have multiple source files called modules in assembly language programming. Each module is represented by a '.asm' or '.src' file or the assembler supported file format similar to the '.c' files in C programming
- The software utility called 'Assembler' performs the translation of assembly code to machine code
- The assemblers for different family of target machines are different. A51 Macro Assembler from Keil software is a popular assembler for the 8051 family micro controller

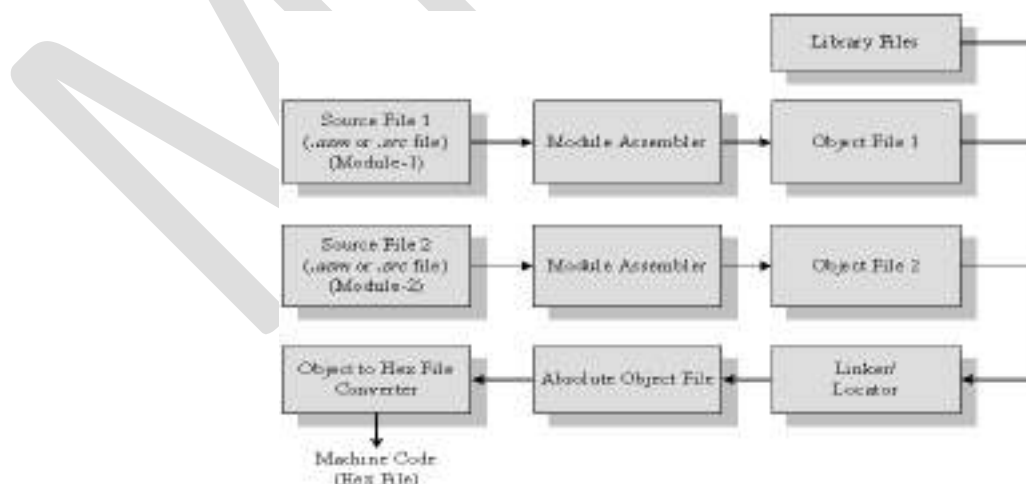


Figure 5: Assembly Language to machine language conversion process

- Each source file can be assembled separately to examine the syntax errors and incorrect assembly instructions
- Assembling of each source file generates a corresponding object file. The object file does not contain the absolute address of where the generated code needs to be placed (a re-locatable code) on the program memory
- The software program called linker/locator is responsible for assigning absolute address to object files during the linking process
- The Absolute object file created from the object files corresponding to different source code modules contain information about the address where each instruction needs to be placed in code memory
- A software utility called 'Object to Hex file converter' translates the absolute object file to corresponding hex file (binary file)

Advantages:

★ 1.Efficient Code Memory & Data Memory Usage (Memory Optimization):

- The developer is well aware of the target processor architecture and memory organization, so optimized code can be written for performing operations.
- This leads to less utilization of code memory and efficient utilization of data memory.

★ 2.High Performance:

- Optimized code not only improves the code memory usage but also improves the total system performance.
- Through effective assembly coding, optimum performance can be achieved for target processor.

★ 3.Low level Hardware Access:

- Most of the code for low level programming like accessing external device specific registers from OS kernel ,device drivers, and low level interrupt routines, etc are making use of direct assembly coding.

★ 4.Code Reverse Engineering:

- It is the process of understanding the technology behind a product by extracting the information from the finished product.
- It can easily be converted into assembly code using a dis-assembler program for the target machine.

Drawbacks:**★ 1.High Development time:**

- The developer takes lot of time to study about architecture ,memory organization, addressing modes and instruction set of target processor/controller.
- More lines of assembly code is required for performing a simple action.

★ 2.Developer dependency:

- There is no common written rule for developing assembly language based applications.

★ 3.Non portable:

- Target applications written in assembly instructions are valid only for that particular family of processors and cannot be re-used for another target processors/controllers.
- If the target processor/controller changes, a complete re-writing of the application using assembly language for new target processor/controller is required.

2. Embedded firmware Development Languages/Options – High Level Language

- The embedded firmware is written in any high level language like C, C++
- A software utility called ‘cross-compiler’ converts the high level language to target processor specific machine code

- The cross-compilation of each module generates a corresponding object file. The object file does not contain the absolute address of where the generated code needs to be placed (a re-locatable code) on the program memory
- The software program called linker/locator is responsible for assigning absolute address to object files during the linking process
- The Absolute object file created from the object files corresponding to different source code modules contain information about the address where each instruction needs to be placed in code memory
- A software utility called 'Object to Hex file converter' translates the absolute object file to corresponding hex file (binary file)

Embedded firmware Development Languages/Options – High Level Language – Source File to Hex File Translation

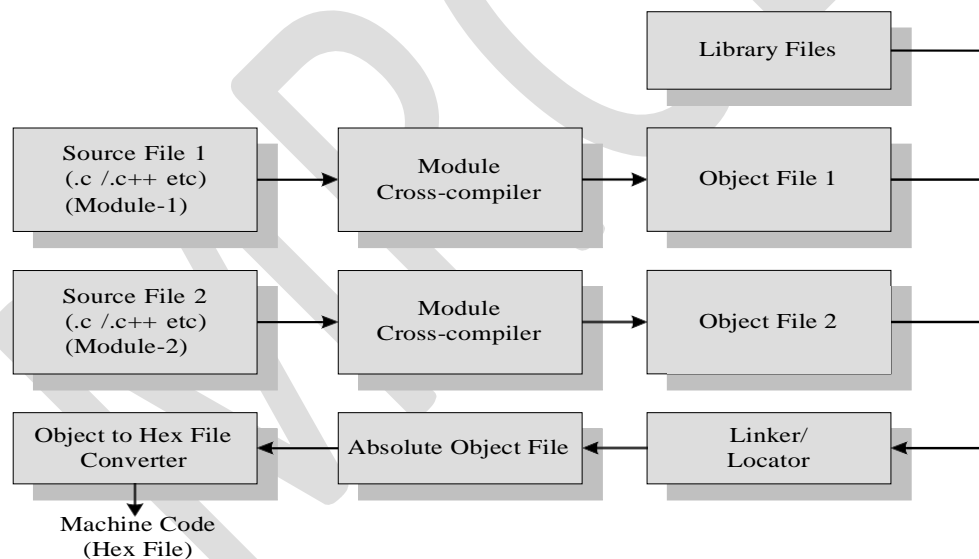


Figure 6: High level language to machine language conversion process

Advantages:

- **Reduced Development time:** Developer requires less or little knowledge on internal hardware details and architecture of the target processor/Controller.
- **Developer independency:** The syntax used by most of the high level languages are universal and a program written high level can easily understand by a second person knowing the syntax of the language
- **Portability:** An Application written in high level language for particular target processor /controller can be easily be converted to another target processor/controller specific application with little or less effort

Drawbacks:

- The cross compilers may not be efficient in generating the optimized target processor specific instructions.
- Target images created by such compilers may be messy and non-optimized in terms of performance as well as code size.
- The investment required for high level language based development tools (IDE) is high compared to Assembly Language based firmware development tools.

Embedded firmware Development Languages/Options – Mixing of Assembly Language with High Level Language

- Embedded firmware development may require the mixing of Assembly Language with high level language or vice versa.
- Interrupt handling, Source code is already available in high level language\Assembly Language etc are examples

- High Level language and low level language can be mixed in three different ways
 - ✓ Mixing Assembly Language with High level language like 'C'
 - ✓ Mixing High level language like 'C' with Assembly Language
 - ✓ In line Assembly
- The passing of parameters and return values between the high level and low level language is cross-compiler specific

1. Mixing Assembly Language with High level language like 'C' (Assembly Language with 'C'):

- Assembly routines are mixed with 'C' in situations where the entire program is written in 'C' and the cross compiler in use do not have built in support for implementing certain features like ISR.
- If the programmer wants to take advantage of the speed and optimized code offered by the machine code generated by hand written assembly rather than cross compiler generated machine code.
- For accessing certain low level hardware ,the timing specifications may be very critical and cross compiler generated machine code may not be able to offer the required time specifications accurately.
- Writing the hardware/peripheral access routine in processor/controller specific assembly language and invoking it from 'C' is the most advised method.
- Mixing 'C' and assembly is little complicated.
- The programmer must be aware of how to pass parameters from the 'C' routine to assembly and values returned from assembly routine to 'C' and how Assembly routine is invoked from the 'C' code.

- Passing parameter to the assembly routine and returning values from the assembly routine to the caller 'C' function and the method of invoking the assembly routine from 'C' code is cross compiler dependent.
- There is no universal written rule for purpose.
- We can get this information from documentation of the cross compiler.
- Different cross compilers implement these features in different ways depending on GPRs and memory supported by target processor/controller

2. Mixing High level language like 'C' with Assembly Language ('C' with Assembly Language)

- The source code is already available in assembly language and routine written in a high level language needs to be included to the existing code.
- The entire source code is planned in Assembly code for various reasons like optimized code, optimal performance, efficient code memory utilization and proven expertise in handling the assembly.
- The functions written in 'C' use parameter passing to the function and returns values to the calling functions.
- The programmer must be aware of how parameters are passed to the function and how values returned from the function and how function is invoked from the assembly language environment.
- Passing parameter to the function and returning values from the function using CPU registers , stack memory and fixed memory.
- Its implementation is cross compiler dependent and varies across compilers.

3.In line Assembly:

- Inline assembly is another technique for inserting the target processor/controller specific assembly instructions at any location of source code written in high level language 'C'
- Inline Assembly avoids the delay in calling an assembly routine from a 'C' code.
- Special keywords are used to indicate the start and end of Assembly instructions
- *E.g #pragma asm*

Mov A,#13H

#pragma endasm

- Keil C51 uses the keywords *#pragma asm* and *#pragma endasm* to indicate a block of code written in assembly.

Text Books:

- 1. Introduction to Embedded Systems – Shibu K.V Mc Graw Hill**
- 2. Embedded System Design-Raj Kamal TMH**

Unit -5

Embedded Programming Concept using C, C++ and Java

5.1 SOFTWARE PROGRAMMING IN ASSEMBLY LANGUAGE (ALP) AND IN HIGH LEVEL LANGUAGE 'C'

Assembly language coding of an application has the following advantages:

1. *It gives a precise control* of the processor internal devices and *full use of processor specific features* in its instruction set and its addressing modes.
2. The machine codes are compact. This is because the codes for declaring the conditions, rules, and data type do not exist. The system thus needs a smaller memory. Excess memory needed does not depend on the programmer data type selection and rule-declarations. It is also not the compiler specific and library functions specific.
3. Device driver codes may need only a few assembly instructions. For example, consider a small- embedded system, a timer device in a microwave oven or an automatic washing machine or an automatic chocolate vending machine. Assembly codes for these can be compact and precise, and are conveniently written.

It becomes convenient to develop the *source files* in C or C++ or Java for complex systems because of the following advantages of high-level languages for such systems.

1. ***The development cycle is short for complex systems*** due to the use of functions (procedures), standard library functions, modular programming approach and top down design. Application programs are structured to ensure that the software is based on sound software engineering principles.
 - (a) Let us recall Example 4.8 of a UART serial line device driver. Direct use of this function makes the repetitive coding redundant as this device is used in many systems. We simply change some of the arguments (for the variables) passed when needed and use it at another instance of the device use.
 - (b) Should the square root codes be written again whenever the square root of another value (argument) is to be taken? The use of the standard library function, square root (), saves the programmer time for coding. New sets of library functions exist in an embedded system specific C or C++ compiler. Exemplary functions are the delay (), wait () and sleep ().
 - (c) Modular programming approach is an approach in which the building blocks are reusable software components. Consider an analogy to an IC (Integrated Circuit). Just as an IC has several circuits integrated into one, similarly a building block may call several functions and library functions. A module should however, be well tested. It must have a well-defined goal and the well-defined data inputs and outputs. It should have only one calling procedure. There should be one return point from it. It should not affect any data other than that which is targeted. [Data Encapsulation.] It must return (report) error conditions encountered during its execution.
 - (d) Bottom up design is a design approach in which programming is first done for the sub-modules of the specific and distinct sets of actions. An example of the modules for specific sets of actions is a program for a software timer, RTCSWT:: run. Programs for delay, counting, finding time intervals and many applications can be written. Then the final program is designed. The approach to this way of designing a program is to first code the basic functional modules and then use these to build a bigger module.
 - (e) Top-Down design is another programming approach in which the *main* program is first designed, then its modules, sub-modules, and finally, the functions.
2. ***Data type declarations provide programming ease.*** For example, there are four types of integers, *int*, *unsigned int*, *short* and *long*. When dealing with positive only values, we declare a variable as *unsigned int*. For example, numTicks (Number of Ticks of a clock before the timeout) has to be unsigned. We need a signed integer, *int* (32 bit) in arithmetical calculations. An integer can also be declared as data type, *short* (16 bit) or *long* (64 bit). To manipulate the text and strings for a character, another data type is *char*. *Each data type is an abstraction for the methods to use, to manipulate, to represent, and for a set of permissible operations.*

3. Type checking makes the program less prone to error. For example, type checking does not permit subtraction, multiplication and division on the *char* data types. Further, it lets + be used for concatenation. [For example, micro + controller concatenates into microcontroller, where micro is an array of *char* values and controller is another array of *char* values.]
4. Control Structures (for examples, *while*, *do - while*, *break* and *for*) and Conditional Statements (for examples, *if*, *if- else*, *else - if* and *switch - case*) make the program-flow path design tasks simple.
5. Portability of non-processor specific codes exists. Therefore, when the hardware changes, only the modules for the device drivers and device management, initialization and locator modules and initial boot up record data need modifications.

Additional advantages of C as a high level languages are as follows:

1. It is a language between low (assembly) and high level language. Inserting the assembly language codes in between is called in-line assembly. A direct hardware control is thus also feasible by in-line assembly, and the complex part of the program can be in high-level language. Example 4.5 showed the use of in-line assembly codes in C for a Port A Driver Program.



High level language programming makes the program development cycle short, enables use of the modular programming approach and lets us follow sound software engineering principles. It facilitates the program development with 'Bottom up design' and 'top down design' approaches. Embedded system programmers have long preferred C for the following reasons: (i) The feature of embedding assembly codes using in-line assembly. (ii) Readily available modules in C compilers for the embedded system and library codes that can directly port into the system-programmer codes.

5.2 'C' PROGRAM ELEMENTS: HEADER AND SOURCE FILES AND PREPROCESSOR DIRECTIVES

The 'C' program elements, header and source files and preprocessor directives are as follows:

Include Directive for the Inclusion of Files

Any C program first includes the header and source files that are readily available. *One does not keep a cow at home when milk is readily available!* A case study of sending a stream of bytes through a network driver card using a TCP/IP protocol is given in Example 11.2. Its program starts with the codes given in Example 4.5 and Example 5.1. The purpose of each included file is mentioned in the comments within the * symbols as per 'C' practice.

Example 5.1

```
# include "vxWorks.h" /* Include VxWorks functions*/
# include "semLib.h" /* Include Semaphore functions Library */
# include "taskLib.h" /* Include multitasking functions
Library */
# include "msgQLib.h" /* Include Message Queue functions Library */
# include "fioLib.h" /* Include File-Device Input-Output functions Library
*/ # include "sysLib.c" /* Include system library for system
functions */
# include "netDrvConfig.txt" /* Include a text file that provides the 'Network Driver Configuration'. It
provides the frame format protocol (SLIP or PPP or Ethernet) description, card description/make, ad-
dress at the system, IP address (s) of the node (s) that drive the card for transmitting or receiving from
the network. */
# include "prctlHandlers.c" /* Include file for the codes for handling and actions as per the protocols
used for driving streams to the network. */
```

Include is a preprocessor directive to include the contents (codes or data) of a file. The files that can be included are given below. Inclusion of all files and specific header files has to be as per require- ments.

- (i) *Including Codes Files*: These are the files for the codes already available. For example, #include *“prctlHandlers.c”*.
- (ii) *Including Constant data Files*: These are the files for the codes and may have the extension *‘.const’*.
- (iii) *Including Strings data Files*: These are the files for the strings and may have the extension *‘.strings’* or *‘.str.’* or *‘.txt’*. For example, #include *“netDrvConfig.txt”* in Example 5.1.
- (iv) *Including initial data Files*: Recall Section 2.4.3 and Examples 2.13 to 2.15. There are files for the initial or default data for the shadow ROM of the embedded system. The boot-up program is copied later into the RAM and may have the extension *‘.init’*. On the other hand, RAM data files have the extension, *‘.data’*.
- (v) *Including basic variables Files*: These are the files for the local or global static variables that are stored in the RAM because they do not possess the initial (default) values. The static means that there is a common not more than one instance of that variable address and it has a static memory allocation. There is only one real time clock, and therefore only one instance of that variable address. [Refer case (iv) Section 5.4.3.] These basic variables are stored in the files with the extension *‘.bss’*.
- (vi) *Including Header Files*: It is a preprocessor directive, which includes the contents (codes or data) of a set of source files. These are the files of a specific module. A header file has the extension *‘.h’*. Examples are as follows. The string manipulation functions are needed in a program using strings. These become available once a header file called *“string.h”* is included. The mathematical functions, *square root, sin, cos, tan, atan* and so on are needed in programs using mathematical expressions. These become available by including a header file, called *“math.h”*. The pre-processor directives will be *‘#include <string.h>’* and *‘#include <math.h>’*. Also included are the header files for the codes in assembly, and for the I/O operations (*conio.h*), for the OS functions and RTOS functions. #include *“vxWorks.h”* in Example 5.1 is directive to compiler, which includes VxWorks RTOS functions.

Note: Certain compilers provide for *conio.h* in place of *stdio.h*. This is because embedded systems usually do not need the file functions for opening, closing, read and write. So when including *stdio.h*, it makes the code too big.

What is the difference between inclusion of a header file, and a text file or data file or constants file? Consider the inclusion of *netDrvConfig.txt* and *math.h*. (i) The header files are well tested and debugged modules. (ii) The header files provide access to standard libraries. (iii) The header file can include several text file or C files. (iv) A text file is description of the texts that contain specific information.

Source Files

Source files are program files for the functions of application software. The source files need to be compiled. A source file will also possess the preprocessor directives of the application and have the *first function from where the processing will start*. This function is called *main* function. Its codes start with *void main ()*. The *main* calls other functions. A source file holds the codes as like the ones given earlier in Examples 4.3 to 4.5.

Configuration Files

Configuration files are the files for the configuration of the system. Device configuration codes can be put in a file of basic variables and included when needed. If these codes are in the file *“serialLine_cfg.h”* then #include *“serialLine_cfg.h”* will be preprocessor directive. Consider another example. #include *“os_cfg.h”*. It will include *os_cfg* header file.

Preprocessor Directives

A preprocessor directive starts with a sharp (hash) sign. These commands are for the following directives to the compiler for processing.

1. *Preprocessor Global Variables*: “#define volatile boolean IntrEnable” is a preprocessor directive in Example 4.6. It means it is a directive before processing to consider *IntrEnable* a global variable of boolean data type and is volatile. *IntrDisable, IntrPortAEnable, IntrPortADisable, STAF* and *STAI* are the other global variables in Example 4.5.

2. *Preprocessor Constants*: “#define false 0” is a preprocessor directive in example 4.3. It means it is a directive before processing to assume ‘false’ as 0. The directive ‘define’ is for allocating pointer value(s) in the program. Consider #define portA (volatile unsigned char *) 0x1000 and #define PIOC (volatile unsigned char *) 0x1001. [Refer to Section 4.2.] 0x1000 and 0x1001 are for the fixed ad-

addresses of portA and PIOC. These are the constants defined here for these 68HC11 registers. Strings can also be defined. Strings are the constants, for example, those used for an initial display on the screen in a mobile system. For example, # define *welcome* “Welcome To ABC Telecom”.

Preprocessor constants, variables, and inclusion of configuration files, text files, header files and library functions are used in C programs.

5.3 PROGRAM ELEMENTS: MACROS AND FUNCTIONS

Table 5.1 lists these elements and gives their uses.

Table 5.1
Uses of the Various Sets of Instructions as the Program Elements

Program Element	Uses	Saves context on the stack before its start and retrieves them on return	Feasibility of nesting one within another
Macro function	Executes a named small collection of codes. No Executes a named set of codes with values passed by Yes the calling program through its arguments. Also returns		None Yes, can call another function and can also be interrupted.
Main-function	a data object when it is not declared as void. It has the context saving and retrieving overheads. Declarations of functions and data types, typedef and No either (i) Executes a named set of codes, calls a set of functions, and calls on the Interrupts the ISRs or (ii) starts an OS Kernel.		None

Program Element	Uses	Saves context on the stack before its start and retrieves them on return	Feasibility of nesting one within another
Reentrant function	Refer Section 5.4.6 (ii)	Yes	Yes to another reentrant function only
Interrupt Service Routine or Device Driver	Declarations of functions and data types, typedef, and Executes a named set of codes. Must be short so that other sources of interrupts are also serviced within the deadlines. Must be either a reentrant routine or must have a solution to the shared data problem.	Yes	To higher priority sources
Task	Refer Section 8.2. Must either be a reentrant routine or must have a solution to the shared data problem.	Yes	None
Recursive function	A function that calls itself. It must be a reentrant function also. Most often its use is avoided in embedded systems due to memory constraints. [Stack grows after each recursive call and its may choke the memory space availability.]	Yes	Yes

Preprocessor Macros: A macro is a collection of codes that is defined in a program by a name. It differs from a function in the sense that once a macro is defined by a name, the compiler puts the

corresponding codes for it at every place where that macro name appears. The ‘enable_Maskable_Intr ()’ and ‘disable_Maskable_Intr ()’ are the macros in Example 4.5. [The pair of brackets is optional. If it is present, it improves readability as it distinguishes a macro from a constant]. Whenever the name enable_Maskable_Intr appears, the compiler places the codes designed for it. Macros, called test macros or test vectors are also designed and used for debugging a system. [Refer Section 7.6.3.]

How does a macro differ from a function? The codes for a function are compiled once only. On calling that function, the processor has to save the context, and on return restore the context. Further, a function may return nothing (*void* declaration case) or return a Boolean value, or an integer or any primitive or reference type of data. [Primitive means similar to an integer or character. Reference type means similar to an array or structure.] The enable_PortA_Intr () and disable_PortA_Intr () are the function calls in Example 4.5. [The brackets are now not optional]. Macros are used for short codes only. This is because, if a function call is used instead of macro, the overheads (context saving and other actions on function call and return) will take a time, $T_{\text{overheads}}$ that is the same order of magnitude as the time, T_{exec} for execution of short codes within a function. We use a function when the $T_{\text{overheads}} \ll T_{\text{exec}}$, and a macro when $T_{\text{overheads}} \sim$ or $> T_{\text{exec}}$.



Macros and functions are used in C programs. Functions are used when the requirement is that the codes should be compiled once only. However, on calling a function, the processor has to save the context, and on return, restore the context. Further, a function may return nothing (*void* declaration case) or return a Boolean value, or an integer or any primitive or reference type of data. Macros are used when short functional codes are to be inserted in a number of places or functions.

5.4 PROGRAM ELEMENTS: DATA TYPES, DATA STRUCTURES, MODIFIERS, STATEMENTS, LOOPS AND POINTERS

Use of Data Types

Whenever a data is named, it will have the address(es) allocated at the memory. The number of addresses allocated depends upon the data type. ‘C’ allows the following primitive data types. The *char* (8 bit) for characters, *byte* (8 bit), *unsigned short* (16 bit), *short* (16 bit), *unsigned int* (32 bit), *int* (32 bit), *long double* (64 bit), *float* (32 bit) and *double* (64 bit). [Certain compilers do not take the ‘byte’ as a data type definition. The ‘char’ is then used instead of ‘byte’. Most C compilers do not take a Boolean variable as data type. As in second line of Example 4.6, *typedef* is used to create a Boolean type variable in the C program.]

A data type appropriate for the hardware is used. For example, a 16-bit timer can have only the unsigned short data type, and its range can be from 0 to 65535 only. The typedef is also used. It is made clear by the following example. A compiler version may not process the declaration as an unsigned byte. The ‘unsigned character’ can then be used as a data type. It can then be declared as follows:

```
typedef unsigned character portAdata #define Pbyte portAdata Pbyte = 0xF1
```

Use of Data Structures: Queues, Stacks, Lists and Trees

Marks (or grades) of a student in the different subjects studied in a semester are put in a proper table. The table in the mark-sheet shows them in an organised way. When there is a large amount of data, it must be organised properly. A data structure is a way of organising large amounts of data. A data element can then be identified and accessed with the help of a few pointers and/or indices and/or functions. [The reader may refer to a standard textbook for the data structure algorithms in C and C++. For example, “Data Structures and Algorithms in C++” by Adam Drozdek from Brooks/Cole Thomson Learning (2001).]

A data structure is an important element of any program. Section 2.5. defines and describes a few important data structures, *stack*, *one-dimensional array*, *queue*, *circular queue*, *pipe*, a *table* (two dimensional array), *lookup table*, *hash table* and *list*. Figures 2.3 to 2.5 showed different data structures and how it is put in the memory blocks in an organised way. Any data structure element can be retrieved.

Table 5.2

Uses of the Various Data Structures in a Program Element

Data Structure	Definition and when used	Example (s) of its use
Queue	It is a structure with a series of elements with the first element waiting for an operation. An operation can be done only in the first in first out (FIFO) mode. It is used when an element is not to be accessible by any index and pointer directly, but only through the FIFO. An element can be inserted only at the end in the series of elements waiting for an operation. There are two pointers, one for deleting after the operation and other for inserting. Both increment after an operation.	(1) Print buffer. Each character is to be printed in FIFO mode. (2) Frames on a network [Each frame also has a queue of a stream of bytes.] Each byte has to be sent for receiving as a FIFO. (3) Image frames in a sequence. [These have to be processed as a FIFO.]
Stack	It is a structure with a series of elements with its last element waiting for an operation. An operation can be done only in the last in first out (LIFO) mode. It is used when an element is not to be accessible by any index or pointer directly, but only through the LIFO. An element can be pushed (inserted) only at the top in the series of elements still waiting for an operation. There is only one pointer used for pop (deleting) after the operation as well as for push (inserting). Pointers increment or decrement after an operation. It depends on insertion or deletion.	(1) Pushing of variables on interrupt or call to another function. (2) Retrieving the pushed data onto a stack.
Array (one dimensional vector)	It is a structure with a series of elements with each element accessible by an identifier name and an index. Its element can be used and operated easily. It is used when each element of the structure is to be given a distinct identity by an index for easy operation. Index starts from 0 and is +ve integers.	$ts = 12 * s(1)$; Total salary, ts is 12 times the first month salary. $marks_weight[4] =$ marks in the subject with index 4 is assigned the same as in the subject with index 0.
Multi-dimensional array	It is a structure with a series of elements each having another sub-series of elements. Each element is accessible by identifier name and two or more indices. It is used when every element of the structure is to be given a distinct identity by two or more indices for easy operation. The dimension of an array equals the number of indices that are needed to distinctly identify an array-element. Indices start from 0 and are +ve integers.	Handling a matrix or tensor. Consider a pixel in an image frame. Consider Quarter-CIF image pixel in 144 x 176 size image frame. [Recall Section 1.2.7.] $pixel[108, 88]$ will represent a pixel at 108-th horizontal row and 88-th vertical column. #See following note also.
List	Each element has a pointer to its next element. Only the first element is identifiable and it is done by list-top pointer (Header). No other element is identifiable and hence is not accessible directly. By going through the first element, and then consecutively through all the succeeding elements, an element can be read, or read and deleted, or can be added to a neighbouring element or replaced by another element.	A series of tasks which are active Each task has pointer for the next task. Another example is a menu that point to a submenu.

Data Structure	Definition and when used	Example (s) of its use
Tree	There is a root element. It has two or more branches each having a daughter element. Each daughter element has two or more daughter elements. The last one does not have daughters. Only the root element is identifiable and it is done by the treetop pointer (Header). No other element is identifiable and hence is not accessible directly. By traversing the root element, then proceeding continuously through all the succeeding daughters, a tree element can be read or read and deleted, or can be added to another daughter or replaced by another element. A tree has data elements arranged as branches. The last daughter, called node has no further daughters. A binary tree is a tree with a maximum of two daughters (branches) in each element.	An example is a directory. It has number of file-folders. Each file-folder has a number of other file folders and so on In the end is a file.

Use of Modifiers

The actions of modifiers are as follows:

- Case (i): Modifier **'auto'** or No modifier, if *outside* a function block, means that there is ROM allocation for the variable by the locator if it is initialised in the program. RAM is allocated by the locator, if it is not initialised in the program.
- Case (ii): Modifier **'auto'** or No modifier, if *inside* the function block, means there is ROM allocation for the variable by the locator if it is initialised in the program. There is no RAM allocation by the locator.
- Case (iii): Modifier **'unsigned'** is modifier for a short or int or long data type. It is a directive to permit only the positive values of 16, 32 or 64 bits, respectively.
- Case (iv): Modifier **'static'** declaration is inside a function block. Static declaration is a directive to the compiler that the variable should be accessible outside the function block also, and there is to be a reserved memory space for it. It then does not save on a stack on context switching to another task. When several tasks are executed in cooperation, the declaration *static* helps. Consider an exemplary declaration, 'private: static void interrupt ISR_RTI ();'. The static declaration here is for the directive to the compiler that the ISR_RTI () function codes limit to the memory block for ISR_RTI () function. The private declaration here means that there are no other instances of that method in any other object. It then does not save on the stack. There is ROM allocation by the locator if it is initialised in the program. There is RAM allocation by the locator if it is not initialised in the program.
- Case (v): Modifier *static* declaration is outside a function block. It is not usable outside the class in which declared or outside the module in which declared. There is ROM allocation by the locator for the function codes.
- Case (vi): Modifier *const* declaration is outside a function block. It must be initialised by a program. For example, #define *const* Welcome_Message "There is a mail for you". There is ROM allocation by the locator.
- Case (vii): Modifier *register* declaration is inside a function block. It must be initialised by a program. For example, *'register CX'*. A CPU register is temporarily allocated when needed. There is no ROM or RAM allocation.
- Case (viii): Modifier *interrupt*. It directs the compiler to save all processor registers on entry to the function codes and restore them on return from that function. [This modifier is prefixed by an underscore, *'_interrupt'* in certain compilers.]
- Case (ix): Modifier *extern*. It directs the compiler to look for the data type declaration or the function in a module other than the one currently in use.
- Case (x): Modifier *volatile* outside a function block is a warning to the compiler that an event can change its value or that its change represents an event. An event example is an interrupt event, hardware event or inter-task communication event. For example, consider a declaration: *'volatile Boolean IntrEnable;'* in Example 4.6. It changes to false at the start of service

by a service routine, if true previously. The compiler does not perform optimization for a *volatile* variable. Let a variable be assigned, $c = 0$. Later, it is assigned $c = 1$. The compiler will ignore statement $c = 0$ during code optimisation and will take $c = 1$. But if c is an event variable, it should not be optimised. $\text{IntrEnable} = 0$ is at the beginning of the service routine in case an interrupt enable variable is used for disabling any interrupt during the period of execution of the ISR. $\text{IntrEnable} = 1$ is executed before return from the ISR. This re-enables the interrupts at the system. Declaration of IntrEnable as *volatile* directs the compiler not to optimise two assignment statements in the same function. There is no ROM or RAM allocation by the locator.

Case (xi): Modifier *volatile static* declaration is inside a function block. Examples are (a) '*volatile static* boolean $\text{RTIEnable} = \text{true};$ ' (b) '*volatile static* boolean $\text{RTISWTEnable};$ ' and (c) '*volatile static* boolean $\text{RTCSWT_F};$ ' The static declaration is for the directive to the compiler that the variable should be accessible outside the function block also, and there is to be a reserved memory space for it; and *volatile* means a directive not to optimise as an event can modify. It then does not save on the stack on context switching to another task. When several tasks are executed in cooperation, the declaration static helps. The compiler does not optimise the code due to declaration *volatile*. There is no ROM or RAM allocation by the locator.

Use of Conditions, Loops and Infinite Loops

Conditional statements are used many times. If a defined condition (s) is fulfilled, the statements within the curly braces after the condition (or a statement without the braces) are executed, otherwise the program proceeds to the next statement or to the next set of statements. Sometimes a set of statements is repeated in a loop. Generally, in case of array, the index changes and the same set is repeated for another element of the array.

Infinite loops are never desired in usual programming. Why? The program will never end and never exit or proceed further to the codes after the loop. *Infinite loop is a feature in embedded system programming!* This is clarified by the following examples. (i) What about switching off the telephone? The system software in the telephone has to be always in a waiting loop that finds the ring on the line. An exit from the loop will make the system hardware redundant. (ii) Recall Example 4.1 of inter-networking program for In_A_Out_B . Port A may give the input at any instance. The system has to execute codes up to the point where there is output at port B, and then return to receive and wait for

another input. The *hardware equivalent of an infinite loop is a ticking system clock (real time clock) or a free running counter.*

Example 5.2 gives a 'C' program design in which the program starts executing from the $\text{main}()$ function. There are calls to the functions and calls on the interrupts in between. It has to return to the start. The system main program is never in a halt state. Therefore, the $\text{main}()$ is in an infinite loop within the start and end.

Example 5.2

```
# define false 0
# define true 1
/*****/ void
main(void) {
/* The Declarations here and initialization here */
.
.
/* Infinite while loop follows. Since the condition set for the while loop is always true, the statements
within the curly braces continue to execute */
while (true) {
/* Codes that repeatedly execute */
.
.
}
/*****/
```

Assume that the function *main* does not have a waiting loop and simply passes the control to an RTOS. Consider a multitasking program. The OS can create a task. The OS can insert a task into the

list. It can delete from the list. Let an OS kernel *preemptively schedule* the running of the various listed tasks. Each task will then have the codes in an infinite loop. [Refer to Chapters 9 and 10 for understanding the various terms used here.] Example 5.3 demonstrates the infinite loops within the tasks.

How do more than one infinite loops co-exist? The code inside waits for a signal or event or a set of events that the kernel transfers to it to run the waiting task. The code inside the loop generates a message that transfers to the kernel. It is detected by the OS kernel, which passes another task message and generates another signal for that task, and preempts the previously running task. Let an event be setting of a flag, and the flag setting is to trigger the running of a task whenever the kernel passes it to the waiting task. The instruction, 'if (flag1) {...};' is to execute the task function for a service if flag1 is true.

Example 5.3

```
# define false 0
# define true 1
/***** void
main (void) {
/* Call RTOS run here */
rtos.run ( );
/* Infinite while loops follows in each task. So never there is return from the RTOS. */
}
*****/ void
task1 (....) {
/* Declarations */
.
.
while (true) {
/* Codes that repeatedly execute */
.
.
/* Codes that execute on an event*/
if (flag1) { ...; }; flag1 =0;
/* Codes that execute for message to the kernel */
message1 ( );
}
}
*****/ void
task2 (...) {
/* Declarations */
.
.
while (true) {
/* Codes that repeatedly execute */
.
.
/* Codes that execute on an event*/
if (flag2) { . ....; }; flag2 =0;
/* Codes that execute for message to the kernel */
message2 ( );
};
}
*****/
taskN (...) {
/* Declarations */
.
.
while (true) {
/* Codes that repeatedly execute */
..
/* Codes that execute on an event*/
if (flagN) { . ..; }; flagN =0;
```

```

/* Codes that execute for message to the kernel */
message2 ( );
};
}
/*****

```

Use of Pointers, NULL Pointers

Pointers are powerful tools when used correctly and according to certain basic principles. Exemplary uses are as follows. Let a byte each be stored at a memory address.

1. Let a port A in system have a buffer register that stores a byte. Now a program using a pointer declares the byte at port A as follows: 'unsigned byte *portA'. [or Pbyte *portA.] The * means 'the contents at'. This declaration means that there is a pointer and an unsigned byte for portA. The compiler will reserve one memory address for that byte. Consider 'unsigned short *timer1'. A pointer *timer1* will point to two bytes, and the compiler will reserve two memory addresses for contents of *timer1*.
2. Consider declarations as follows. void *portAdata; The void means the undefined data type for portAdata. The compiler will allocate for the *portAdata without any type check.
3. A pointer can be assigned a constant fixed address as in Example 4.5. Recall two preprocessor directives: '# define portA (volatile unsigned byte *) 0x1000' and '# define PIOC (volatile unsigned byte *) 0x1001'. Alternatively, the addresses in a function can be assigned as follows. 'volatile unsigned byte * portA = (unsigned byte *) 0x1000' and 'volatile unsigned byte *PIOC = (unsigned byte *) 0x1001'. An instruction, 'portA ++;' will make the portA pointer point to the next address and to which is the PIOC.
4. Consider, unsigned byte portAdata; unsigned byte *portA = &portAdata. The first statement directs the compiler to allocate one memory address for portAdata because there is a byte each at an address. The & (ampersand sign) means 'at the address of'. This declaration means the positive number of 8 bits (byte) pointed by portA is replaced by the byte at the address of portAdata. The right side of the expression evaluates the contained byte from the address, and the left side puts that byte at the pointed address. Since the right side variable portAdata is not a declared pointer, the ampersand sign is kept to point to its address so that the right side pointer gets the contents (bits) from that address. [Note: The equality sign in a program statement means 'is replaced by'].
5. Consider two statements, 'unsigned short *timer1;' and 'timer1++;'. The second statement adds 0x0002 in the address of timer1. Why? timer1 ++ means point to next address, and unsigned short declaration allocated two addresses for timer1. [timer1 ++; or timer1 +=1 or timer = timer +1; will have identical actions.] Therefore, the next address is 0x0002 more than the address of timer1 that was originally defined. Had the declaration been 'unsigned int' (in case of 32 bit timer), the second statement would have incremented the address by 0x0004. When the index increments by 1 in case of an array of characters, the pointer to the previous element actually increments by 1, and thus the address will increment by 0x0004 in case of an array of integers. For array data type, * is never put before the identifier name, but an index is put within a pair of square brackets after the identifier. Consider a declaration, 'unsigned char portAMessageString [80];'. The port A message is a string, which is an array of 80 characters. Now, portAMessageString is itself a pointer to an address without the star sign before it. However, *portAMessageString will now refer to all the 80 characters in the string. portAMessageString [20] will refer to the twentieth element (character) in the string. Assume that there is a list of RTCSWT (Real Time Clock interrupts triggered Software Timers) timers that are active at an instant. The top of the list can be pointed as '*RTCSWT_List.top' using the pointer. RTCSWT_List.top is now the pointer to the top of the contents in a memory for a list of the active RTCSWTs.
6. Consider the statement 'RTCSWT_List.top ++;' It increments this pointer in a loop. It *will not point* to the next top of another object in the list (another RTCSWT) but to some address that depends on the memory addresses allocated to an item in the RTCSWT_List. Let ListNow be a pointer within the memory block of the list top element. A statement '*RTCSWT_List.ListNow = *RTCSWT_List.top;' will do the following. RTCSWT_List pointer is now replaced by RTCSWT list-top pointer and now points to the next list element (object). [Note: RTCSWT_List.top ++ for pointer to the next list-object can only be used when RTCSWT_List

elements are placed in an array. This is because an array is analogous to consecutively located elements of the list at the memory. Recall Table 5.2.]

7. A NULL pointer declares as following: `#define NULL (void*) 0x0000`. [We can assign any address instead of 0x0000 that is not in use in a given hardware.] NULL pointer is very useful. Consider a statement: `while (* RTCSWT_List. ListNow -> state != NULL) { numRunning ++;}`. When a pointer to ListNow in a list of software timers that are running at present is not NULL, then only execute the set of statements in the given pair of opening and closing curly braces. One of the important uses of the NULL pointer is in a list. The last element to point to the end of a list, or to no more contents in a queue or empty stack, queue or list.

Use of Function Calls

Table 5.1 gives the meanings of the various sets of instructions in the C program. There are functions and a special function for starting the program execution, `void main (void)`. Given below are the steps to be followed when using a function in the program.

1. *Declaring a function:* Just as each variable has to have a declaration, each function must be declared. Consider an example. Declare a function as follows: `int run (int indexRTCSWT, unsigned int maxLength, unsigned int numTicks, SWT_Type swtType, SWT_Action swtAction, boolean loadEnable);`. Here `int` specifies the returned data type. The `run` is the function name. There are arguments inside the brackets. Data type of each argument is also declared. A modifier is needed to specify the data type of the returned element (variable or object) from any function. Here, the data type is specified as an integer. [A modifier for specifying the returned element may be also be *static*, *volatile*, *interrupt* and *extern*.]
2. *Defining the statements in the function:* Just as each variable has to be given the contents or value, each function must have its statements. Consider the statements of the function `run`. These are *within a pair of curly braces* as follows: `int RTCSWT:: run (int indexRTCSWT, unsigned int maxLength, unsigned int numTicks, SWT_Type swtType, SWT_Action swtAction, boolean loadEnable) {...};`. The last statement in a function is for the *return* and may also be for returning an element.
3. *Call to a function:* Consider an example: `if (delay_F == true & & SWTDelayIEnable == true) ISR_Delay ();`. There is a call on fulfilling a condition. The call can occur several times and can be repeatedly made. On each *call*, the values of the arguments given within the pair of bracket pass for use in the function statements.

(i) Passing the Values (elements)

The values are copied into the arguments of the functions. When the function is executed in this way, it does not change a variable's value at the *called* program. A function can only use the copied values in its own variables through the arguments. Consider a statement, 'run (int *indexRTCSWT*, unsigned int *maxLength*, unsigned int *numTicks*, SWT_Type *swtType*, SWT_Action *swtAction*, boolean *loadEnable*) {...}'. Function 'run' arguments *indexRTCSWT*, *maxLength*, *numTick*, *swtType*, and *loadEnable* original values in the calling program during execution of the codes will remain unchanged. The advantage is that the same values are present on return from the function. The arguments that are *passed by the values* are saved temporarily on a stack and retrieved on return from the function.

(ii) Reentrant Function

Reentrant function is usable by the several tasks and routines synchronously (at the same time). This is because all its argument values are retrievable from the stack. A function is called **reentrant function** when the following three conditions are satisfied.

1. *All the arguments pass the values and none of the argument is a pointer (address) whenever a calling function calls that function.* There is no pointer as an argument in the above example of function 'run'.
2. *When an operation is not atomic, that function should not operate on any variable, which is declared outside the function or which an interrupt service routine uses or which is a global variable but passed by reference and not passed by value as an argument into the function.* [The value of such a variable or variables, which is not local, does not save on the stack when there is call to another program.]

Recall Section 2.1 for understanding *atomic operation*. The following is an example that clarifies it further. Assume that at a server (software), there is a 32 bit variable *count* to count the number of clients (software) needing service. There is no option except to declare the *count* as a global variable that shares with all clients. Each client on a connection to a server sends a call to increment the *count*. The implementation by the assembly code for increment at that memory location is non-atomic when (i) the processor is of eight bits, and (ii) the server-compiler design is such that it does not account for the possibility of interrupt in-between the four instructions that implement the increment of 32-bit count on 8-bit processor. There will be a wrong value with the server after an instance when interrupt occurs midway during implementing an increment of *count*.

3. *That function does not call any other function that is not itself Reentrant.* Let *RTI_Count* be a global declaration. Consider an ISR, *ISR_RTI*. Let an '*RTI_Count* ++;' instruction be where the *RTI_Count* is variable for counts on a real-time clock interrupt. Here *ISR_RTI* is a not a Reentrant routine because the second condition may not be fulfilled in the given processor hardware. There is no precaution that may be taken here by the programmer against shared data problems at the address of the *RTI_Count* because there may be no operation that modifies *RTI_Counts* in any other routine or function than the *ISR_RTI*. But if there is another operation that modifies the *RTI_Count* the shared-data problem will arise. [Refer to Section 8.2.1 for a solution.]

(iii) Passing the References

When an argument value to a function passes through a pointer, the function can change this value. On returning from this function, the new value will be available in the calling program or another function called by this function. [There is no saving on stack of a value that either (a) *passes through a pointer in the function-arguments* or (b) *operates in the function as a global variable* or (c) *operates through a variable declared outside the function block*.]

Multiple Function Calls in Cyclic Order in the Main

One of the most common methods is for the multiple function-calls to be made in a cyclic order in an infinite loop of the *main*. Recall the 64 kbps network problem of Example 4.1. Let us design the C codes given in Example 5.3 for an infinite loop for this problem. Example 5.4 shows how the multiple function *calls* are defined in the main for execution in the cyclic orders. Figure 5.1 shows the model adopted here.

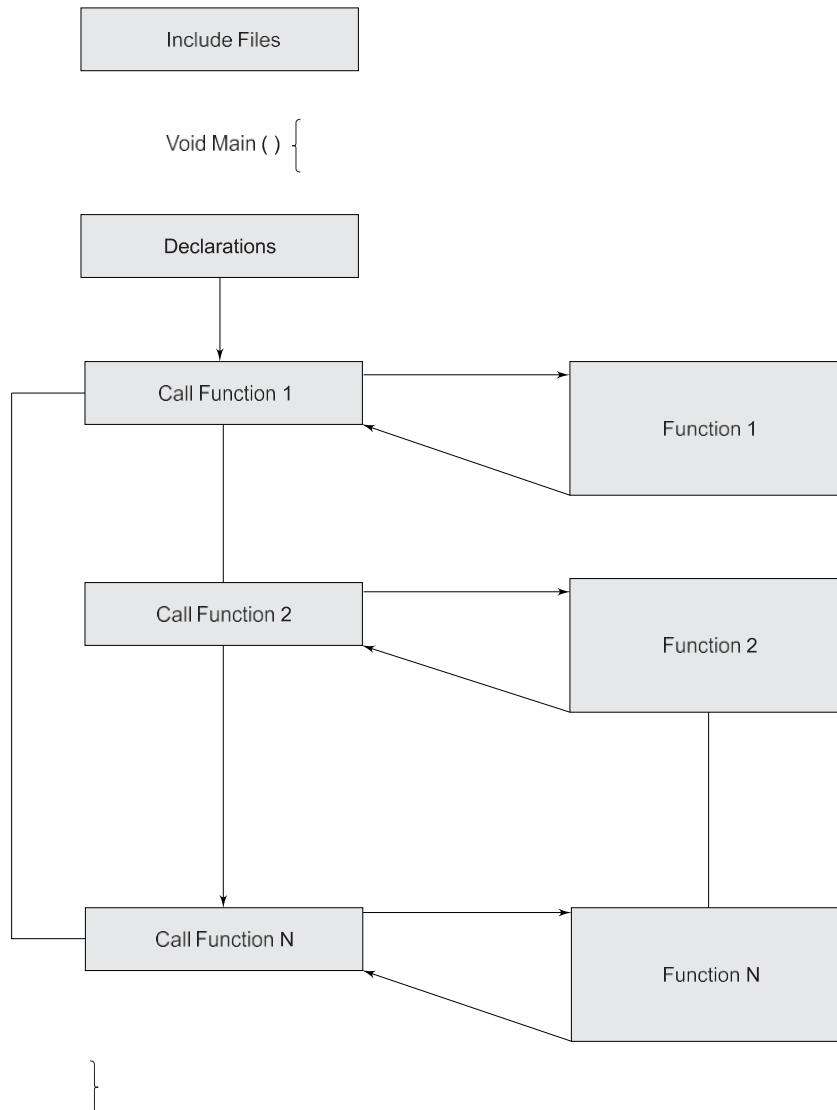


Figure 5.1

PROGRAMMING MODEL FOR MULTIPLE FUNCTION CALLS IN 'MAIN()' FUNCTION

Example 5.4

```

typedef unsigned char int8bit;
# define int8bit boolean
# define false 0
# define true 1
void main (void) {
/* The Declarations of all variables, pointers, functions here and also initializations here */
unsigned char *portAdata;
boolean charAFlag;
boolean checkPortAChar ( );
void inPortA (unsigned char *);
void decipherPortAData (unsigned char *);

```

```

void encryptPortAData (unsigned char *);
void outPortB (unsigned char *);
.
while (true) {
/* Codes that repeatedly execute */
/* Function for availability check of a character at port A*/
while (charAFlag != true) checkPortAChar ( );
/* Function for reading PortA character*/
inPortA (unsigned char *portAdata);
/* Function for deciphering */
decipherPortAData (unsigned char *portAdata);
/* Function for encoding */
encryptPortAData (unsigned char *portAdata);
/* Function for retransmit output to PortB*/
outPort B (unsigned char *portAdata);
};
}

```

5.8 EMBEDDED PROGRAMMING IN C++

Objected Oriented Programming

An *objected oriented language* is used when there is a need for re-usability of the defined object or set of objects that are common within a program or between the many *applications*. When a large program is to be made, an object-oriented language offers many advantages. *Data encapsulation, design of reusable software components* and *inheritance* are the advantages derived from the OOPs.

An object-oriented language provides for defining the objects and methods that manipulate the objects without modifying their definitions. It provides for the data and methods for encapsulation. An object can be characterised by the following:

1. An *identity* (a reference to a memory block that holds its state and behavior).
2. A *state* (its data, property, fields and attributes).
3. A *behavior* (method or methods that can manipulate the *state* of the object).

In a procedure-based language, like FORTRAN, COBOL, Pascal and C, large programs are split into simpler functional blocks and statements. In an object-oriented language like Smalltalk, C++ or Java, logical groups (also known as *classes*) are first made. Each group defines the data and the methods of using the data. A set of these groups then gives an application program. Each group has internal user-level fields for the data and the methods of processing that data at these fields. Each group can then create many objects by copying the group and making it functional. Each object is functional. Each object can interact with other objects to process the user's data. The language provides for formation of classes by the definition of a group of objects having similar attributes and common behavior. A class *creates the objects*. An *object is an instance of a class*.

Embedded Programming in C++

1. Programming advantages of C++

C++ is an *object oriented Program (OOP) language*, which in addition, supports the *procedure oriented codes of C*. Program coding in C++ codes provides the advantage of objected oriented programming as well as the advantage of C and in-line assembly. Programming concepts for embedded programming in C++ are as follows:

- (i) A class binds all the member functions together for creating objects. The objects will have memory allocation as well as default assignments to its variables that are not declared *static*. Let us assume that each software timer that gets the count input from a real time clock is an object. Now consider the codes for a C++ *class RTCSWT*. A number of software timer objects can be created as the instances of *RTCSWT*.
- (ii) A class can derive (inherit) from another class also. Creating a *child* class from *RTCSWT* as a *parent* class creates a new application of the *RTCSWT*.

- (iii) Methods (C functions) can have same name in the inherited class. This is called *method overloading*. Methods can have the same name as well as the same number and type of arguments in the inherited class. This is called *method overriding*. These are the two significant features that are extremely useful in a large program.
- (iv) Operators in C++ can be overloaded like in method overloading. Recall the following statements and expressions in Example 5.8. The operators ++ and ! are overloaded to perform a set of operations. [Usually the ++ operator is used for post-increment and pre-increment and the ! operator is used for a *not* operation.]


```
const OrderedList & operator ++ ( ) {if (ListNow != NULL) ListNow = ListNow -> pNext;
return *this;}
boolean int OrderedList & operator ! ( ) const {return (ListNow != NULL) ;};
```

[Java does not support operator overloading, except for the + operator. It is used for summation as well string-concatenation.]

There is *struct* that binds all the member functions together in C. But a C++ *class* has object features. It can be extended and child classes can be derived from it. A number of child classes can be derived from a common class. This feature is called polymorphism. A class can be declared as public or private. The data and methods access is restricted when a class is declared private. *Struct* does not have these features.

2. Disadvantages of C++

Program codes become lengthy, particularly when certain features of the standard C++ are used. Examples of these features are as follows:

- (a) Template.
- (b) Multiple Inheritance (Deriving a class from many parents).
- (c) Exceptional handling.
- (d) Virtual base classes.
- (e) Classes for IO Streams. [Two library functions are *cin* (for character (s) in) and *cout* (for character (s) out). The I/O stream class library provides for the input and output streams of characters (bytes). It supports *pipes*, *sockets* and *file management features*. Refer to Section 8.3 for the use of these in inter task communications.]

3. Can optimization codes be used in Embedded C++ programs to eliminate the disadvantages?

Embedded system codes can be optimised when using an OOP language by the following

- (a) Declare private as many classes as possible. It helps in optimising the generated codes.
- (b) Use *char*, *int* and *boolean* (scalar data types) in place of the objects (reference data types) as arguments and use local variables as much as feasible.
- (c) Recover memory already used once by changing the reference to an object to NULL.

A *special compiler for an embedded system* can facilitate the disabling of specific features provided in C++. Embedded C++ is a version of C++ that provides for a selective disabling of the above features so that there is a less runtime overhead and less runtime library. The solutions for the library functions are available and ported in C directly. The IO stream library functions in an embedded C++ compiler are also reentrant. So using embedded C++ compilers or the special compilers make the C++ a significantly more powerful coding language than C for embedded systems.

GNU C/C++ compilers (called *gcc*) find extensive use in the C++ environment in embedded software development. Embedded C++ is a new programming tool with a compiler that provides a small runtime library. It satisfies small runtime RAM needs by selectively de-configuring features like, template, multiple inheritance, virtual base class, etc. when there is a less runtime overhead and when the less runtime library using solutions are available. Selectively removed (de-configured) features could

be template, run time type identification, multiple Inheritance, exceptional handling, virtual base classes, IO streams and foundation classes. [Examples of foundation classes are GUIs (graphic user interfaces). Exemplary GUIs are the buttons, checkboxes or radios.]

An embedded system C++ compiler (other than *gcc*) is Diab compiler from Diab Data. It also provides the target (embedded system processor) specific optimisation of the codes. [Section 5.12] The run-time analysis tools check the expected run time error and give a profile that is visually interactive.



Embedded C++ is a C++ version, which makes large program development simpler by providing object-oriented programming (OOP) features of using an object, which binds state and behavior and which is defined by an instance of a class. We use objects in a way that minimises memory needs and run-time overheads in the system. Embedded system programmers use C++ due to the OOP features of software re-usability, extendibility, polymorphism, function overriding and overloading along portability of C codes and in-line assembly codes. C++ also provides for overloading of operators. A compiler, *gcc*, is popularly used for embedded C++ codes compilation. Diab compiler has two special features: (i) processor specific code optimisation and (ii) Run time analysis tools for finding expected run-time errors.

5.9 EMBEDDED PROGRAMMING IN JAVA

Java has advantages for embedded programming as follows:

1. Java is completely an OOP language.
2. Java has in-built support for creating multiple threads. [For the definition of thread and its similarity in certain respects to task refer to Section 8.1.] It obviates the need for an operating system (OS) based scheduler [Section I.5.6] for handling the tasks.
3. Java is the language for most Web applications and allows machines of different types to communicate on the Web.
4. There is a huge class library on the network that makes program development quick.
5. Platform independence in hosting the compiled codes on the network is because Java generates the byte codes. These are executed on an installed JVM (Java Virtual Machine) on a machine. [Virtual machines (VM) in embedded systems are stored at the ROM.] Platform independence gives *portability* with respect to the processor used.
6. Java does not permit pointer manipulation instructions. So it is robust in the sense that memory leaks and memory related errors do not occur. A memory leak occurs, for example, when attempting to write to the end of a bounded array.
7. Java byte codes that are generated need a larger memory when a method has more than 3 or 4 local variables.
8. Java being platform independent is expected to run on a machine with an RISC like instruction execution with few addressing modes only.

Disadvantages of Java

An embedded Java system may need a minimum of 512 kB ROM and 512 kB RAM because of the need to first install JVM and run the application.

Java Card

Use of J2ME (Java 2 Micro Edition) or Java Card or EmbeddedJava helps in reducing the code size to 8 kB for the usual applications like smart card. How? The following are the methods.

1. Use core classes only. Classes for basic run time environment form the VM internal format and only the programmer's new Java classes are not in internal format.
2. Provide for configuring the run time environment. Examples of configuring are *deleting the exception handling classes, user defined class loaders, file classes, AWT classes, synchronized threads, thread groups, multidimensional arrays, and long and floating data types*. Other configuring examples are adding the specific classes for connections when needed, datagrams, input output and streams.
3. Create one object at a time when running the multiple threads.
4. Reuse the objects instead of using a larger number of objects.
5. Use scalar types only as long as feasible.

A smart card (Section 11.4) is an electronic circuit with a memory and CPU or a synthesised VLSI circuit. It is packed like an ATM card. For smart cards, there is Java card technology. [Refer to <http://www.java.sun.com/products/javacard>.] Internal formats for the run time environments are available mainly for the few classes in Java card technology. Java classes used are the connections, datagrams, input output and streams, security and cryptography.

Described above are the advantages and disadvantages of Java applications in the embedded system. JavaCard, EmbeddedJava and J2ME (Java 2 Micro Edition) are three versions of Java that generate a reduced code size.

Consider an embedded system such as a smart card. It is a simple application that uses a running JavaCard. The Java advantage of platform independency in byte codes is an asset. The smart card connects to a remote server. The card stores the user account past balance and user details for the remote server information in an encrypted format. It deciphers and communicates to the server the user needs after identifying and certifying the user. The intensive codes for the complex application run at the server. *A restricted run time environment exists in Java classes* for connections, datagrams, character-input output and streams, security and cryptography only.

[For EmbeddedJava, refer to <http://www.sun.java.com/embeddedjava>. It provides an embedded run time environment and a closed exclusive system. Every method, class and run time library is optional].

J2ME provides the optimised run-time environment. Instead of the use of packages, J2ME provides for the codes for the core classes only. These codes are stored at the ROM of the embedded system. It provides for two alternative configurations, Connected Device Configuration (CDC) and Connected Limited Device Configurations (CLDC). CDC inherits a few classes from packages for *net*, *security*, *io*, *reflect*, *security.cert*, *text*, *text.resources*, *util*, *jar* and *zip*. CLDC does not provide for the applets, awt, beans, math, net, rmi, security and sql and text packages in java.lang. There is a separate javax.microedition.io package in CLDC configuration. A PDA (personal digital assistant) uses CDC or CLDC.

There is a scaleable OS feature in J2ME. There is a new virtual machine, KVM as an alternative to JVM. When using the KVM, the system needs a 64 kB instead of 512 kB run time environment. KVM features are as follows:

1. Use of the following data types is optional. **(i)** Multidimensional arrays, **(ii)** long 64-bit integer and **(iii)** floating points.
2. Errors are handled by the program classes, which inherit only a few needed error handling classes from the java I/O package for the Exceptions.
3. Use of a separate set of APIs (application program interfaces) instead of JINI. JINI is portable. But in the embedded system, the ROM has the application already ported and the user does not change it.
4. There is no verification of the classes. KVM presumes the classes as already validated.
5. There is no object finalization. The garbage collector does not have to do time consuming changes in the object for finalization
6. The class loader is not available to the user program. The KVM provides the loader.
7. Thread groups are not available.
8. There is no use of java.lang.reflection. Thus, there are no interfaces that do the object serialization, debugging and profiling.

J2ME need not be restricted to configure the JVM to limit the classes. The configuration can be augmented by Profiler classes. For example, MIDP (Mobile Information Device Profiler). A profile defines the support of Java to a device family. The profiler is a layer between the application and the configuration. For example, MIDP is between CLDC and application. Between the device and configuration, there is an OS, which is specific to the device needs.

A mobile information device has the followings.

1. A touch screen or keypad.
2. A minimum of 96 x 54 pixel color or monochrome display.
3. Wireless networking.
4. A minimum of 32 kB RAM, 8 kB EEPROM or flash for data and 128 kB ROM.
5. MIDP used as in PDAs, mobile phones and pagers.

MIDP classes describe the displaying text. It describes the network connectivity. For example, for HTTP, it provides support for small databases stored in EEPROM or flash memory. It schedules the applications and supports the timers.

LECTURE NOTES

ON

SOFTWARE TESTING METHODOLOGIES

III B. Tech II semester

INDEX

S.No.	Unit	Introduction: Purpose of testing	Pg.No.
1.	I	Introduction: Purpose of testing	1
2.		Dichotomies, Model for Testing,	3
3.		Consequences of Bugs, Taxonomy of Bugs	7
4.	II	Flow graphs and Path testing	15
5.		Transaction Flow Testing	34
6.	III	Dataflow Testing	38
7.		Domain Testing	59
8.	IV	Paths, Path products and Regular expressions	68
9.		Logic Based Testing	92
10.	V	State, State Graphs and Transition testing	106
		Graph Matrices and Applications	112

UNIT- I

INTRODUCTION

What is testing?

Testing is the process of exercising or evaluating a system or system components by manual or automated means to verify that it satisfies specified requirements.

The Purpose of Testing

Testing consumes at least half of the time and work required to produce a functional program.

- MYTH: Good programmers write code without bugs. (It's wrong!!!)
- History says that even well written programs still have 1-3 bugs per hundred statements.

Productivity and Quality in Software:

- In production of consumer goods and other products, every manufacturing stage is subjected to quality control and testing from component to final stage.
- If flaws are discovered at any stage, the product is either discarded or cycled back for rework and correction.
- Productivity is measured by the sum of the costs of the material, the rework, and the discarded components, and the cost of quality assurance and testing.
- There is a tradeoff between quality assurance costs and manufacturing costs: If sufficient time is not spent in quality assurance, the reject rate will be high and so will be the net cost. If inspection is good and all errors are caught as they occur, inspection costs will dominate, and again the net cost will suffer.
- Testing and Quality assurance costs for 'manufactured' items can be as low as 2% in consumer products or as high as 80% in products such as space-ships, nuclear reactors, and aircrafts, where failures threaten life. Whereas the manufacturing cost of software is trivial.
- The biggest part of software cost is the cost of bugs: the cost of detecting them, the cost of correcting them, the cost of designing tests that discover them, and the cost of running those tests.
- For software, quality and productivity are indistinguishable because the cost of a software copy is trivial.
- Testing and Test Design are parts of quality assurance should also focus on bug prevention. A prevented bug is better than a detected and corrected bug.

Phases in a tester's mental life:

Phases in a tester's mental life can be categorized into the following 5 phases:

1. **Phase 0: (Until 1956: Debugging Oriented)** There is no difference between testing and debugging. Phase thinking was the norm in early days of software development till testing emerged as a discipline.
2. **Phase 1: (1957-1978: Demonstration Oriented)** the purpose of testing here is to show that software works. Highlighted during the late 1970s. This failed because the probability of showing that software works 'decreases' as testing increases. I.e. the more you test, the more likely you will find a bug.
3. **Phase 2: (1979-1982: Destruction Oriented)** the purpose of testing is to show that software doesn't work. This also failed because the software will never get released as you will find one bug or the other. Also, a bug corrected may also lead to another bug.
4. **Phase 3: (1983-1987: Evaluation Oriented)** the purpose of testing is not to prove anything but to reduce the perceived risk of not working to an acceptable value (Statistical Quality Control). Notion is that testing does improve the product to the extent that testing catches bugs and to the extent that those bugs are fixed. The product is released when the confidence on that product is high enough. (Note: This is applied to large software products with millions of code and years of use.)
5. **Phase 4: (1988-2000: Prevention Oriented)** Testability is the factor considered here. One reason is to reduce the labor of testing. Other reason is to check the testable and non-testable code. Testable code has fewer bugs than the code that's hard to test. Identifying the testing techniques to test the code is the main key here.

Test Design:

We know that the software code must be designed and tested, but many appear to be unaware that tests themselves must be designed and tested. Tests should be properly designed and tested before applying it to the actual code.

Testing isn't everything:

There are approaches other than testing to create better software. Methods other than testing include:

1. **Inspection Methods:** Methods like walkthroughs, desk checking, formal inspections and code reading appear to be as effective as testing but the bugs caught don't completely overlap.
2. **Design Style:** While designing the software itself, adopting stylistic objectives such as testability, openness and clarity can do much to prevent bugs.
3. **Static Analysis Methods:** Includes formal analysis of source code during compilation. In earlier days, it is a routine job of the programmer to do that. Now, the compilers have taken over that job.
4. **Languages:** The source language can help reduce certain kinds of bugs. Programmers find new bugs while using new languages.

5. **Development Methodologies and Development Environment:** The development process and the environment in which that methodology is embedded can prevent many kinds of bugs.

Dichotomies:

- **Testing Versus Debugging:**

Many people consider both as same. Purpose of testing is to show that a program has bugs. The purpose of testing is to find the error or misconception that led to the program's failure and to design and implement the program changes that correct the error.

Debugging usually follows testing, but they differ as to goals, methods and most important psychology. The below table shows few important differences between testing and debugging.

Testing	Debugging
Testing starts with known conditions, uses predefined procedures and has predictable outcomes.	Debugging starts from possibly unknown initial conditions and the end cannot be predicted except statistically.
Testing can and should be planned, designed and scheduled.	Procedure and duration of debugging cannot be so constrained.
Testing is a demonstration of error or apparent correctness.	Debugging is a deductive process.
Testing proves a programmer's failure.	Debugging is the programmer's vindication (Justification).
Testing, as executes, should strive to be predictable, dull, constrained, rigid and inhuman.	Debugging demands intuitive leaps, experimentation and freedom.
Much testing can be done without design knowledge.	Debugging is impossible without detailed design knowledge.
Testing can often be done by an outsider.	Debugging must be done by an insider.
Much of test execution and design can be automated.	Automated debugging is still a dream.

- **Function versus Structure:**

- Tests can be designed from a functional or a structural point of view.
- In **Functional testing**, the program or system is treated as a black box. It is subjected to inputs, and its outputs are verified for conformance to specified behavior. Functional testing takes the user point of view- bother about functionality and features and not the program's implementation.

- o In **Structural testing** does look at the implementation details. Things such as programming style, control method, source language, database design, and coding details dominate structural testing.

- o Both Structural and functional tests are useful, both have limitations, and both target different kinds of bugs. Functional tests can detect all bugs but would take infinite time to do so. Structural tests are inherently finite but cannot detect all errors even if completely executed.

- **Designer versus Tester:**

- o Test designer is the person who designs the tests where as the tester is the one actually tests the code. During functional testing, the designer and tester are probably different persons. During unit testing, the tester and the programmer merge into one person.
- o Tests designed and executed by the software designers are by nature biased towards structural consideration and therefore suffer the limitations of structural testing.

- **Modularity versus Efficiency:**

A module is a discrete, well-defined, small component of a system. Smaller the modules, difficult to integrate; larger the modules, difficult to understand. Both tests and systems can be modular. Testing can and should likewise be organized into modular components. Small, independent test cases can be designed to test independent modules.

- **Small versus Large:**

Programming in large means constructing programs that consists of many components written by many different programmers. Programming in the small is what we do for ourselves in the privacy of our own offices. Qualitative and Quantitative changes occur with size and so must testing methods and quality criteria.

- **Builder versus Buyer:**

Most software is written and used by the same organization. Unfortunately, this situation is dishonest because it clouds accountability. If there is no separation between builder and buyer, there can be no accountability.

- The different roles / users in a system include:
 1. **Builder:** Who designs the system and is accountable to the buyer.
 2. **Buyer:** Who pays for the system in the hope of profits from providing services?
 3. **User:** Ultimate beneficiary or victim of the system. The user's interests are also guarded by.
 4. **Tester:** Who is dedicated to the builder's destruction?
 5. **Operator:** Who has to live with the builders' mistakes, the buyers' murky (unclear) specifications, testers' oversights and the users' complaints?

MODEL FOR TESTING:

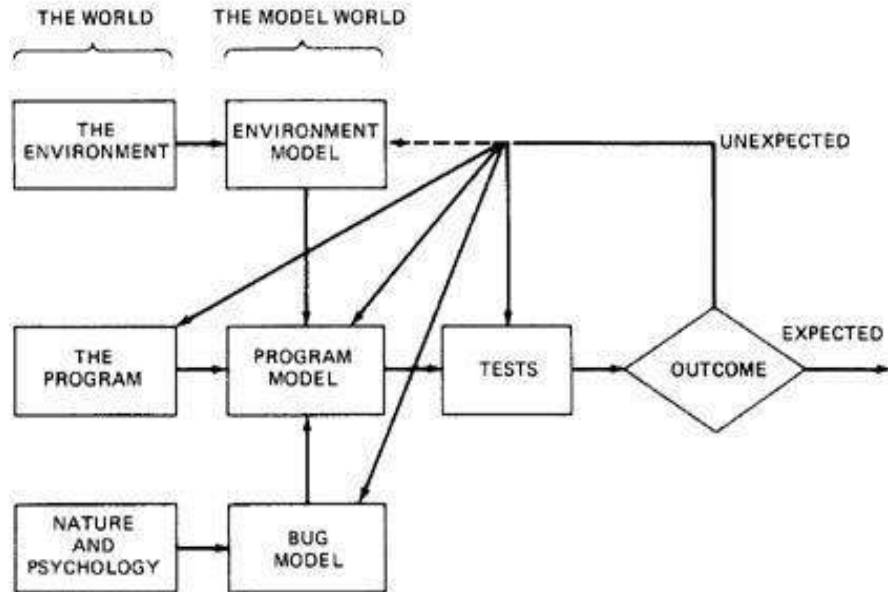


Figure 1.1: A Model for Testing

Above figure is a model of testing process. It includes three models: A model of the environment, a model of the program and a model of the expected bugs.

- **Environment:**

- A Program's environment is the hardware and software required to make it run. For online systems, the environment may include communication lines, other systems, terminals and operators.
- The environment also includes all programs that interact with and are used to create the program under test - such as OS, linkage editor, loader, compiler, utility routines.
- Because the hardware and firmware are stable, it is not smart to blame the environment for bugs.

- **Program:**

- Most programs are too complicated to understand in detail.
- The concept of the program is to be simplified in order to test it.
- If simple model of the program doesn't explain the unexpected behavior, we may have to modify that model to include more facts and details. And if that fails, we may have to modify the program.

- **Bugs:**

- Bugs are more insidious (deceiving but harmful) than ever we expect them to be.
- An unexpected test result may lead us to change our notion of what a bug is and our model of bugs.
- Some optimistic notions that many programmers or testers have about bugs are usually unable to test effectively and unable to justify the dirty tests most programs need.
- **Optimistic notions about bugs:**
 1. **Benign Bug Hypothesis:** The belief that bugs are nice, tame and logical. (Benign: Not Dangerous)

2. **Bug Locality Hypothesis:** The belief that a bug discovered with in a component affects only that component's behavior.
3. **Control Bug Dominance:** The belief those errors in the control structures (if, switch etc) of programs dominate the bugs.
4. **Code / Data Separation:** The belief that bugs respect the separation of code and data.
5. **Lingua Salvatore Est.:** The belief that the language syntax and semantics (e.g. Structured Coding, Strong typing, etc) eliminates most bugs.
6. **Corrections Abide:** The mistaken belief that a corrected bug remains corrected.
7. **Silver Bullets:** The mistaken belief that X (Language, Design method, representation, environment) grants immunity from bugs.
8. **Sadism Suffices:** The common belief (especially by independent tester) that a sadistic streak, low cunning, and intuition are sufficient to eliminate most bugs. Tough bugs need methodology and techniques.
9. **Angelic Testers:** The belief that testers are better at test design than programmers is at code design.

- **Tests:**

- Tests are formal procedures, Inputs must be prepared, Outcomes should predict, tests should be documented, commands need to be executed, and results are to be observed. All these errors are subjected to error

- **We do three distinct kinds of testing on a typical software system. They are:**

1. **Unit / Component Testing:** A **Unit** is the smallest testable piece of software that can be compiled, assembled, linked, loaded etc. A unit is usually the work of one programmer and consists of several hundred or fewer lines of code. **Unit Testing** is the testing we do to show that the unit does not satisfy its functional specification or that its implementation structure does not match the intended design structure. A **Component** is an integrated aggregate of one or more units. **Component Testing** is the testing we do to show that the component does not satisfy its functional specification or that its implementation structure does not match the intended design structure.
2. **Integration Testing:** **Integration** is the process by which components are aggregated to create larger components. **Integration Testing** is testing done to show that even though the components were individually satisfactory (after passing component testing), checks the combination of components are incorrect or inconsistent.
3. **System Testing:** A **System** is a big component. **System Testing** is aimed at revealing bugs that cannot be attributed to components. It includes testing for performance, security, accountability, configuration sensitivity, startup and recovery.

- **Role of Models:** The art of testing consists of creating, selecting, exploring, and revising models. Our ability to go through this process depends on the number of different models we have at hand and their ability to express a program's behavior.

CONSEQUENCES OF BUGS:

- **Importance of bugs:** The importance of bugs depends on frequency, correction cost, installation cost, and consequences.
 1. **Frequency:** How often does that kind of bug occur? Pay more attention to the more frequent bug types.
 2. **Correction Cost:** What does it cost to correct the bug after it is found? The cost is the sum of 2 factors: (1) the cost of discovery (2) the cost of correction. These costs go up dramatically later in the development cycle when the bug is discovered. Correction cost also depends on system size.
 3. **Installation Cost:** Installation cost depends on the number of installations: small for a single user program but more for distributed systems. Fixing one bug and distributing the fix could exceed the entire system's development cost.
 4. **Consequences:** What are the consequences of the bug? Bug consequences can range from mild to catastrophic.

A reasonable metric for bug importance is

Importance= (\$) = Frequency * (Correction cost + Installation cost + Consequential cost)

- **Consequences of bugs:** The consequences of a bug can be measure in terms of human rather than machine. Some consequences of a bug on a scale of one to ten are:
 1. **Mild:** The symptoms of the bug offend us aesthetically (gently); a misspelled output or a misaligned printout.
 2. **Moderate:** Outputs are misleading or redundant. The bug impacts the system's performance.
 3. **Annoying:** The system's behavior because of the bug is dehumanizing. *E.g.* Names are truncated or arbitrarily modified.
 4. **Disturbing:** It refuses to handle legitimate (authorized / legal) transactions. The ATM won't give you money. My credit card is declared invalid.
 5. **Serious:** It loses track of its transactions. Not just the transaction itself but the fact that the transaction occurred. Accountability is lost.
 6. **Very Serious:** The bug causes the system to do the wrong transactions. Instead of losing your paycheck, the system credits it to another account or converts deposits to withdrawals.
 7. **Extreme:** The problems aren't limited to a few users or to few transaction types. They are frequent and arbitrary instead of sporadic infrequent) or for unusual cases.
 8. **Intolerable:** Long term unrecoverable corruption of the database occurs and the corruption is not easily discovered. Serious consideration is given to shutting the system down.
 9. **Catastrophic:** The decision to shut down is taken out of our hands because the system fails.
 10. **Infectious:** What can be worse than a failed system? One that corrupt other systems even though it does not fall in itself ; that erodes the social physical environment; that melts nuclear reactors and starts war.

- **Flexible severity rather than absolutes:**

- Quality can be measured as a combination of factors, of which number of bugs and their severity is only one component.
- Many organizations have designed and used satisfactory, quantitative, quality metrics.
- Because bugs and their symptoms play a significant role in such metrics, as testing progresses, you see the quality rise to a reasonable value which is deemed to be safe to ship the product.
- The factors involved in bug severity are:
 1. **Correction Cost:** Not so important because catastrophic bugs may be corrected easier and small bugs may take major time to debug.
 2. **Context and Application Dependency:** Severity depends on the context and the application in which it is used.
 3. **Creating Culture Dependency:** What's important depends on the creators of software and their cultural aspirations. Test tool vendors are more sensitive about bugs in their software than games software vendors.
 4. **User Culture Dependency:** Severity also depends on user culture. Naive users of PC software go crazy over bugs where as pros (experts) may just ignore.
 5. **The software development phase:** Severity depends on development phase. Any bugs gets more severe as it gets closer to field use and more severe the longer it has been around.

TAXONOMY OF BUGS:

- There is no universally correct way to categorize bugs. The taxonomy is not rigid.
 - A given bug can be put into one or another category depending on its history and the programmer's state of mind.
 - The major categories are: (1) Requirements, Features and Functionality Bugs (2) Structural Bugs (3) Data Bugs (4) Coding Bugs (5) Interface, Integration and System Bugs (6) Test and Test Design Bugs.
- ✓ **Requirements, Features and Functionality Bugs:** Various categories in Requirements, Features and Functionality bugs include:

1. Requirements and Specifications Bugs:

- Requirements and specifications developed from them can be incomplete, ambiguous, or self-contradictory. They can be misunderstood or impossible to understand.
- The specifications that don't have flaws in them may change while the design is in progress. The features are added, modified and deleted.
- Requirements, especially, as expressed in specifications are a major source of expensive bugs.
- The range is from a few percentages to more than 50%, depending on the application and environment.
- What hurts most about the bugs is that they are the earliest to invade the system and the last to leave.

2. Feature Bugs:

- Specification problems usually create corresponding feature problems.
- A feature can be wrong, missing, or superfluous (serving no useful purpose). A missing feature or case is easier to detect and correct. A wrong feature could have deep design implications.
- Removing the features might complicate the software, consume more resources, and foster more bugs.

3. Feature Interaction Bugs:

- Providing correct, clear, implementable and testable feature specifications is not enough.
- Features usually come in groups or related features. The features of each group and the interaction of features within the group are usually well tested.
- The problem is unpredictable interactions between feature groups or even between individual features. For example, your telephone is provided with call holding and call forwarding. The interactions between these two features may have bugs.
- Every application has its peculiar set of features and a much bigger set of unspecified feature interaction potentials and therefore results in feature interaction bugs.

Specification and Feature Bug Remedies:

- Most feature bugs are rooted in human to human communication problems. One solution is to use high-level, formal specification languages or systems.
- Such languages and systems provide short term support but in the long run, does not solve the problem.
- **Short term Support:** Specification languages facilitate formalization of requirements and inconsistency and ambiguity analysis.
- **Long term Support:** Assume that we have a great specification language and that can be used to create unambiguous, complete specifications with unambiguous complete tests and consistent test criteria.
- The specification problem has been shifted to a higher level but not eliminated.

Testing Techniques for functional bugs: Most functional test techniques- that is those techniques which are based on a behavioral description of software, such as transaction flow testing, syntax testing, domain testing, logic testing and state testing are useful in testing functional bugs.

✓ **Structural bugs:** Various categories in Structural bugs include:

1. Control and Sequence Bugs:

- Control and sequence bugs include paths left out, unreachable code, improper nesting of loops, loop-back or loop termination criteria incorrect, missing process steps, duplicated processing, unnecessary processing, rampaging, GOTO's, ill-conceived (not properly planned) switches, spaghetti code, and worst of all, pachinko code.
- One reason for control flow bugs is that this area is amenable (supportive) to theoretical treatment.
- Most of the control flow bugs are easily tested and caught in unit testing.
- Another reason for control flow bugs is that use of old code especially ALP & COBOL code are dominated by control flow bugs.

- Control and sequence bugs at all levels are caught by testing, especially structural testing, more specifically path testing combined with a bottom line functional test based on a specification.

2. Logic Bugs:

- Bugs in logic, especially those related to misunderstanding how case statements and logic operators behave singly and combinations
- Also includes evaluation of Boolean expressions in deeply nested IF-THEN-ELSE constructs.
- If the bugs are parts of logical (i.e. boolean) processing not related to control flow, they are characterized as processing bugs.
- If the bugs are parts of a logical expression (i.e. control-flow statement) which is used to direct the control flow, then they are categorized as control-flow bugs.

3. Processing Bugs:

- Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection and general processing.
- Examples of Processing bugs include: Incorrect conversion from one data representation to other, ignoring overflow, improper use of greater-than-or-equal etc
- Although these bugs are frequent (12%), they tend to be caught in good unit testing.

4. Initialization Bugs:

- Initialization bugs are common. Initialization bugs can be improper and superfluous.
- Superfluous bugs are generally less harmful but can affect performance.
- Typical initialization bugs include: Forgetting to initialize the variables before first use, assuming that they are initialized elsewhere, initializing to the wrong format, representation or type etc
- Explicit declaration of all variables, as in Pascal, can reduce some initialization problems.

5. Data-Flow Bugs and Anomalies:

- Most initialization bugs are special case of data flow anomalies.
- A data flow anomaly occurs where there is a path along which we expect to do something unreasonable with data, such as using an uninitialized variable, attempting to use a variable before it exists, modifying and then not storing or using the result, or initializing twice without an intermediate use.

✓ Data bugs:

- Data bugs include all bugs that arise from the specification of data objects, their formats, the number of such objects, and their initial values.
- Data Bugs are at least as common as bugs in code, but they are often treated as if they did not exist at all.
- Code migrates data: Software is evolving towards programs in which more and more of the control and processing functions are stored in tables.
- Because of this, there is an increasing awareness that bugs in code are only half the battle and the data problems should be given equal attention.

Dynamic Data Vs Static data:

- Dynamic data are transitory. Whatever their purpose their lifetime is relatively short, typically the processing time of one transaction. A storage object may be used to hold dynamic data of different types, with different formats, attributes and residues.
- Dynamic data bugs are due to leftover garbage in a shared resource. This can be handled in one of the three ways: (1) Clean up after the use by the user (2) Common Cleanup by the resource manager (3) No Clean up
- Static Data are fixed in form and content. They appear in the source code or database directly or indirectly, for example a number, a string of characters, or a bit pattern.
- Compile time processing will solve the bugs caused by static data.

Information, parameter, and control:

Static or dynamic data can serve in one of three roles, or in combination of roles: as a parameter, for control, or for information.

Content, Structure and Attributes:

- Content can be an actual bit pattern, character string, or number put into a data structure. Content is a pure bit pattern and has no meaning unless it is interpreted by a hardware or software processor. All data bugs result in the corruption or misinterpretation of content.
- **Structure** relates to the size, shape and numbers that describe the data object, which is memory location used to store the content. (E.g. A two dimensional array).
- **Attributes** relates to the specification meaning that is the semantics associated with the contents of a data object. (E.g. an integer, an alphanumeric string, a subroutine). The severity and subtlety of bugs increases as we go from content to attributes because the things get less formal in that direction.

✓ Coding bugs:

- Coding errors of all kinds can create any of the other kind of bugs.
- Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntax checking.
- If a program has many syntax errors, then we should expect many logic and coding bugs.
- The documentation bugs are also considered as coding bugs which may mislead the maintenance programmers.

✓ Interface, integration, and system bugs:

Various categories of bugs in Interface, Integration, and System Bugs are:

1. External Interfaces:

- The external interfaces are the means used to communicate with the world.
- These include devices, actuators, sensors, input terminals, printers, and communication lines.
- The primary design criterion for an interface with outside world should be robustness.

- All external interfaces, human or machine should employ a protocol. The protocol may be wrong or incorrectly implemented.
- Other external interface bugs are: invalid timing or sequence assumptions related to external signals
- Misunderstanding external input or output formats.
- Insufficient tolerance to bad input data.

2. Internal Interfaces:

- Internal interfaces are in principle not different from external interfaces but they are more controlled.
- A best example for internal interfaces is communicating routines.
- The external environment is fixed and the system must adapt to it but the internal environment, which consists of interfaces with other components, can be negotiated.
- Internal interfaces have the same problem as external interfaces.

3. Hardware Architecture:

- Bugs related to hardware architecture originate mostly from misunderstanding how the hardware works.
- Examples of hardware architecture bugs: address generation error, i/o device operation / instruction error, waiting too long for a response, incorrect interrupt handling etc.
- The remedy for hardware architecture and interface problems is twofold: (1) Good Programming and Testing (2) Centralization of hardware interface software in programs written by hardware interface specialists.

4. Operating System Bugs:

- Program bugs related to the operating system are a combination of hardware architecture and interface bugs mostly caused by a misunderstanding of what it is the operating system does.
- Use operating system interface specialists, and use explicit interface modules or macros for all operating system calls.
- This approach may not eliminate the bugs but at least will localize them and make testing easier.

5. Software Architecture:

- Software architecture bugs are the kind that called - interactive.
- Routines can pass unit and integration testing without revealing such bugs.
- Many of them depend on load, and their symptoms emerge only when the system is stressed.
- Sample for such bugs: Assumption that there will be no interrupts, Failure to block or un block interrupts, Assumption that memory and registers were initialized or not initialized etc
- Careful integration of modules and subjecting the final system to a stress test are effective methods for these bugs.

6. Control and Sequence Bugs (Systems Level):

These bugs include: Ignored timing, Assuming that events occur in a specified sequence, Working on data before all the data have arrived from disc, Waiting for an impossible combination of prerequisites, Missing, wrong, redundant or superfluous process steps.

The remedy for these bugs is highly structured sequence control. Specialize, internal, sequence control mechanisms are helpful.

7. Resource Management Problems:

- Memory is subdivided into dynamically allocated resources such as buffer blocks, queue blocks, task control blocks, and overlay buffers.
- External mass storage units such as disc's, are subdivided into memory resource pools.
- Some resource management and usage bugs: Required resource not obtained, Wrong resource used, Resource is already in use, Resource dead lock etc
- **Resource Management Remedies:** A design remedy that prevents bugs is always preferable to a test method that discovers them.
- The design remedy in resource management is to keep the resource structure simple: the fewest different kinds of resources, the fewest pools, and no private resource management.

8. Integration Bugs:

- Integration bugs are bugs having to do with the integration of, and with the interfaces between, working and tested components.
- These bugs results from inconsistencies or incompatibilities between components.
- The communication methods include data structures, call sequences, registers, semaphores, and communication links and protocols results in integration bugs.
- The integration bugs do not constitute a big bug category (9%) they are expensive category because they are usually caught late in the game and because they force changes in several components and/or data structures.

9. System Bugs:

- System bugs covering all kinds of bugs that cannot be ascribed to a component or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating systems.
- There can be no meaningful system testing until there has been thorough component and integration testing.
- System bugs are infrequent (1.7%) but very important because they are often found only after the system has been fielded.

✓ TEST AND TEST DESIGN BUGS:

- Testing: testers have no immunity to bugs. Tests require complicated scenarios and databases.
- They require code or the equivalent to execute and consequently they can have bugs.
- Test criteria: if the specification is correct, it is correctly interpreted and implemented, and a proper test has been designed; but the criterion by which the software's behavior is

judged may be incorrect or impossible. So, a proper test criteria has to be designed. The more complicated the criteria, the likelier they are to have bugs.

Remedies: The remedies of test bugs are:

- 1. Test Debugging:** The first remedy for test bugs is testing and debugging the tests. Test debugging, when compared to program debugging, is easier because tests, when properly designed are simpler than programs and do not have to make concessions to efficiency.
- 2. Test Quality Assurance:** Programmers have the right to ask how quality in independent testing is monitored.
- 3. Test Execution Automation:** The history of software bug removal and prevention is indistinguishable from the history of programming automation aids. Assemblers, loaders, compilers are developed to reduce the incidence of programming and operation errors. Test execution bugs are virtually eliminated by various test execution automation tools.
- 4. Test Design Automation:** Just as much of software development has been automated, much test design can be and has been automated. For a given productivity rate, automation reduces the bug count - be it for software or be it for tests.

UNIT II

FLOW GRAPHS, PATH TESTING AND TRANSACTION FLOW TESTING

BASICS OF PATH TESTING:

- **Path Testing:**
 - Path Testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through the program.
 - If the set of paths are properly chosen then we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement has been executed at least once.
 - Path testing techniques are the oldest of all structural test techniques.
 - Path testing is most applicable to new software for unit testing. It is a structural technique.
 - It requires complete knowledge of the program's structure.
 - It is most often used by programmers to unit test their own code.
 - The effectiveness of path testing rapidly deteriorates as the size of the software aggregate under test increases.
 - **The Bug Assumption:**
 - The bug assumption for the path testing strategies is that something has gone wrong with the software that makes it take a different path than intended.
 - As an example "GOTO X" where "GOTO Y" had been intended.
 - Structured programming languages prevent many of the bugs targeted by path testing: as a consequence the effectiveness for path testing for these languages is reduced and for old code in COBOL, ALP, FORTRAN and Basic, the path testing is indispensable.
 - **Control Flow Graphs:**
 - The control flow graph is a graphical representation of a program's control structure. It uses the elements named process blocks, decisions, and junctions.
 - The flow graph is similar to the earlier flowchart, with which it is not to be confused.
 - **Flow Graph Elements:** A flow graph contains four different types of elements.
(1) Process Block (2) Decisions (3) Junctions (4) Case Statements
- 1. Process Block:**
- A process block is a sequence of program statements uninterrupted by either decisions or junctions.
 - It is a sequence of statements such that if any one of statement of the block is executed, then all statement thereof are executed.
 - Formally, a process block is a piece of straight line code of one statement or hundreds of statements.
- A process has one entry and one exit. It can consist of a single statement or instruction, a sequence of statements or instructions, a single entry/exit subroutine, a macro or function call, or a sequence of these.

2. Decisions:

- A decision is a program point at which the control flow can diverge.
- Machine language conditional branch and conditional skip instructions are examples of decisions.
- Most of the decisions are two-way but some are three way branches in control flow.

3. Case Statements:

- A case statement is a multi-way branch or decisions.
- Examples of case statement are a jump table in assembly language, and the PASCAL case statement.
- From the point of view of test design, there are no differences between Decisions and Case Statements

4. Junctions:

- A junction is a point in the program where the control flow can merge.
- Examples of junctions are: the target of a jump or skip instruction in ALP, a label that is a target of GOTO.

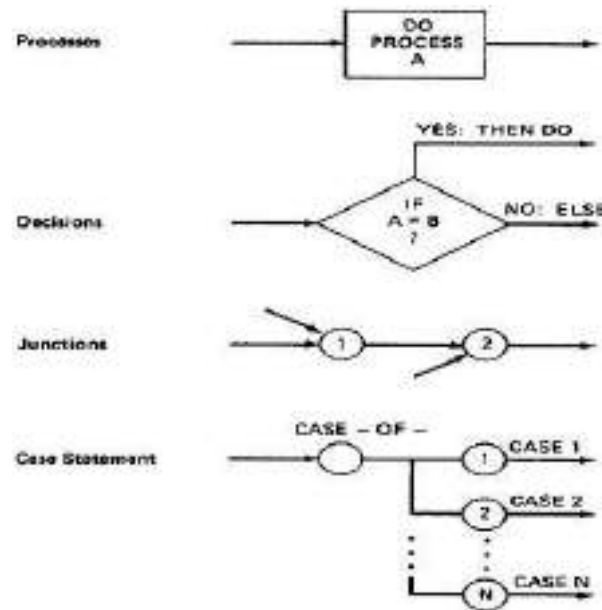


Figure 2.1: Flow graph Elements

Control Flow Graphs Vs Flowcharts:

- o A program's flow chart resembles a control flow graph.
- o In flow graphs, we don't show the details of what is in a process block.
- o In flow charts every part of the process block is drawn.
- o The flowchart focuses on process steps, where as the flow graph focuses on control flow of the program.
- o The act of drawing a control flow graph is a useful tool that can help us clarify the control flow and data flow issues.

Notational Evolution:

The control flow graph is simplified representation of the program's structure. The notation changes made in creation of control flow graphs:

- o The process boxes weren't really needed. There is an implied process on every line joining junctions and decisions.
- o We don't need to know the specifics of the decisions, just the fact that there is a branch.
- o The specific target label names aren't important-just the fact that they exist. So we can replace them by simple numbers.
- o To understand this, we will go through an example (Figure 2.2) written in a FORTRAN like programming language called **Programming Design Language (PDL)**. The program's corresponding flowchart (Figure 2.3) and flow graph (Figure 2.4) were also provided below for better understanding.
- o The first step in translating the program to a flowchart is shown in Figure 2.3, where we have the typical one-for-one classical flowchart. Note that complexity has increased, clarity has decreased, and that we had to add auxiliary labels (LOOP, XX, and YY), which have no actual program counterpart. In Figure 2.4 we merged the process steps and replaced them with the single process box.
- o We now have a control flow graph. But this representation is still too busy. We simplify the notation further to achieve Figure 2.5, where for the first time we can really see what the control flow looks like.

```
CODE* (PDL)

INPUT X, Y
Z := X + Y
V := X - Y
IF Z >= 0 GOTO SAM
JOE: Z := Z - 1
SAM: Z := Z + V
FOR U = 0 TO Z
  V(U), U(V) := (Z + V) * U
  IF V(U) = 0 GOTO JOE
  Z := Z - 1
  IF Z = 0 GOTO ELL
  U := U + 1
NEXT U

V(U-1) := V(U+1) + U(V-1)
ELL: V(U+U(V)) := U + V
IF U = V GOTO JOE
IF U > V THEN U := Z
Z := U
END
```

* A contrived horror

Figure 2.2: Program Example (PDL)

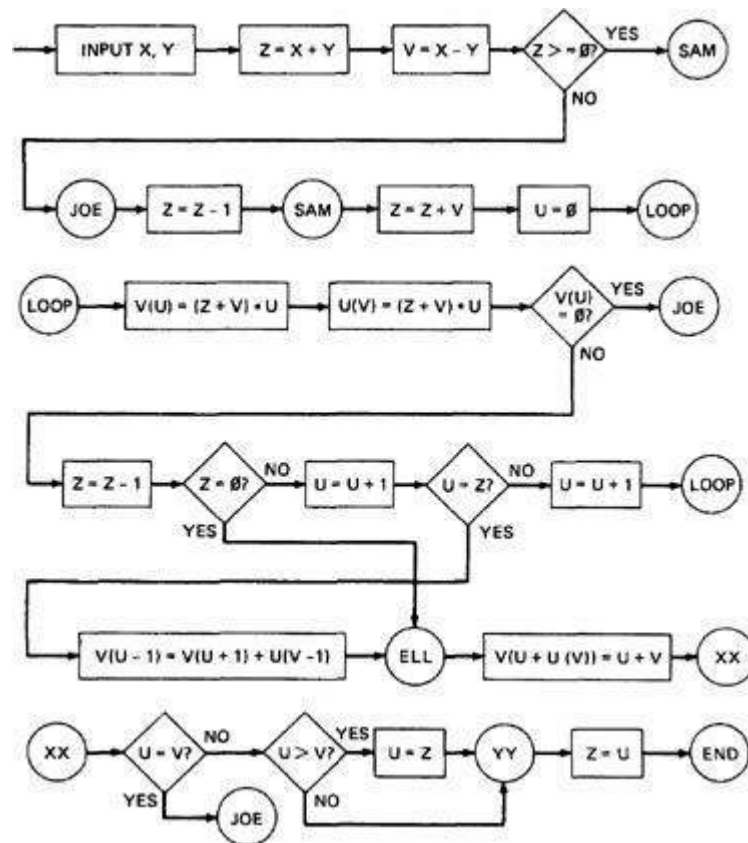


Figure 2.3: One-to-one flowchart for example program in Figure 2.2

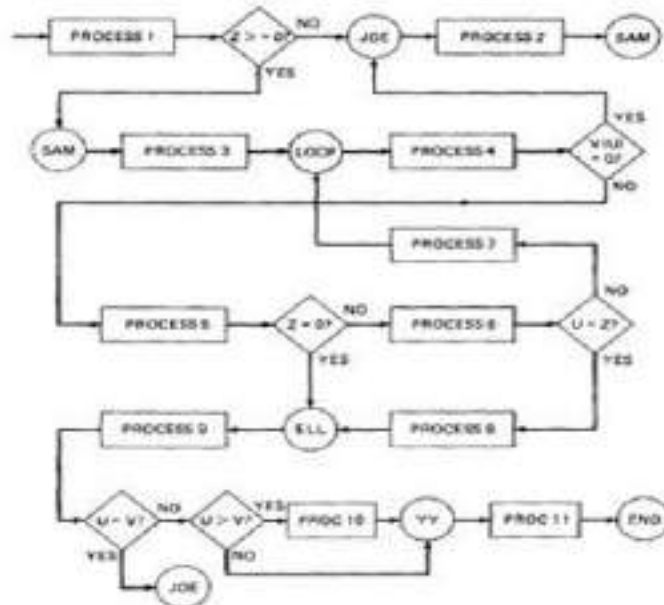


Figure 2.4: Control Flow graph for example in Figure 2.2

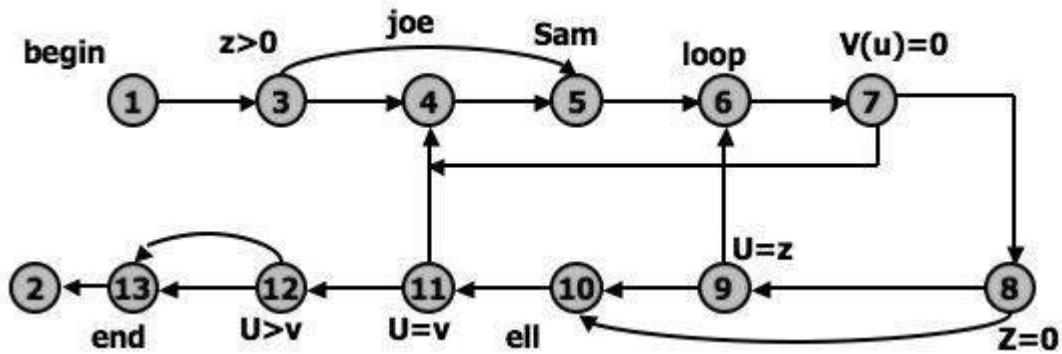


Figure 2.5: Simplified Flow graph Notation

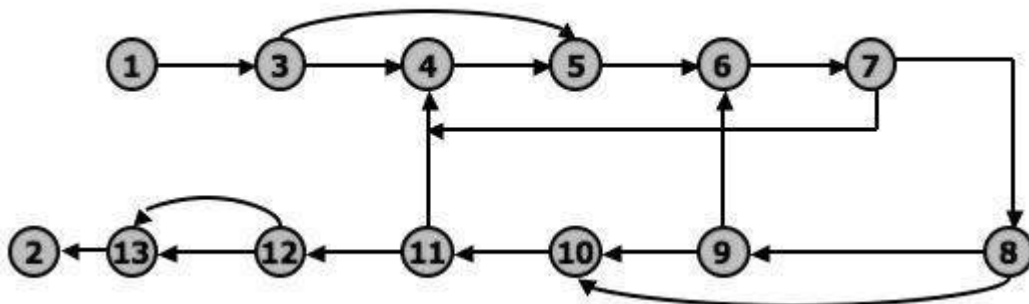


Figure 2.6: Even Simplified Flow graph Notation

The final transformation is shown in Figure 2.6, where we've dropped the node numbers to achieve an even simpler representation. The way to work with control flow graphs is to use the simplest possible representation - that is, no more information than you need to correlate back to the source program or PDL.

LINKED LIST REPRESENTATION:

Although graphical representations of flow graphs are revealing, the details of the control flow inside a program they are often inconvenient.

In linked list representation, each node has a name and there is an entry on the list for each link in the flow graph. Only the information pertinent to the control flow is shown.

Linked List representation of Flow Graph:

1 (BEGIN)	: 3	
2 (END)	:	Exit, no outlink
3 (Z>0?)	: 4 (FALSE)	
	: 5 (TRUE)	
4 (JOE)	: 5	
5 (SAM)	: 6	
6 (LOOP)	: 7	
7 (V(U)=0?)	: 4 (TRUE)	
	: 8 (FALSE)	
8 (Z=0?)	: 9 (FALSE)	
	:10 (TRUE)	
9 (U=Z?)	: 6 (FALSE) = LOOP	
	:10 (TRUE) = ELL	
10 (ELL)	:11	
11 (U=V?)	: 4 (TRUE) = JOE	
	:12 (FALSE)	
12 (U>V?)	:13 (TRUE)	
	:13 (FALSE)	
13	: 2 (END)	

Figure 2.7: Linked List Control Flow graph Notation FLOWGRAPH -

PROGRAM CORRESPONDENCE:

A flow graph is a pictorial representation of a program and not the program itself, just as a topographic map.

You can't always associate the parts of a program in a unique way with flow graph parts because many program structures, such as if-then-else constructs, consists of a combination of decisions, junctions, and processes.

The translation from a flow graph element to a statement and vice versa is not always unique. (See Figure 2.8)

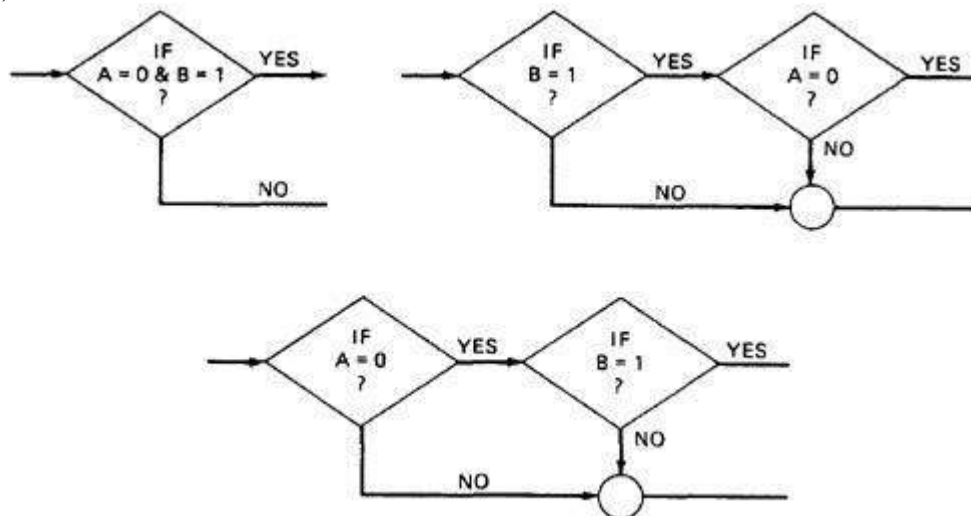


Figure 2.8: Alternative Flow graphs for same logic (Statement "IF (A=0) AND (B=1) THEN ...").

An improper translation from flow graph to code during coding can lead to bugs, and improper translation during the test design lead to missing test cases and causes undiscovered bugs.

FLOWGRAPH AND FLOWCHART GENERATION:

Flowcharts can be

1. Handwritten by the programmer.
2. Automatically produced by a flowcharting program based on a mechanical analysis of the source code.
3. Semi automatically produced by a flow charting program based in part on structural analysis of the source code and in part on directions given by the programmer.

There are relatively few control flow graph generators.

PATH TESTING - PATHS, NODES AND LINKS:

Path: A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same junction, decision, or exit.

- o A path may go through several junctions, processes, or decisions, one or more times.
- o Paths consist of segments.
- o The segment is a link - a single process that lies between two nodes.
- o A path segment is succession of consecutive links that belongs to some path.
- o The length of path measured by the number of links in it and not by the number of the instructions or statements executed along that path.
- o The name of a path is the name of the nodes along the path.

FUNDAMENTAL PATH SELECTION CRITERIA:

There are many paths between the entry and exit of a typical routine.

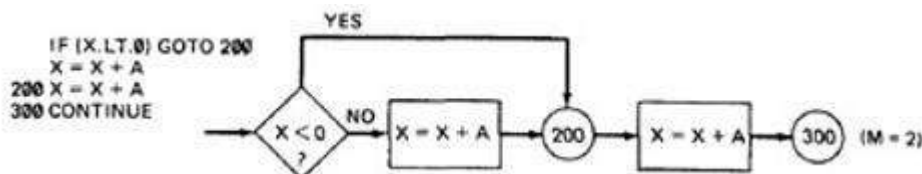
Every decision doubles the number of potential paths. And every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.

Defining complete testing:

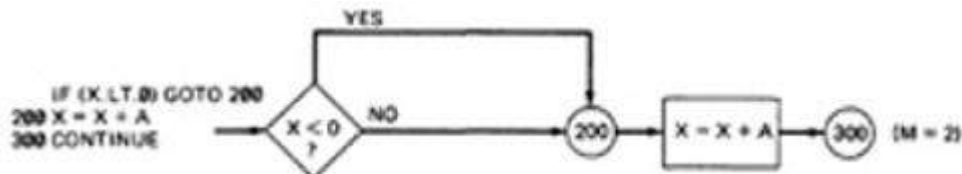
1. Exercise every path from entry to exit.
2. Exercise every statement or instruction at least once.
3. Exercise every branch and case statement, in each direction at least once.

If prescription 1 is followed then 2 and 3 are automatically followed. But it is impractical for most routines. It can be done for the routines that have no loops, in which it is equivalent to 2 and 3 prescriptions.

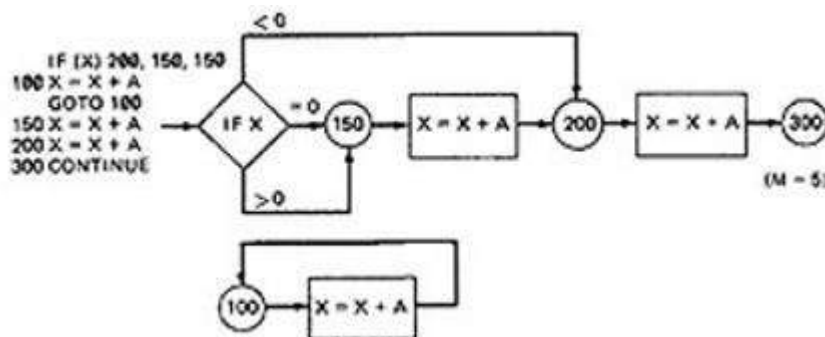
EXAMPLE: Here is the correct version.



For X negative, the output is $X + A$, while for X greater than or equal to zero, the output is $X + 2A$. Following prescription 2 and executing every statement, but not every branch, would not reveal the bug in the following incorrect version:



A negative value produces the correct answer. Every statement can be executed, but if the test cases do not force each branch to be taken, the bug can remain hidden. The next example uses a test based on executing each branch but does not force the execution of all statements:



The hidden loop around label 100 is not revealed by tests based on prescription 3 alone because no test forces the execution of statement 100 and the following GOTO statement. Furthermore, label 100 is not flagged by the compiler as an unreferenced label and the subsequent GOTO does not refer to an undefined label.

A **Static Analysis** (that is, an analysis based on examining the source code or structure) cannot determine whether a piece of code is or is not reachable. There could be subroutine calls with parameters that are subroutine labels, or in the above example there could be a GOTO that targeted label 100 but could never achieve a value that would send the program to that label.

Only a **Dynamic Analysis** (that is, an analysis based on the code's behavior while running - which is to say, to all intents and purposes, testing) can determine whether code is reachable or not and therefore distinguish between the ideal structure we think we have and the actual, buggy structure.

PATH TESTING CRITERIA:

Any testing strategy based on paths must at least both exercise every instruction and take branches in all directions.

A set of tests that does this is not complete in an absolute sense, but it is complete in the sense that anything less must leave something untested.

So we have explored three different testing criteria or strategies out of a potentially infinite family of strategies.

i. **Path Testing (P_{inf}):**

1. Execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through the program.
2. If we achieve this prescription, we are said to have achieved 100% path coverage. This is the strongest criterion in the path testing strategy family: it is generally impossible to achieve.

ii. **Statement Testing (P_1):**

1. Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved 100% statement coverage.
2. An alternate equivalent characterization is to say that we have achieved 100% node coverage. We denote this by C1.
3. This is the weakest criterion in the family: testing less than this for new software is unconscionable (unprincipled or cannot be accepted) and should be criminalized.

iii. **Branch Testing (P_2):**

1. Execute enough tests to assure that every branch alternative has been exercised at least once under some test.
2. If we do enough tests to achieve this prescription, then we have achieved 100% branch coverage.
3. An alternative characterization is to say that we have achieved 100% link coverage.
4. For structured software, branch testing and therefore branch coverage strictly includes statement coverage.
5. We denote branch coverage by C2.

Commonsense and Strategies:

- Branch and statement coverage are accepted today as the minimum mandatory testing requirement.
- The question "why not use a judicious sampling of paths?, what is wrong with leaving some code, untested?" is ineffectual in the view of common sense and experience since: **(1.)** Not testing a piece of a code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs. **(2.)** The high probability paths are always thoroughly tested if only to demonstrate that the system works properly.
- **Which paths to be tested?** You must pick enough paths to achieve C1+C2. The question of what is the fewest number of such paths is interesting to the designer of test tools that help automate the path testing, but it is not crucial to the pragmatic (practical) design of tests. It is better to make many simple paths than a few complicated paths.

▪ Path Selection Example:

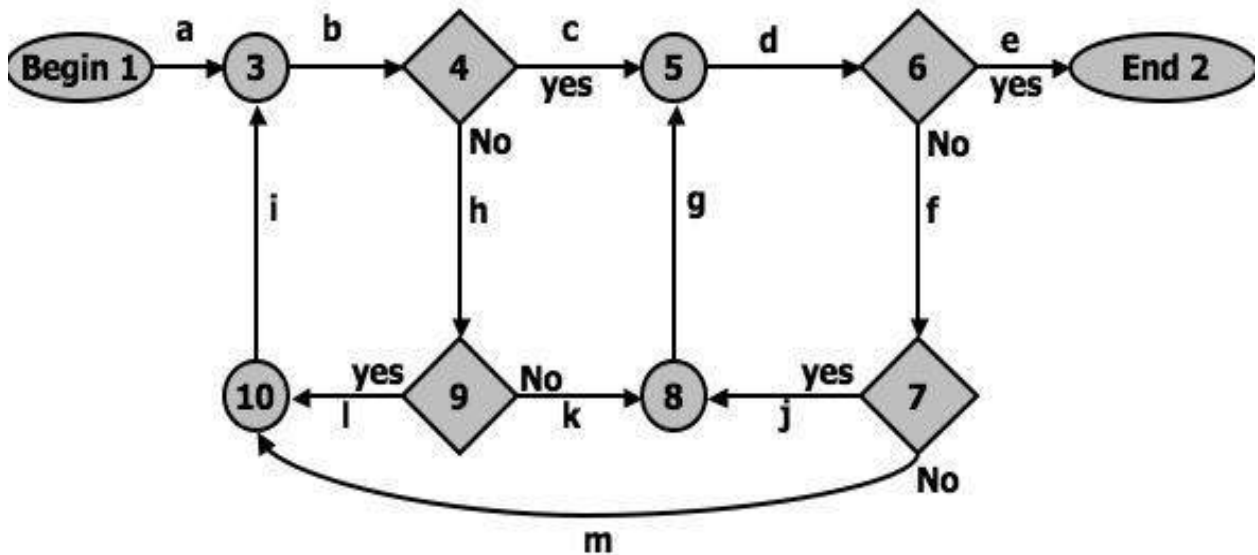


Figure 2.9: An example flow graph to explain path selection Practical

Suggestions in Path Testing:

1. Draw the control flow graph on a single sheet of paper.
2. Make several copies - as many as you will need for coverage (C1+C2) and several more.
3. Use a yellow highlighting marker to trace paths. Copy the paths onto master sheets.
4. Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1+C2.
5. As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.
6. The above paths lead to the following table considering Figure 2.9:

PATHS	DECISIONS				PROCESS-LINK												
	4	6	7	9	a	b	c	d	e	f	g	h	i	j	k	l	m
abode	YES	YES			✓	✓	✓	✓	✓								
abhkgde	NO	YES		NO	✓	✓		✓	✓		✓	✓			✓		
abhlibcde	NO,YES	YES		YES	✓	✓	✓	✓	✓			✓	✓			✓	
abodfjgde	YES	NO,YES	YES		✓	✓	✓	✓	✓	✓	✓			✓			
abodfmibcde	YES	NO,YES	NO		✓	✓	✓	✓	✓	✓			✓				✓

7. After you have traced a covering path set on the master sheet and filled in the table for every path, check the following:

1. Does every decision have a YES and a NO in its column? (C2)
2. Has every case of all case statements been marked? (C2)
3. Is every three - way branch (less, equal, greater) covered? (C2)
4. Is every link (process) covered at least once? (C1)

8. Revised Path Selection Rules:

- Pick the simplest, functionally sensible entry/exit path.
- Pick additional paths as small variation from previous paths. Pick paths that do not have loops rather than paths that do. Favor short paths that make sense over paths that don't.
- Pick additional paths that have no obvious functional meaning only if it's necessary to provide coverage.
- Be comfortable with your chosen paths. Play your hunches (guesses) and give your intuition free reign as long as you achieve C1+C2.
- Don't follow rules slavishly (blindly) - except for coverage.

LOOPS:

Cases for a single loop: A Single loop can be covered with two cases: Looping and Not looping. But, experience shows that many loop-related bugs are not discovered by C1+C2. Bugs hide themselves in corners and congregate at boundaries - in the cases of loops, at or around the minimum or maximum number of times the loop can be iterated. The minimum number of iterations is often zero, but it need not be.

CASE 1: Single loop, Zero minimum, N maximum, No excluded values

1. Try bypassing the loop (zero iterations). If you can't, you either have a bug, or zero is not the minimum and you have the wrong case.
2. Could the loop-control variable be negative? Could it appear to specify a negative number of iterations? What happens to such a value?
3. One pass through the loop.
4. Two passes through the loop.
5. A typical number of iterations, unless covered by a previous test.
6. One less than the maximum number of iterations.
7. The maximum number of iterations.
8. Attempt one more than the maximum number of iterations. What prevents the loop-control variable from having this value? What will happen with this value if it is forced?

CASE 2: Single loop, Non-zero minimum, No excluded values

1. Try one less than the expected minimum. What happens if the loop control variable's value is less than the minimum? What prevents the value from being less than the minimum?
2. The minimum number of iterations.
3. One more than the minimum number of iterations.
4. Once, unless covered by a previous test.
5. Twice, unless covered by a previous test.
6. A typical value.
7. One less than the maximum value.
8. The maximum number of iterations.
9. Attempt one more than the maximum number of iterations.

CASE 3: Single loops with excluded values

- Treat single loops with excluded values as two sets of tests consisting of loops without excluded values, such as case 1 and 2 above.
- Example, the total range of the loop control variable was 1 to 20, but that values 7, 8,9,10 were excluded. The two sets of tests are 1-6 and 11-20.
- The test cases to attempt would be 0,1,2,4,6,7 for the first range and 10,11,15,19,20,21 for the second range.

Kinds of Loops: There are only three kinds of loops with respect to path testing:

- **Nested Loops:**

The number of tests to be performed on nested loops will be the exponent of the tests performed on single loops.

As we cannot always afford to test all combinations of nested loops' iterations values. Here's a tactic used to discard some of these values:

1. Start at the inner most loop. Set all the outer loops to their minimum values.
2. Test the minimum, minimum+1, typical, maximum-1, and maximum for the innermost loop, while holding the outer loops at their minimum iteration parameter values. Expand the tests as required for out of range and excluded values.
3. If you've done the outmost loop, GOTO step 5, else move out one loop and set it up as in step 2 with all other loops set to typical values.
4. Continue outward in this manner until all loops have been covered.
5. Do all the cases for all loops in the nest simultaneously.

- **Concatenated Loops:**

Concatenated loops fall between single and nested loops with respect to test cases. Two loops are concatenated if it's possible to reach one after exiting the other while still on a path from entrance to exit.

If the loops cannot be on the same path, then they are not concatenated and can be treated as individual loops.

- **Horrible Loops:**

A horrible loop is a combination of nested loops, the use of code that jumps into and out of loops, intersecting loops, hidden loops, and cross connected loops.

Makes iteration value selection for test cases an awesome and ugly task, which is another reason such structures should be avoided.

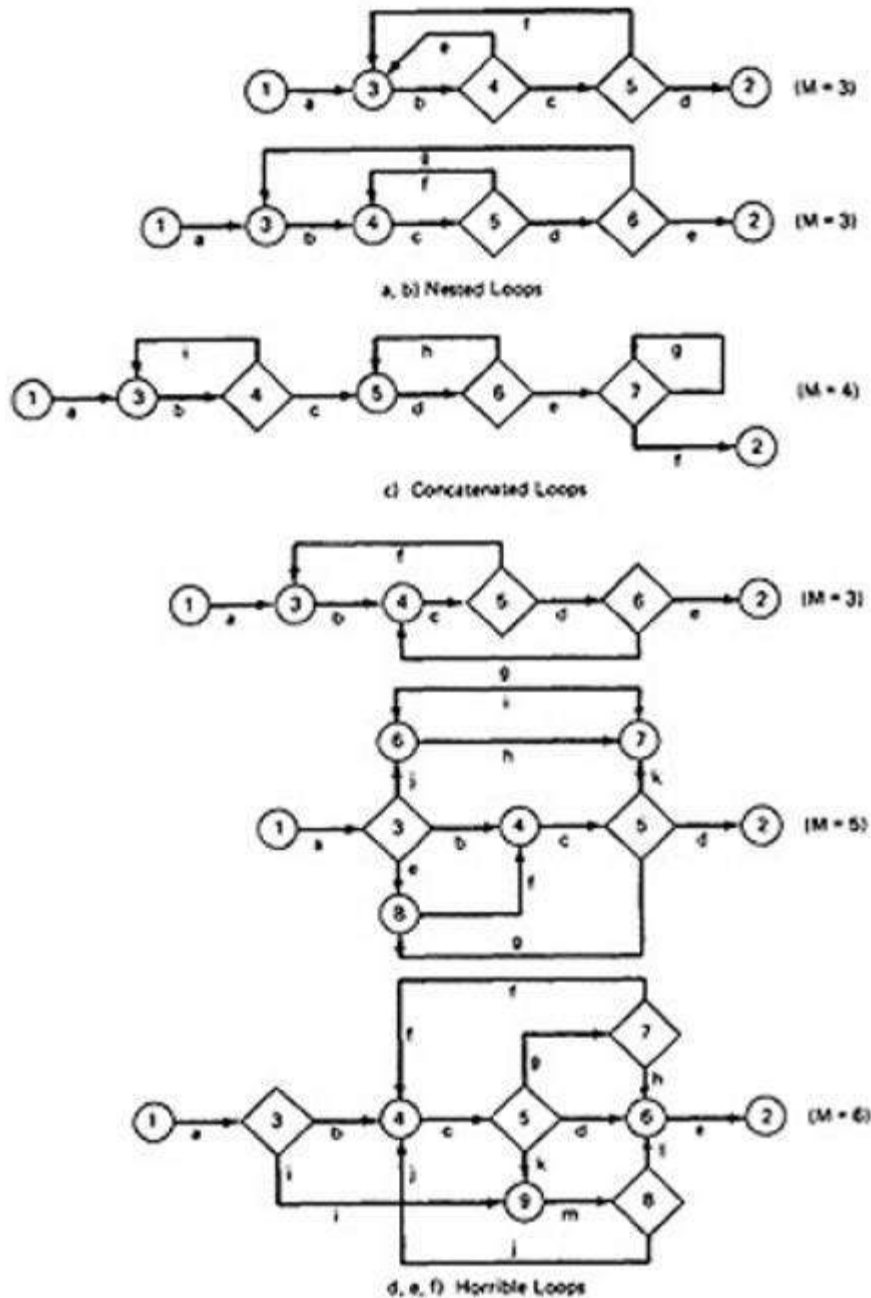


Figure 2.10: Example of Loop types

Loop Testing Time:

Any kind of loop can lead to long testing time, especially if all the extreme value cases are to attempted (Max-1, Max, Max+1).

- This situation is obviously worse for nested and dependent concatenated loops.
- Consider nested loops in which testing the combination of extreme values lead to long test times. Several options to deal with:
- Prove that the combined extreme cases are hypothetically possible, they are not possible in the real world

- Put in limits or checks that prevent the combined extreme cases. Then you have to test the software that implements such safety measures.

PREDICATES, PATH PREDICATES AND ACHIEVABLE PATHS:

PREDICATE: The logical function evaluated at a decision is called Predicate. The direction taken at a decision depends on the value of decision variable. Some examples are: $A > 0$, $x + y \geq 90$

PATH PREDICATE: A predicate associated with a path is called a Path Predicate. For example, "x is greater than zero", " $x + y \geq 90$ ", "w is either negative or equal to 10 is true" is a sequence of predicates whose truth values will cause the routine to take a specific path.

MULTIWAY BRANCHES:

- The path taken through a multiway branch such as a computed GOTO's, case statement, or jump tables cannot be directly expressed in TRUE/FALSE terms.
- Although, it is possible to describe such alternatives by using multi valued logic, an expedient (practical approach) is to express multiway branches as an equivalent set of if..then..else statements.
- For example a three way case statement can be written as: If case=1 DO A1 ELSE (IF Case=2 DO A2 ELSE DO A3 ENDIF)ENDIF.

INPUTS:

- In testing, the word input is not restricted to direct inputs, such as variables in a subroutine call, but includes all data objects referenced by the routine whose values are fixed prior to entering it.
- For example, inputs in a calling sequence, objects in a data structure, values left in registers, or any combination of object types.
- The input for a particular test is mapped as a one dimensional array called as an Input Vector.

PREDICATE INTERPRETATION:

- The simplest predicate depends only on input variables.
- For example if x_1, x_2 are inputs, the predicate might be $x_1 + x_2 \geq 7$, given the values of x_1 and x_2 the direction taken through the decision is based on the predicate is determined at input time and does not depend on processing.
- Another example, assume a predicate $x_1 + y \geq 0$ that along a path prior to reaching this predicate we had the assignment statement $y = x_2 + 7$. Although our predicate depends on processing, we can substitute the symbolic expression for y to obtain an equivalent predicate $x_1 + x_2 + 7 \geq 0$.
- The act of symbolic substitution of operations along the path in order to express the predicate solely in terms of the input vector is called **predicate interpretation**.
- Sometimes the interpretation may depend on the path; for example, INPUT X
ON X GOTO A, B, C, ...

A: Z := 7 @ GOTO HEM B: Z := -7 @ GOTO HEM C: Z := 0 @ GOTO HEM

.....

HEM: DO SOMETHING

.....

HEN: IF Y + Z > 0 GOTO ELL ELSE GOTO EMM

The predicate interpretation at HEN depends on the path we took through the first multiway branch. It yields for the three cases respectively, if $Y+7>0$, $Y-7>0$, $Y>0$.

- The path predicates are the specific form of the predicates of the decisions along the selected path after interpretation.

INDEPENDENCE OF VARIABLES AND PREDICATES:

- The path predicates take on truth values based on the values of input variables, either directly or indirectly.
- If a variable's value does not change as a result of processing, that variable is independent of the processing.
- If the variable's value can change as a result of the processing, the variable is process dependent.
- A predicate whose truth value can change as a result of the processing is said to be **process dependent** and one whose truth value does not change as a result of the processing is **process independent**.
- Process dependence of a predicate does not always follow from dependence of the input variables on which that predicate is based.

CORRELATION OF VARIABLES AND PREDICATES:

Two variables are correlated if every combination of their values cannot be independently specified.

Variables whose values can be specified independently without restriction are called uncorrelated.

A pair of predicates whose outcomes depend on one or more variables in common are said to be correlated predicates.

For example, the predicate $X==Y$ is followed by another predicate $X+Y == 8$. If we select X and Y values to satisfy the first predicate, we might have forced the 2nd predicate's truth value to change.

- Every path through a routine is achievable only if all the predicates in that routine are uncorrelated.

PATH PREDICATE EXPRESSIONS:

- A path predicate expression is a set of boolean expressions, all of which must be satisfied to achieve the selected path.
- Example:
 $X1+3X2+17 \geq 0$ $X3=17$
 $X4-X1 \geq 14X2$

- Any set of input values that satisfy all of the conditions of the path predicate expression will force the routine to the path.
- Sometimes a predicate can have an OR in it.
- Example:

A: $X5 > 0$	E: $X6 < 0$
B: $X1 + 3X2 + 17$	B: $X1 + 3X2 + 17$
≥ 0	≥ 0
C: $X3 = 17$	C: $X3 = 17$
D: $X4 - X1 \geq 14X2$	D: $X4 - X1 \geq 14X2$

- Boolean algebra notation to denote the boolean expression:
 $ABCD + EBCD = (A + E)BCD$

PREDICATE COVERAGE:

- **Compound Predicate:** Predicates of the form A OR B, A AND B and more complicated Boolean expressions are called as compound predicates.
- Sometimes even a simple predicate becomes compound after interpretation. Example: the predicate if (x=17) whose opposite branch is if x.NE.17 which is equivalent to $x > 17$. Or. $X < 17$.
- Predicate coverage is being the achieving of all possible combinations of truth values corresponding to the selected path have been explored under some test.
- As achieving the desired direction at a given decision could still hide bugs in the associated predicates

TESTING BLINDNESS:

- Testing Blindness is a pathological (harmful) situation in which the desired path is achieved for the wrong reason.
- There are three types of Testing Blindness:

□ Assignment Blindness:

- Assignment blindness occurs when the buggy predicate appears to work correctly because the specific value chosen for an assignment statement works with both the correct and incorrect predicate.
- For Example:

Correct	Buggy
X = 7	X = 7
.....
if Y > 0	if X+Y > 0
then ...	then ...

- If the test case sets Y=1 the desired path is taken in either case, but there is still a bug.

□ Equality Blindness:

- Equality blindness occurs when the path selected by a prior predicate results in a value that works both for the correct and buggy predicate.

- For Example:

Correct	Buggy
if Y = 2 then if X+Y > 3 then ...	if Y = 2 then if X > 1 then ...

- The first predicate if y=2 forces the rest of the path, so that for any positive value of x. the path taken at the second predicate will be the same for the correct and buggy version.

□ **Self Blindness:**

- Self blindness occurs when the buggy predicate is a multiple of the correct predicate and as a result is indistinguishable along that path.
- For Example:

Correct	Buggy
X = A if X-1 > 0 then ...	X = A if X+A-2 > 0 then ...

1. The assignment (x=a) makes the predicates multiples of each other, so the direction taken is the same for the correct and buggy version.

□ **PATH SENSITIZING:**

○ **Review: achievable and unachievable paths:**

1. We want to select and test enough paths to achieve a satisfactory notion of test completeness such as C1+C2.
2. Extract the programs control flow graph and select a set of tentative covering paths.
3. For any path in that set, interpret the predicates along the path as needed to express them in terms of the input vector. In general individual predicates are compound or may become compound as a result of interpretation.
4. Trace the path through, multiplying the individual compound predicates to achieve a boolean expression such as

$$(A+BC) (D+E) (FGH) (IJ) (K) (L) (L).$$

5. Multiply out the expression to achieve a sum of products form:

$$ADFGHIJKL+AEFGHIJKL+BCDFGHIJKL+BCEFGHIJKL$$

6. Each product term denotes a set of inequalities that if solved will yield an input vector that will drive the routine along the designated path.
7. Solve any one of the inequality sets for the chosen path and you have found a set of input values for the path.
8. If you can find a solution, then the path is achievable.
9. If you can't find a solution to any of the sets of inequalities, the path is unachievable.
10. The act of finding a set of solutions to the path predicate expression is called **PATH SENSITIZATION**.

○ HEURISTIC PROCEDURES FOR SENSITIZING PATHS:

1. This is a workable approach, instead of selecting the paths without considering how to sensitize, attempt to choose a covering path set that is easy to sensitize and pick hard to sensitize paths only as you must to achieve coverage.
2. Identify all variables that affect the decision.
3. Classify the predicates as dependent or independent.
4. Start the path selection with uncorrelated, independent predicates.
5. If coverage has not been achieved using independent uncorrelated predicates, extend the path set using correlated predicates.
6. If coverage has not been achieved extend the cases to those that involve dependent predicates.
7. Last, use correlated, dependent predicates.

□ PATH INSTRUMENTATION:

1. Path instrumentation is what we have to do to confirm that the outcome was achieved by the intended path.
2. **Co-incident Correctness:** The coincidental correctness stands for achieving the desired outcome for wrong reason.

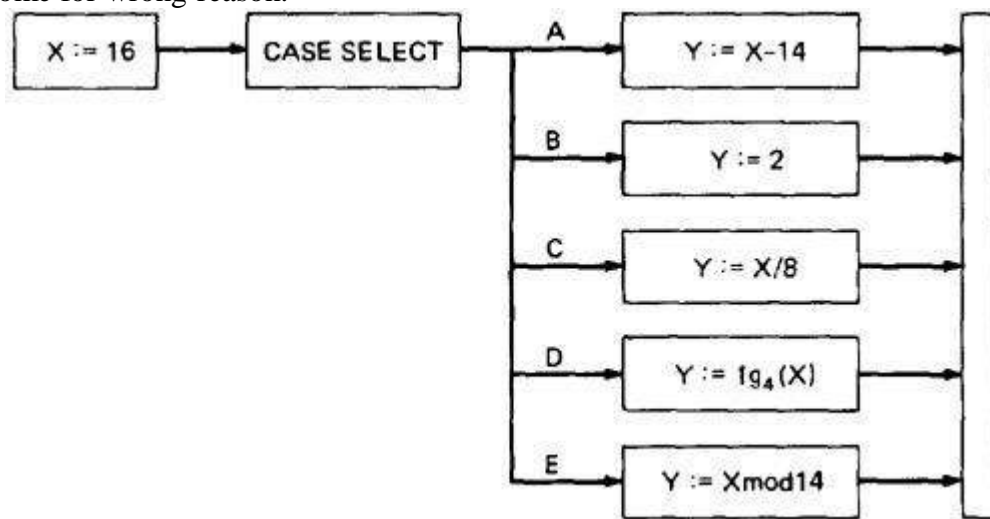


Figure 2.11: Coincidental Correctness

The above figure is an example of a routine that, for the (unfortunately) chosen input value ($X = 16$), yields the same outcome ($Y = 2$) no matter which case we select. Therefore, the tests chosen this way will not tell us whether we have achieved coverage. For example, the five cases could be totally jumbled and still the outcome would be the same. **Path Instrumentation** is what we have to do to confirm that the outcome was achieved by the intended path.

- The types of instrumentation methods include:

1. Interpretive Trace Program:

- An interpretive trace program is one that executes every statement in order and records the intermediate values of all calculations, the statement labels traversed etc.
- If we run the tested routine under a trace, then we have all the information we need to confirm the outcome and, furthermore, to confirm that it was achieved by the intended path.
- The trouble with traces is that they give us far more information than we need. In fact, the typical trace program provides so much information that confirming the path from its massive output dump is more work than simulating the computer by hand to confirm the path.

2. Traversal Marker or Link Marker:

- A simple and effective form of instrumentation is called a traversal marker or link marker.
- Name every link by a lower case letter.
- Instrument the links so that the link's name is recorded when the link is executed.
- The succession of letters produced in going from the routine's entry to its exit should, if there are no bugs, exactly correspond to the path name.

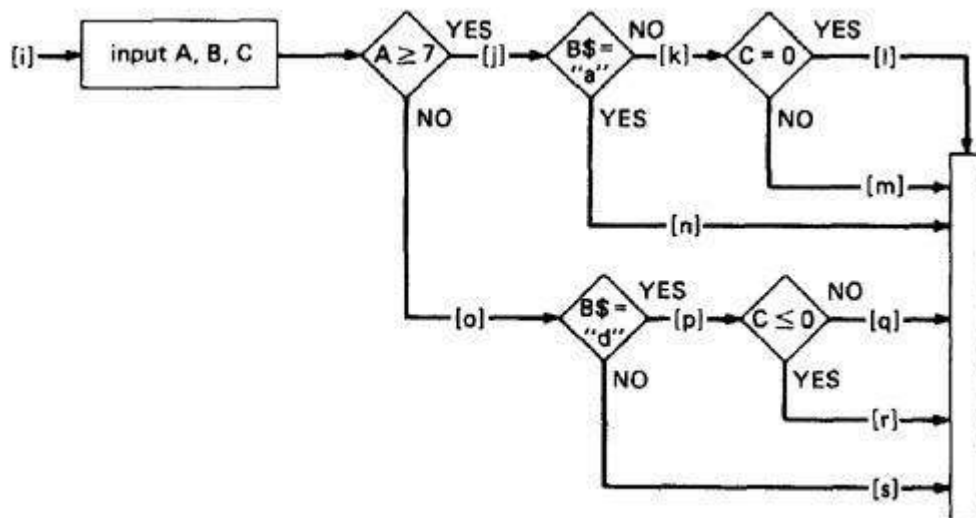


Figure 2.12: Single Link Marker Instrumentation

- **Why Single Link Markers aren't enough:** Unfortunately, a single link marker may not do the trick because links can be chewed by open bugs.

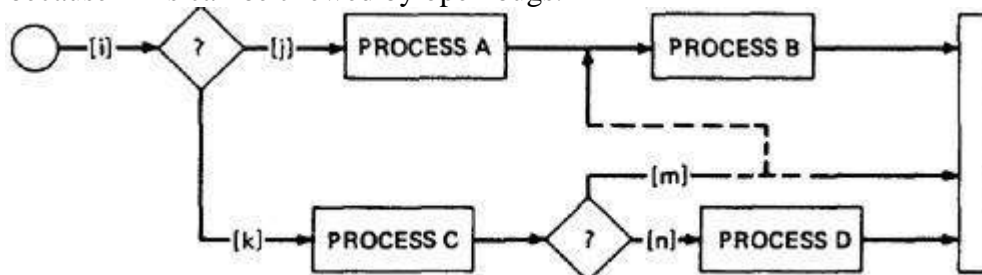


Figure 2.13: Why Single Link Markers aren't enough.

We intended to traverse the ikm path, but because of a rampaging GOTO in the middle of the m link, we go to process B. If coincidental correctness is against us, the outcomes will be the same and we won't know about the bug.

□ **Two Link Marker Method:**

The solution to the problem of single link marker method is to implement two markers per link: one at the beginning of each link and one at the end.

The two link markers now specify the path name and confirm both the beginning and end of the link.

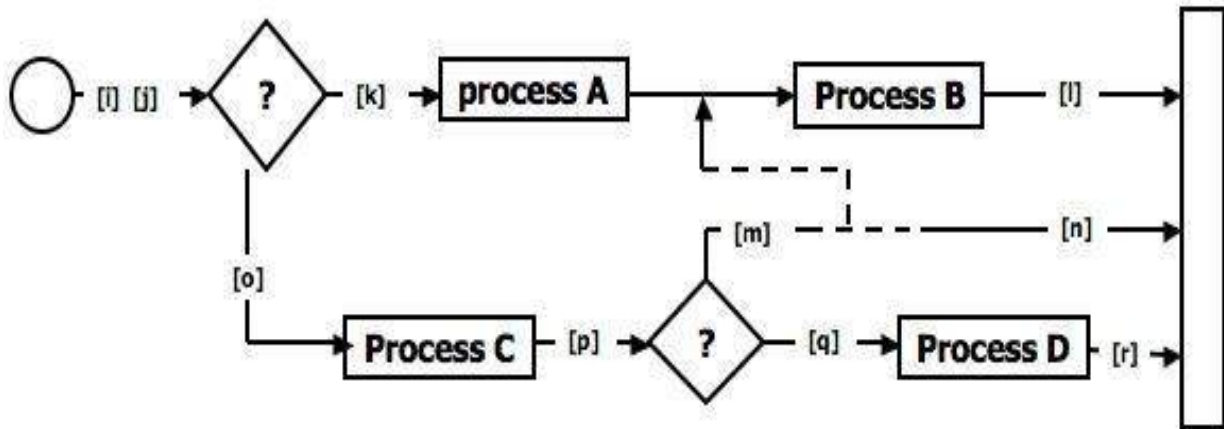


Figure 2.14: Double Link Marker Instrumentation

- **Link Counter:** A less disruptive (and less informative) instrumentation method is based on counters. Instead of a unique link name to be pushed into a string when the link is traversed, we simply increment a link counter. We now confirm that the path length is as expected. The same problem that led us to double link markers also leads us to double link counters.

TRANSACTION FLOW TESTING INTRODUCTION:

- A transaction is a unit of work seen from a system user's point of view.
- A transaction consists of a sequence of operations, some of which are performed by a system, persons or devices that are outside of the system.
- Transaction begins with Birth-that is they are created as a result of some external act.
- At the conclusion of the transaction's processing, the transaction is no longer in the system.
- **Example of a transaction:** A transaction for an online information retrieval system might consist of the following steps or tasks:
 - Accept input (tentative birth)
 - Validate input (birth)
 - Transmit acknowledgement to requester
 - Do input processing
 - Search file
 - Request directions from user
 - Accept input
 - Validate input

- Process request
- Update file
- Transmit output
- Record transaction in log and clean up (death)

• TRANSACTION FLOW GRAPHS:

- Transaction flows are introduced as a representation of a system's processing.
- The methods that were applied to control flow graphs are then used for functional testing.
- Transaction flows and transaction flow testing are to the independent system tester what control flows are path testing is to the programmer.
- The transaction flow graph is to create a behavioral model of the program that leads to functional testing.
- The transaction flow graph is a model of the structure of the system's behavior (functionality).
- An example of a Transaction Flow is as follows:

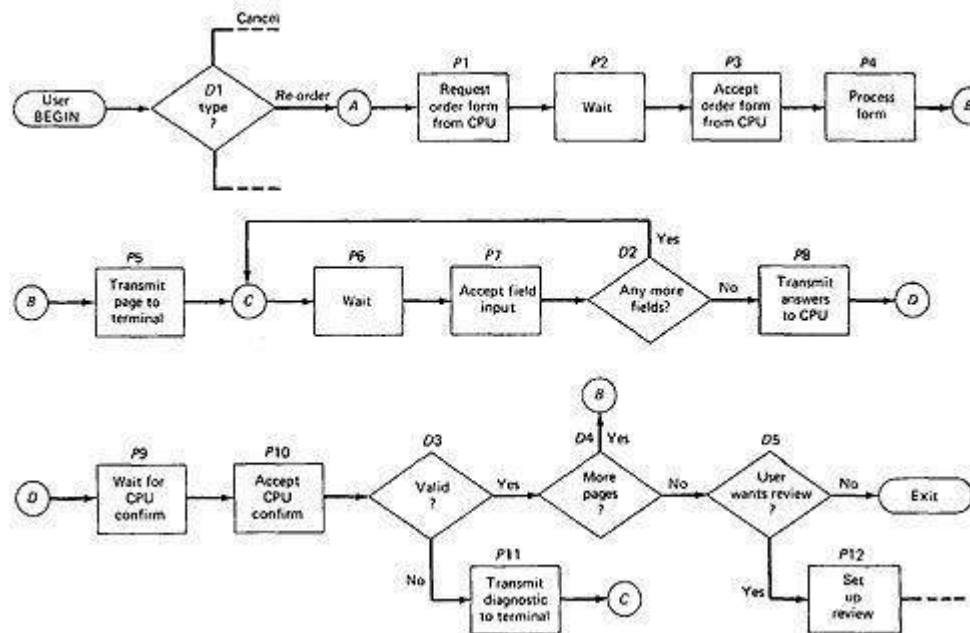


Figure 3.1: An Example of a Transaction Flow

• USAGE:

- Transaction flows are indispensable for specifying requirements of complicated systems, especially online systems.
- A big system such as an air traffic control or airline reservation system, has not hundreds, but thousands of different transaction flows.
- The flows are represented by relatively simple flowgraphs, many of which have a single straight-through path.
- Loops are infrequent compared to control flowgraphs.
- The most common loop is used to request a retry after user input errors. An ATM system, for example, allows the user to try, say three times, and will take the card away

the fourth time.

- **COMPLICATIONS:**

- In simple cases, the transactions have a unique identity from the time they're created to the time they're completed.
- In many systems the transactions can give birth to others, and transactions can also merge.
- **Births:** There are three different possible interpretations of the decision symbol, or nodes with two or more out links. It can be a Decision, Biosis or Mitosis.
 1. **Decision:** Here the transaction will take one alternative or the other alternative but not both. (See Figure 3.2 (a))
 2. **Biosis:** Here the incoming transaction gives birth to a new transaction, and both transaction continue on their separate paths, and the parent retains its identity. (See Figure 3.2 (b))
 3. **Mitosis:** Here the parent transaction is destroyed and two new transactions are created. (See Figure 3.2 (c))

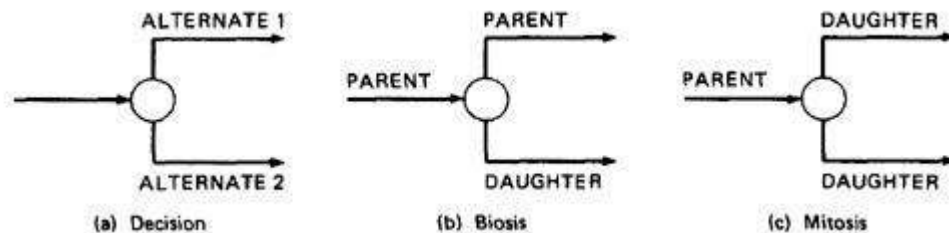


Figure 3.2: Nodes with multiple outlinks

Mergers: Transaction flow junction points are potentially as troublesome as transaction flow splits. There are three types of junctions: (1) Ordinary Junction (2) Absorption (3) Conjugation

1. **Ordinary Junction:** An ordinary junction which is similar to the junction in a control flow graph. A transaction can arrive either on one link or the other. (See Figure 3.3 (a))
2. **Absorption:** In absorption case, the predator transaction absorbs prey transaction. The prey gone but the predator retains its identity. (See Figure 3.3 (b))
3. **Conjugation:** In conjugation case, the two parent transactions merge to form a new daughter. In keeping with the biological flavor this case is called as conjugation. (See Figure 3.3 (c))

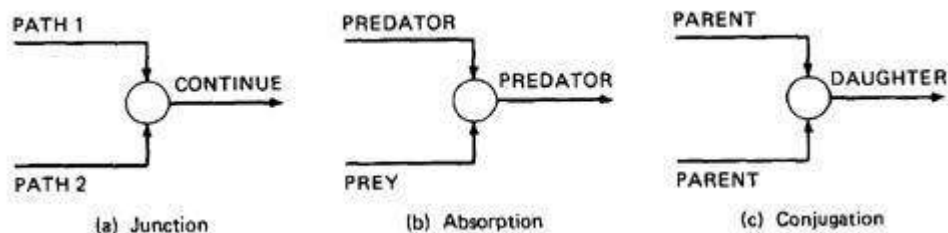


Figure 3.3: Transaction Flow Junctions and Mergers

We have no problem with ordinary decisions and junctions. Births, absorptions, and conjugations are as problematic for the software designer as they are for the software modeler and the test designer; as a consequence, such points have more than their share of bugs. The common problems are: lost daughters, wrongful deaths, and illegitimate births.

TRANSACTION FLOW TESTING TECHNIQUES:

- **GET THE TRANSACTIONS FLOWS:**
 - Complicated systems that process a lot of different, complicated transactions should have explicit representations of the transactions flows, or the equivalent.
 - Transaction flows are like control flow graphs, and consequently we should expect to have them in increasing levels of detail.
 - The system's design documentation should contain an overview section that details the main transaction flows.
 - Detailed transaction flows are a mandatory pre requisite to the rational design of a system's functional test.

- **INSPECTIONS, REVIEWS AND WALKTHROUGHS:**
 - Transaction flows are natural agenda for system reviews or inspections.
 - In conducting the walkthroughs, you should:
 - Discuss enough transaction types to account for 98%-99% of the transaction the system is expected to process.
 - Discuss paths through flows in functional rather than technical terms.
 - Ask the designers to relate every flow to the specification and to show how that transaction, directly or indirectly, follows from the requirements.
 - Make transaction flow testing the corner stone of system functional testing just as path testing is the corner stone of unit testing.
 - Select additional flow paths for loops, extreme values, and domain boundaries.
 - Design more test cases to validate all births and deaths.
 - Publish and distribute the selected test paths through the transaction flows as early as possible so that they will exert the maximum beneficial effect on the project.

- **PATH SELECTION:**
 - Select a set of covering paths (c1+c2) using the analogous criteria you used for structural path testing.
 - Select a covering set of paths based on functionally sensible transactions as you would for control flow graphs.
 - Try to find the most tortuous, longest, strangest path from the entry to the exit of the transaction flow.

- **PATH SENSITIZATION:**
 - Most of the normal paths are very easy to sensitize-80% - 95% transaction flow coverage (c1+c2) is usually easy to achieve.
 - The remaining small percentage is often very difficult.
 - Sensitization is the act of defining the transaction. If there are sensitization problems on the easy paths, then bet on either a bug in transaction flows or a design bug.

- **PATH INSTRUMENTATION:**
 - Instrumentation plays a bigger role in transaction flow testing than in unit path testing.
 - The information of the path taken for a given transaction must be kept with that transaction and can be recorded by a central transaction dispatcher or by the individual processing modules.
 - In some systems, such traces are provided by the operating systems or a running log.

UNIT III

DATA FLOW TESTING

BASICS OF DATA FLOW TESTING:

□ DATA FLOW TESTING:

- Data flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects.
- For example, pick enough paths to assure that every data object has been initialized prior to use or that all defined objects have been used for something.
- **Motivation:** It is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, one should not feel confident about a program without having seen the effect of using the value produced by each and every computation.

□ DATA FLOW MACHINES:

- There are two types of data flow machines with different architectures. (1) Von Neumann machines (2) Multi-instruction, multi-data machines (MIMD).
- **Von Neumann Machine Architecture:**
 - Most computers today are von-neumann machines.
 - This architecture features interchangeable storage of instructions and data in the same memory units.
 - The Von Neumann machine Architecture executes one instruction at a time in the following, micro instruction sequence:
 - Fetch instruction from memory
 - Interpret instruction
 - Fetch operands
 - Process or Execute
 - Store result
 - Increment program counter
 - GOTO 1
- **Multi-instruction, Multi-data machines (MIMD) Architecture:**
 - These machines can fetch several instructions and objects in parallel.
 - They can also do arithmetic and logical operations simultaneously on different data objects.
 - The decision of how to sequence them depends on the compiler.

□ BUG ASSUMPTION:

The bug assumption for data-flow testing strategies is that control flow is generally correct and that something has gone wrong with the software so that data objects are not available when they should be, or silly things are being done to data objects.

- Also, if there is a control-flow problem, we expect it to have symptoms that can be detected by data-flow analysis.
- Although we'll be doing data-flow testing, we won't be using data flow graphs as such.

Rather, we'll use an ordinary control flow graph annotated to show what happens to the data objects of interest at the moment.

□ DATA FLOW GRAPHS:

- The data flow graph is a graph consisting of nodes and directed links.
- We will use a control graph to show what happens to data objects of interest at that moment.
- Our objective is to expose deviations between the data flows we have and the data flows we want.

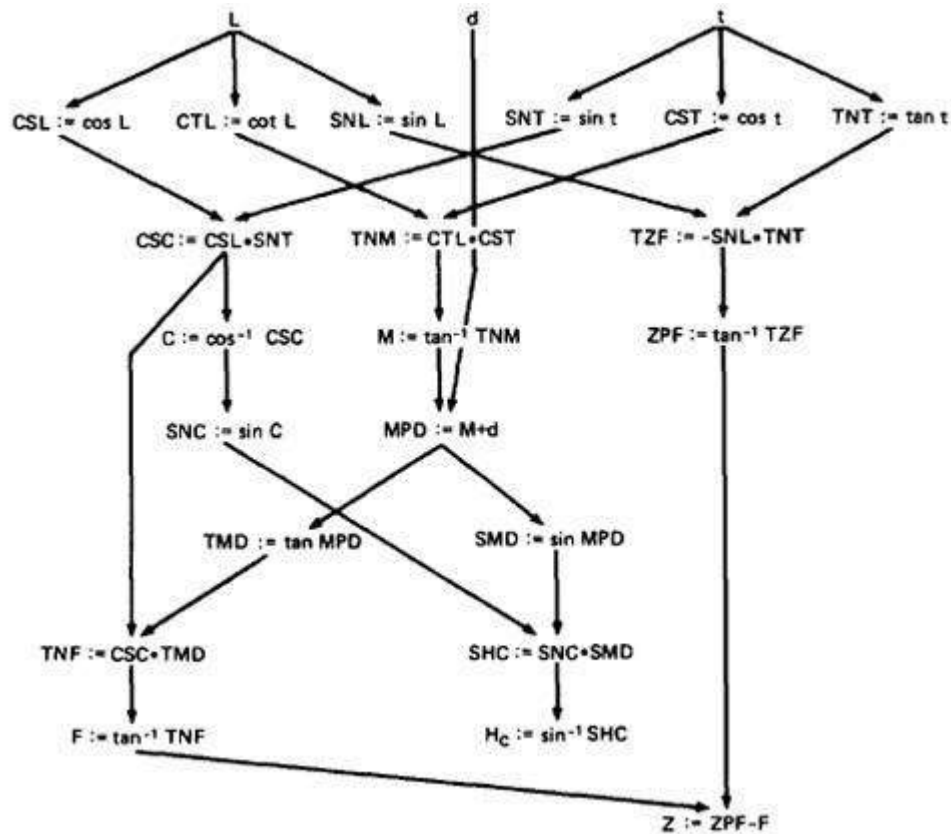


Figure 3.4: Example of a data flow graph

○ Data Object State and Usage:

- Data Objects can be created, killed and used.
- They can be used in two distinct ways: (1) In a Calculation (2) As a part of a Control Flow Predicate.
- The following symbols denote these possibilities:
 1. **Defined:** d - defined, created, initialized etc
 2. **Killed or undefined:** k - killed, undefined, released etc
 3. **Usage:** u - used for something (c - used in Calculations, p - used in a predicate)

1. Defined (d):

- An object is defined explicitly when it appears in a data declaration.

- Or implicitly when it appears on the left hand side of the assignment.
- It is also to be used to mean that a file has been opened.
- A dynamically allocated object has been allocated.
- Something is pushed on to the stack.
- A record written.

2.Killed or Undefined (k):

- An object is killed on undefined when it is released or otherwise made unavailable.
- When its contents are no longer known with certitude (with absolute certainty / perfectness).
- Release of dynamically allocated objects back to the availability pool.
- Return of records.
- The old top of the stack after it is popped.
- An assignment statement can kill and redefine immediately. For example, if A had been previously defined and we do a new assignment such as $A := 17$, we have killed A's previous value and redefined A

3.Usage (u):

- A variable is used for computation (c) when it appears on the right hand side of an assignment statement.
- A file record is read or written.
- It is used in a Predicate (p) when it appears directly in a predicate.

DATA FLOW ANOMALIES:

An anomaly is denoted by a two-character sequence of actions. For example, ku means that the object is killed and then used, where as dd means that the object is defined twice without an intervening usage.

What an anomaly is depend on the application.

There are nine possible two-letter combinations for d, k and u. some are bugs, some are suspicious, and some are okay.

- 1 **dd** :- probably harmless but suspicious. Why define the object twice without an intervening usage?
- 2 **dk** :- probably a bug. Why define the object without using it?
- 3 **du** :- the normal case. The object is defined and then used.
- 4 **kd** :- normal situation. An object is killed and then redefined.
- 5 **kk** :- harmless but probably buggy. Did you want to be sure it was really killed?
- 6 **ku** :- a bug. the object doesnot exist.
- 7 **ud** :- usually not a bug because the language permits reassignment at almost any time.
- 8 **uk** :- normal situation.
- 9 **uu** :- normal situation.

In addition to the two letter situations, there are six single letter situations. We will use a leading dash to mean that nothing of interest (d,k,u) occurs prior to the action noted along the entry-exit path of interest.

A trailing dash to mean that nothing happens after the point of interest to the exit.

They possible anomalies are:

- 1 **-k** :- possibly anomalous because from the entrance to this point on the path, the variable had not been defined. We are killing a variable that does not exist.

- 2 **-d** :- okay. This is just the first definition along this path.
- 3 **-u** :- possibly anomalous. Not anomalous if the variable is global and has been previously defined.
- 4 **k** :- not anomalous. The last thing done on this path was to kill the variable.
- 5 **d** :- possibly anomalous. The variable was defined and not used on this path. But this could be a global definition.
- 6 **u** :- not anomalous. The variable was used but not killed on this path. Although this sequence is not anomalous, it signals a frequent kind of bug. If d and k mean dynamic storage allocation and return respectively, this could be an instance in which a dynamically allocated object was not returned to the pool after use.

DATA FLOW ANOMALY STATE GRAPH:

Data flow anomaly model prescribes that an object can be in one of four distinct states:

0. **K** :- undefined, previously killed, doesnot exist
1. **D** :- defined but not yet used for anything
2. **U** :- has been used for computation or in predicate
3. **A** :- anomalous

These capital letters (K, D, U, A) denote the state of the variable and should not be confused with the program action, denoted by lower case letters.

Unforgiving Data - Flow Anomaly Flow Graph: Unforgiving model, in which once a variable becomes anomalous it can never return to a state of grace.

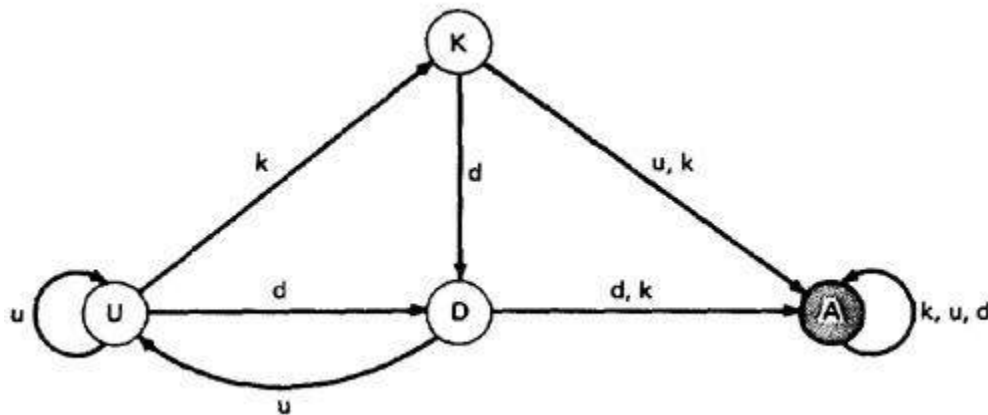


Figure 3.5: Unforgiving Data Flow Anomaly State Graph

Assume that the variable starts in the K state - that is, it has not been defined or does not exist. If an attempt is made to use it or to kill it (e.g., say that we're talking about opening, closing, and using files and that 'killing' means closing), the object's state becomes anomalous (state A) and, once it is anomalous, no action can return the variable to a working state.

If it is defined (d), it goes into the D, or defined but not yet used, state. If it has been defined (D) and redefined (d) or killed without use (k), it becomes anomalous, while usage (u) brings it to the U state. If in U, redefinition (d) brings it to D, u keeps it in U, and k kills it.

Forgiving Data - Flow Anomaly Flow Graph: Forgiving model is an alternate model where redemption (recover) from the anomalous state is possible

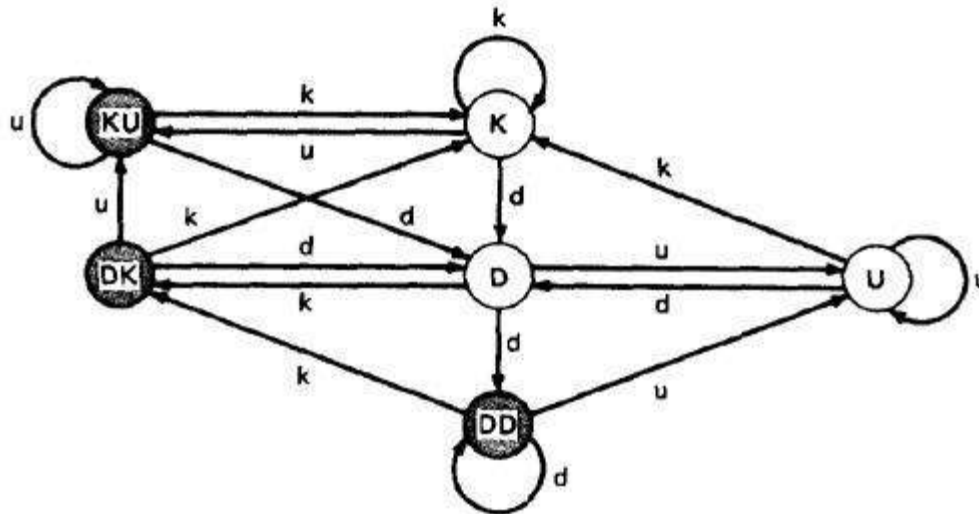


Figure 3.6: Forgiving Data Flow Anomaly State Graph

This graph has three normal and three anomalous states and he considers the kk sequence not to be anomalous. The difference between this state graph and Figure 3.5 is that redemption is possible. A proper action from any of the three anomalous states returns the variable to a useful working state.

The point of showing you this alternative anomaly state graph is to demonstrate that the specifics of an anomaly depends on such things as language, application, context, or even your frame of mind. In principle, you must create a new definition of data flow anomaly (e.g., a new state graph) in each situation. You must at least verify that the anomaly definition behind the theory or imbedded in a data flow anomaly test tool is appropriate to your situation.

STATIC Vs DYNAMIC ANOMALY DETECTION:

Static analysis is analysis done on source code without actually executing it. For example: source code syntax error detection is the static analysis result.

Dynamic analysis is done on the fly as the program is being executed and is based on intermediate values that result from the program's execution. For example: a division by zero warning is the dynamic result.

If a problem, such as a data flow anomaly, can be detected by static analysis methods, then it doesn't belongs in testing - it belongs in the language processor.

There is actually a lot more static analysis for data flow analysis for data flow anomalies going on in current language processors.

For example, language processors which force variable declarations can detect (-u) and (ku) anomalies. But still there are many things for which current notions of static analysis are INADEQUATE.

Why Static Analysis isn't enough? There are many things for which current notions of static analysis are inadequate. They are:

- **Dead Variables:** Although it is often possible to prove that a variable is dead or alive at a given point in the program, the general problem is unsolvable.
- **Arrays:** Arrays are problematic in that the array is defined or killed as a single object, but reference is to specific locations within the array. Array pointers are usually dynamically calculated, so there's no way to do a static analysis to validate the pointer value. In many languages, dynamically allocated arrays contain garbage unless explicitly initialized and therefore, -u anomalies are possible.
- **Records and Pointers:** The array problem and the difficulty with pointers is a special case of multipart data structures. We have the same problem with records and the pointers to them. Also, in many applications we create files and their names dynamically and there's no way to determine, without execution, whether such objects are in the proper state on a given path or, for that matter, whether they exist at all.
- **Dynamic Subroutine and Function Names in a Call:** subroutine or function name is a dynamic variable in a call. What is passed, or a combination of subroutine names and data objects, is constructed on a specific path. There's no way, without executing the path, to determine whether the call is correct or not.
- **False Anomalies:** Anomalies are specific to paths. Even a "clear bug" such as ku may not be a bug if the path along which the anomaly exists is unachievable. Such "anomalies" are false anomalies. Unfortunately, the problem of determining whether a path is or is not achievable is unsolvable.
- **Recoverable Anomalies and Alternate State Graphs:** What constitutes an anomaly depends on context, application, and semantics. How does the compiler know which model I have in mind? It can't because the definition of "anomaly" is not fundamental. The language processor must have a built-in anomaly definition with which you may or may not (with good reason) agree.
- **Concurrency, Interrupts, System Issues:** As soon as we get away from the simple single- task uniprocessor environment and start thinking in terms of systems, most anomaly issues become vastly more complicated.

How often do we define or create data objects at an interrupt level so that they can be processed by a lower-priority routine? Interrupts can make the "correct" anomalous and the "anomalous" correct. True concurrency (as in an MIMD machine) and pseudo concurrency (as in multiprocessing) systems can do the same to us. Much of integration and system testing is aimed at detecting data-flow anomalies that cannot be detected in the context of a single routine.

Although static analysis methods have limits, they are worth using and a continuing trend in language processor design has been better static analysis methods, especially for data flow anomaly detection. That's good because it means there's less for us to do as testers and we have far too much to do as it is.

DATA FLOW MODEL:

The data flow model is based on the program's control flow graph - Don't confuse that with the program's data flow graph.

Here we annotate each link with symbols (for example, d, k, u, c, and p) or sequences of symbols (for example, dd, du, ddd) that denote the sequence of data operations on that link with respect to the variable of interest. Such annotations are called link weights.

The control flow graph structure is same for every variable: it is the weights that change.

Components of the model:

1. To every statement there is a node, whose name is unique. Every node has at least one outlink and at least one inlink except for exit nodes and entry nodes.
2. Exit nodes are dummy nodes placed at the outgoing arrowheads of exit statements (e.g., END, RETURN), to complete the graph. Similarly, entry nodes are dummy nodes placed at entry statements (e.g., BEGIN) for the same reason.
3. The outlink of simple statements (statements with only one outlink) are weighted by the proper sequence of data-flow actions for that statement. Note that the sequence can consist of more than one letter. For example, the assignment statement $A := A + B$ in most languages is weighted by cd or possibly ckd for variable A. Languages that permit multiple simultaneous assignments and/or compound statements can have anomalies within the statement. The sequence must correspond to the order in which the object code will be executed for that variable.
4. Predicate nodes (e.g., IF-THEN-ELSE, DO WHILE, CASE) are weighted with the p - use(s) on every outlink, appropriate to that outlink.
5. Every sequence of simple statements (e.g., a sequence of nodes with one inlink and one outlink) can be replaced by a pair of nodes that has, as weights on the link between them, the concatenation of link weights.
6. If there are several data-flow actions on a given link for a given variable, then the weight of the link is denoted by the sequence of actions on that link for that variable.
7. Conversely, a link with several data-flow actions on it can be replaced by a succession of equivalent links, each of which has at most one data-flow action for any variable.

Let us consider the example:

```
CODE* (PDL)

INPUT X, Y
Z := X + Y
V := X - Y
IF Z >= 0 GOTO SAM
JOE: Z := Z - 1
SAM: Z := Z + V
FOR U = 0 TO Z
  V(U), U(V) := (Z + V) * U
  IF V(U) = 0 GOTO JOE
  Z := Z - 1
  IF Z = 0 GOTO ELL
  U := U + 1
NEXT U

V(U-1) := V(U+1) + U(V-1)
ELL: V(U+U(V)) := U + V
IF U = V GOTO JOE
IF U > V THEN U := Z
Z := U
END
```

* A contrived horror

Figure 3.7: Program Example (PDL)

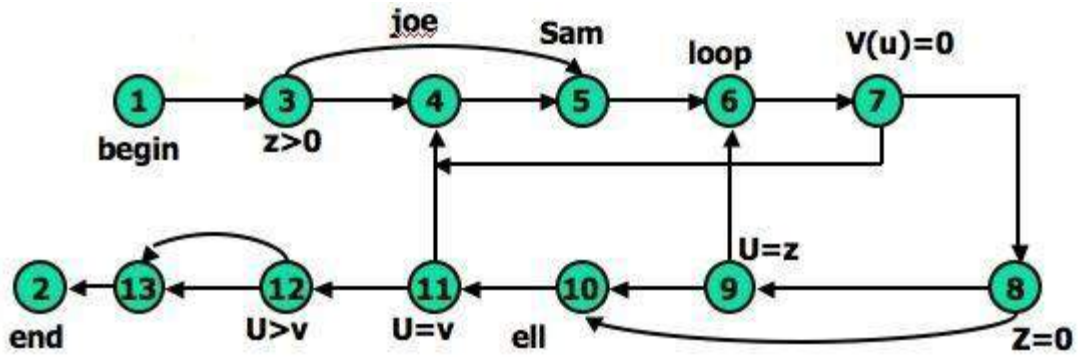


Figure 3.8: Unannotated flow graph for example program in Figure 3.7

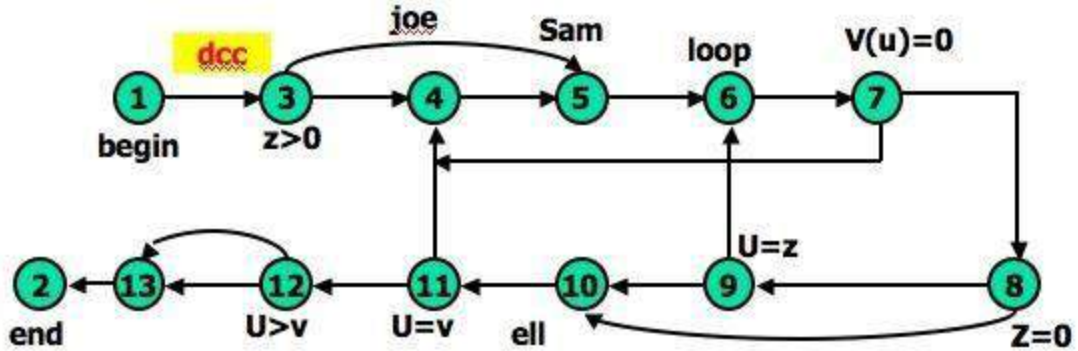


Figure 3.9: Control flow graph annotated for X and Y data flows.

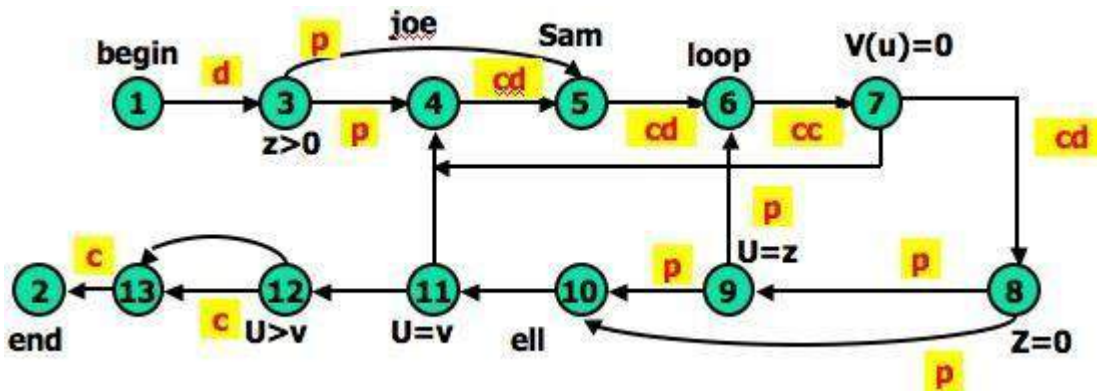


Figure 3.10: Control flow graph annotated for Z data flow.

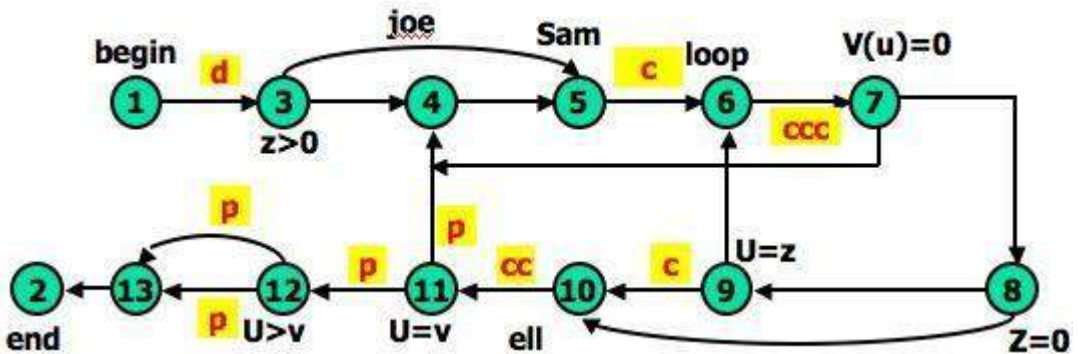


Figure 3.11: Control flow graph annotated for V data flow.

STRATEGIES OF DATA FLOW TESTING:

- **INTRODUCTION:**

- Data Flow Testing Strategies are structural strategies.
- In contrast to the path-testing strategies, data-flow strategies take into account what happens to data objects on the links in addition to the raw connectivity of the graph.
- In other words, data flow strategies require data-flow link weights (d,k,u,c,p).
- Data Flow Testing Strategies are based on selecting test path segments (also called **sub paths**) that satisfy some characteristic of data flows for all data objects.
- For example, all sub paths that contain a d (or u, k, du, dk).
- A strategy X is **stronger** than another strategy Y if all test cases produced under Y are included in those produced under X - conversely for **weaker**.

- **TERMINOLOGY:**

1. **Definition-Clear Path Segment**, with respect to variable X, is a connected sequence of links such that X is (possibly) defined on the first link and not redefined or killed on any subsequent link of that path segment. 11 paths in Figure 3.9 are definition clear because variables X and Y are defined only on the first link (1,3) and not thereafter. In Figure 3.10, we have a more complicated situation. The following path segments are definition-clear: (1,3,4), (1,3,5), (5,6,7,4), (7,8,9,6,7), (7,8,9,10), (7,8,10), (7,8,10,11). Subpath (1,3,4,5) is not definition-clear because the variable is defined on (1,3) and again on (4,5). For practice, try finding all the definition-clear subpaths for this routine (i.e., for all variables).
2. **Loop-Free Path Segment** is a path segment for which every node in it is visited at most once. For Example, path (4,5,6,7,8,10) in Figure 3.10 is loop free, but path (10,11,4,5,6,7,8,10,11,12) is not because nodes 10 and 11 are each visited twice.
3. **Simple path segment** is a path segment in which at most one node is visited twice. For example, in Figure 3.10, (7,4,5,6,7) is a simple path segment. A simple path segment is either loop-free or if there is a loop, only one node is involved.
4. A **du path** from node i to k is a path segment such that if the last link has a computational use of X, then the path is simple and definition-clear; if the penultimate (last but one) node is j - that is, the path is (i,p,q,...,r,s,t,j,k) and link (j,k) has a predicate use - then the path from i to j is both loop-free and definition-clear.

STRATEGIES: The structural test strategies discussed below are based on the program's control flow graph. They differ in the extent to which predicate uses and/or computational uses of variables are included in the test set. Various types of data flow testing strategies in decreasing order of their effectiveness are:

All - du Paths (ADUP): The all-du-paths (ADUP) strategy is the strongest data-flow testing strategy discussed here. It requires that every du path from every definition of every variable to every some test.

For variable X and Y: In Figure 3.9, because variables X and Y are used only on link (1,3), any test that starts at the entry satisfies this criterion (for variables X and Y, but not for all variables as required by the strategy).

For variable Z: The situation for variable Z (Figure 3.10) is more complicated because the variable is redefined in many places. For the definition on link (1,3) we must exercise paths that include subpaths (1,3,4) and (1,3,5). The definition on link (4,5) is covered by any path that includes (5,6), such as subpath (1,3,4,5,6, ...). The (5,6) definition requires paths that include subpaths (5,6,7,4) and (5,6,7,8).

For variable V: Variable V (Figure 3.11) is defined only once on link (1,3). Because V has a predicate use at node 12 and the subsequent path to the end must be forced for both directions at node 12, the all-du-paths strategy for this variable requires that we exercise all loop-free entry/exit paths and at least one path that includes the loop caused by (11,4).

Note that we must test paths that include both subpaths (3,4,5) and (3,5) even though neither of these has V definitions. They must be included because they provide alternate du paths to the V use on link (5,6). Although (7,4) is not used in the test set for variable V, it will be included in the test set that covers the predicate uses of array variable V() and U.

The all-du-paths strategy is a strong criterion, but it does not take as many tests as it might seem at first because any one test simultaneously satisfies the criterion for several definitions and uses of several different variables.

All Uses Strategy (AU): The all uses strategy is that at least one definition clear path from every definition of every variable to every use of that definition be exercised under some test.

Just as we reduced our ambitions by stepping down from all paths (P) to branch coverage (C2), say, we can reduce the number of test cases by asking that the test set should include at least one path segment from every definition to every use that can be reached by that definition.

For variable V: In Figure 3.11, ADUP requires that we include subpaths (3,4,5) and (3,5) in some test because subsequent uses of V, such as on link (5,6), can be reached by either alternative. In AU either (3,4,5) or (3,5) can be used to start paths, but we don't have to use both. Similarly, we can skip the (8,10) link if we've included the (8,9,10) subpath.

Note the hole. We must include (8,9,10) in some test cases because that's the only way to reach the c use at link (9,10) - but suppose our bug for variable V is on link (8,10) after all? Find a covering set of paths under AU for Figure 3.11.

All p-uses/some c-uses strategy (APU+C) : For every variable and every definition of that variable, include at least one definition free path from the definition to every predicate use; if there are definitions of the variables that are not covered by the above prescription, then add computational use test cases as required to cover every definition.

For variable Z: In Figure 3.10, for APU+C we can select paths that all take the upper link (12,13) and therefore we do not cover the c-use of Z: but that's okay according to the strategy's definition because every definition is covered.

Links (1,3), (4,5), (5,6), and (7,8) must be included because they contain definitions for variable Z. Links (3,4), (3,5), (8,9), (8,10), (9,6), and (9,10) must be included because they contain predicate uses of Z. Find a covering set of test cases under APU+C for all variables in this example - it only takes two tests.

For variable V: In Figure 3.11, APU+C is achieved for V by (1,3,5,6,7,8,10,11,4,5,6,7,8,10,11,12[upper], 13,2) and (1,3,5,6,7,8,10,11,12[lower], 13,2). Note that the c-use at (9,10) need not be included under the APU+C criterion.

All c-uses/some p-uses strategy (ACU+P) : The all c-uses/some p-uses strategy (ACU+P) is to first ensure coverage by computational use cases and if any definition is not covered by the previously selected paths, add such predicate use cases as are needed to assure that every definition is included in some test.

For variable Z: In Figure 3.10, ACU+P coverage is achieved for Z by path (1,3,4,5,6,7,8,10, 11,12,13[lower], 2), but the predicate uses of several definitions are not covered. Specifically, the (1,3) definition is not covered for the (3,5) p-use, the (7,8) definition is not covered for the (8,9), (9,6) and (9, 10) p-uses.

The above examples imply that APU+C is stronger than branch coverage but ACU+P may be weaker than, or incomparable to, branch coverage.

All Definitions Strategy (AD) : The all definitions strategy asks only every definition of every variable be covered by at least one use of that variable, be that use a computational use or a predicate use.

For variable Z: Path (1,3,4,5,6,7,8, . . .) satisfies this criterion for variable Z, whereas any entry/exit path satisfies it for variable V.

From the definition of this strategy we would expect it to be weaker than both ACU+P and APU+C.

1. All Predicate Uses (APU), All Computational Uses (ACU) Strategies : The all predicate uses strategy is derived from APU+C strategy by dropping the requirement that we include a c- use for the variable if there are no p-uses for the variable. The all computational uses strategy is derived from ACU+P strategy by dropping the requirement that we include a p-use for the variable if there are no c-uses for the variable.

It is intuitively obvious that ACU should be weaker than ACU+P and that APU should be weaker than APU+C.

ORDERING THE STRATEGIES:

Figure 3.12 compares path-flow and data-flow testing strategies. The arrows denote that the strategy at the arrow's tail is stronger than the strategy at the arrow's head

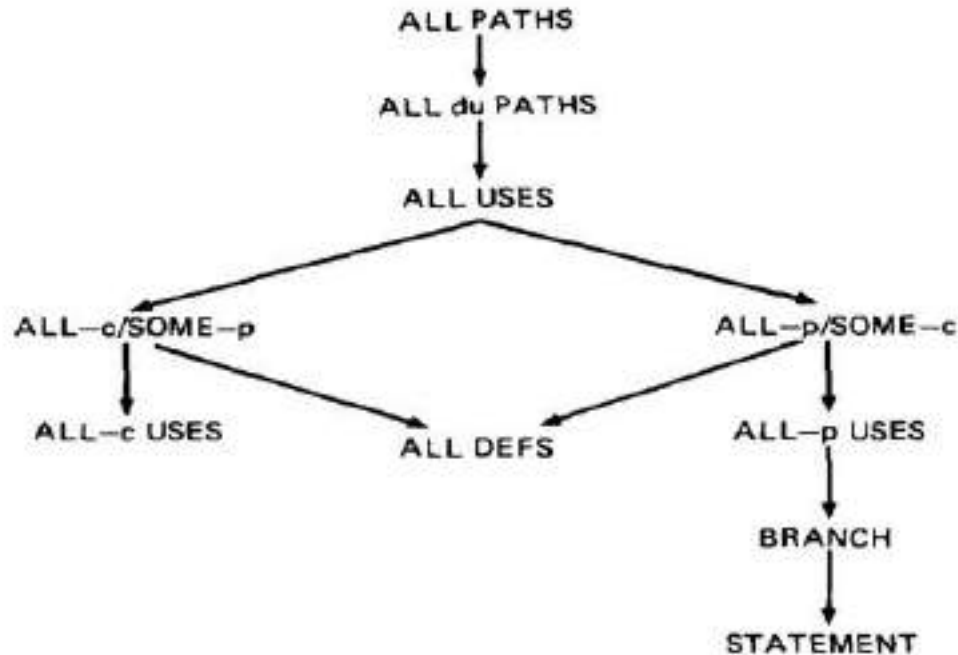


Figure 3.12: Relative Strength of Structural Test Strategies.

- The right-hand side of this graph, along the path from "all paths" to "all statements" is the more interesting hierarchy for practical applications.
- Note that although ACU+P is stronger than ACU, both are incomparable to the predicate-biased strategies. Note also that "all definitions" is not comparable to ACU or APU.

SLICING AND DICING:

- A (static) program **slice** is a part of a program (e.g., a selected set of statements) defined with respect to a given variable X (where X is a simple variable or a data vector) and a statement i : it is the set of all statements that could (potentially, under static analysis) affect the value of X at statement i - where the influence of a faulty statement could result from an improper computational use or predicate use of some other variables at prior statements.
- If X is incorrect at statement i , it follows that the bug must be in the program slice for X with respect to i
- A program **dice** is a part of a slice in which all statements which are known to be correct have been removed.
- In other words, a dice is obtained from a slice by incorporating information obtained through testing or experiment (e.g., debugging).

- The debugger first limits her scope to those prior statements that could have caused the faulty value at statement *i* (the slice) and then eliminates from further consideration those statements that testing has shown to be correct.
- Debugging can be modeled as an iterative procedure in which slices are further refined by dicing, where the dicing information is obtained from ad hoc tests aimed primarily at eliminating possibilities. Debugging ends when the dice has been reduced to the one faulty statement.
- **Dynamic slicing** is a refinement of static slicing in which only statements on achievable paths to the statement in question are included.

DOMAIN TESTING

DOMAINS AND PATHS:

- **INTRODUCTION:**

- **Domain:** In mathematics, domain is a set of possible values of an independent variable or the variables of a function.
- Programs as input data classifiers: domain testing attempts to determine whether the classification is or is not correct.
- Domain testing can be based on specifications or equivalent implementation information.
- If domain testing is based on specifications, it is a functional test technique.
- If domain testing is based implementation details, it is a structural test technique.
- For example, you're doing domain testing when you check extreme values of an input variable.

All inputs to a program can be considered as if they are numbers. For example, a character string can be treated as a number by concatenating bits and looking at them as if they were a binary integer. This is the view in domain testing, which is why this strategy has a mathematical flavor.

- **THE MODEL:** The following figure is a schematic representation of domain testing.

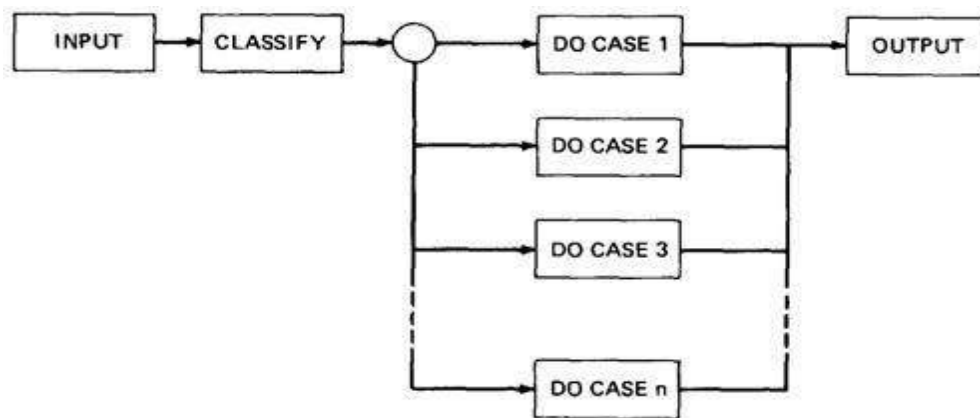


Figure 4.1: Schematic Representation of Domain Testing.

- Before doing whatever it does, a routine must classify the input and set it moving on the right path.

- An invalid input (e.g., value too big) is just a special processing case called 'reject'.
- The input then passes to a hypothetical subroutine rather than on calculations.
- In domain testing, we focus on the classification aspect of the routine rather than on the calculations.
- Structural knowledge is not needed for this model - only a consistent, complete specification of input values for each case.
- We can infer that for each case there must be at least one path to process that case.
- **A DOMAIN IS A SET:**
 - An input domain is a set.
 - If the source language supports set definitions (E.g. PASCAL set types and C enumerated types) less testing is needed because the compiler does much of it for us.
 - Domain testing does not work well with arbitrary discrete sets of data objects.
 - Domain for a loop-free program corresponds to a set of numbers defined over the input vector.
- **DOMAINS, PATHS AND PREDICATES:**
 - In domain testing, predicates are assumed to be interpreted in terms of input vector variables.
 - If domain testing is applied to structure, then predicate interpretation must be based on actual paths through the routine - that is, based on the implementation control flow graph.
 - Conversely, if domain testing is applied to specifications, interpretation is based on a specified data flow graph for the routine; but usually, as is the nature of specifications, no interpretation is needed because the domains are specified directly.
 - For every domain, there is at least one path through the routine.
 - There may be more than one path if the domain consists of disconnected parts or if the domain is defined by the union of two or more domains.
 - Domains are defined their boundaries. Domain boundaries are also where most domain bugs occur.
 - For every boundary there is at least one predicate that specifies what numbers belong to the domain and what numbers don't.
For example, in the statement IF $x > 0$ THEN ALPHA ELSE BETA we know that numbers greater than zero belong to ALPHA processing domain(s) while zero and smaller numbers belong to BETA domain(s).
 - A domain may have one or more boundaries - no matter how many variables define it. For example, if the predicate is $x^2 + y^2 < 16$, the domain is the inside of a circle of radius 4 about the origin. Similarly, we could define a spherical domain with one boundary but in three variables.
 - Domains are usually defined by many boundary segments and therefore by many predicates. i.e. the set of interpreted predicates traversed on that path (i.e., the path's predicate expression) defines the domain's boundaries.
- **A DOMAIN CLOSURE:**
 - A domain boundary is **closed** with respect to a domain if the points on the boundary belong to the domain.
 - If the boundary points belong to some other domain, the boundary is said to be **open**.

- Figure 4.2 shows three situations for a one-dimensional domain - i.e., a domain defined over one input variable; call it x

The importance of domain closure is that incorrect closure bugs are frequent domain bugs. For example, $x \geq 0$ when $x > 0$ was intended

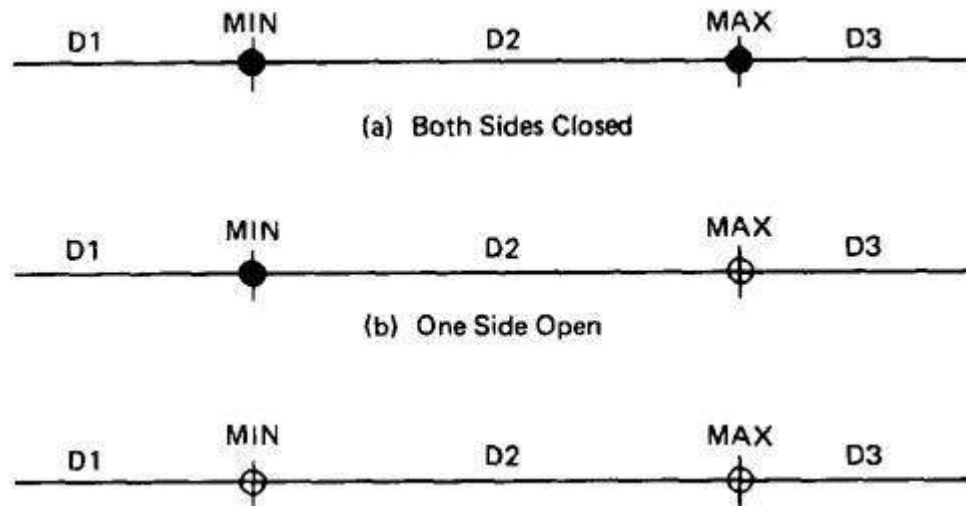


Figure 4.2: Open and Closed Domains.

- **DOMAIN DIMENSIONALITY:**

- Every input variable adds one dimension to the domain.
- One variable defines domains on a number line.
- Two variables define planar domains.
- Three variables define solid domains.
- Every new predicate slices through previously defined domains and cuts them in half.
- Every boundary slices through the input vector space with a dimensionality which is less than the dimensionality of the space.
- Thus, planes are cut by lines and points, volumes by planes, lines and points and n-spaces by hyperplanes.

- **BUG ASSUMPTION:**

- The bug assumption for the domain testing is that processing is okay but the domain definition is wrong.
- An incorrectly implemented domain means that boundaries are wrong, which may in turn mean that control flow predicates are wrong.
- Many different bugs can result in domain errors. Some of them are:

Domain Errors:

- **Double Zero Representation:** In computers or Languages that have a distinct positive and negative zero, boundary errors for negative zero are common.
- **Floating point zero check:** A floating point number can equal zero only if the previous definition of that number set it to zero or if it is subtracted from itself or multiplied by zero. So the floating points zero check to be done against an epsilon value.
- **Contradictory domains:** An implemented domain can never be ambiguous or contradictory, but a specified domain can. A contradictory

domain specification means that at least two supposedly distinct domains overlap.

- **Ambiguous domains:** Ambiguous domains mean that union of the domains is incomplete. That is there are missing domains or holes in the specified domains. Not specifying what happens to points on the domain boundary is a common ambiguity.
- **Over specified Domains:** his domain can be overloaded with so many conditions that the result is a null domain. Another way to put it is to say that the domain's path is unachievable.
- **Boundary Errors:** Errors caused in and around the boundary of a domain. Example, boundary closure bug, shifted, tilted, missing, extra boundary.
- **Closure Reversal:** A common bug. The predicate is defined in terms of \geq . The programmer chooses to implement the logical complement and incorrectly uses \leq for the new predicate; i.e., $x \geq 0$ is incorrectly negated as $x \leq 0$, thereby shifting boundary values to adjacent domains.
- **Faulty Logic:** Compound predicates (especially) are subject to faulty logic transformations and improper simplification. If the predicates define domain boundaries, all kinds of domain bugs can result from faulty logic manipulations.
- **RESTRICTIONS TO DOMAIN TESTING:** Domain testing has restrictions, as do other testing techniques. Some of them include:
 - **Co-incidental Correctness:** Domain testing isn't good at finding bugs for which the outcome is correct for the wrong reasons. If we're plagued by coincidental correctness we may misjudge an incorrect boundary. Note that this implies weakness for domain testing when dealing with routines that have binary outcomes (i.e., TRUE/FALSE)
 - **Representative Outcome:** Domain testing is an example of **partition testing**. Partition-testing strategies divide the program's input space into domains such that all inputs within a domain are equivalent (not equal, but equivalent) in the sense that any input represents all inputs in that domain.
 - If the selected input is shown to be correct by a test, then processing is presumed correct, and therefore all inputs within that domain are expected (perhaps unjustifiably) to be correct. Most test techniques, functional or structural, fall under partition testing and therefore make this representative outcome assumption. For example, x^2 and 2^x are equal for $x = 2$, but the functions are different. The functional differences between adjacent domains are usually simple, such as $x + 7$ versus $x + 9$, rather than x^2 versus 2^x .

Simple Domain Boundaries and Compound Predicates: Compound predicates in which each part of the predicate specifies a different boundary are not a problem: for example, $x \geq 0$ AND $x < 17$, just specifies two domain boundaries by one compound predicate. As

an example of a compound predicate that specifies one boundary, consider: $x = 0$ AND $y \geq 7$ AND $y \leq 14$. This predicate specifies one boundary equation ($x = 0$) but alternates closure, putting it in one or the other domain depending on whether $y < 7$ or $y > 14$. Treat compound predicates with respect because they're more complicated than they seem.

- **Functional Homogeneity of Bugs:** Whatever the bug is, it will not change the functional form of the boundary predicate. For example, if the predicate is $ax \geq b$, the bug will be in the value of a or b but it will not change the predicate to $ax \leq b$, say.
- **Linear Vector Space:** Most papers on domain testing, assume linear boundaries - not a bad assumption because in practice most boundary predicates are linear.
- **Loop Free Software:** Loops are problematic for domain testing. The trouble with loops is that each iteration can result in a different predicate expression (after interpretation), which means a possible domain boundary change.

NICE AND UGLY DOMAINS:

• NICE DOMAINS:

- **Where do these domains come from?**
Domains are and will be defined by an imperfect iterative process aimed at achieving (user, buyer, voter) satisfaction.
- Implemented domains can't be incomplete or inconsistent. Every input will be processed (rejection is a process), possibly forever. Inconsistent domains will be made consistent.
- Conversely, specified domains can be incomplete and/or inconsistent. Incomplete in this context means that there are input vectors for which no path is specified, and inconsistent means that there are at least two contradictory specifications over the same segment of the input space.
- Some important properties of nice domains are: **Linear, Complete, Systematic, And Orthogonal, Consistently closed, Convex and simply connected.**
- To the extent that domains have these properties domain testing is easy as testing gets.
- The bug frequency is lesser for nice domain than for ugly domains.

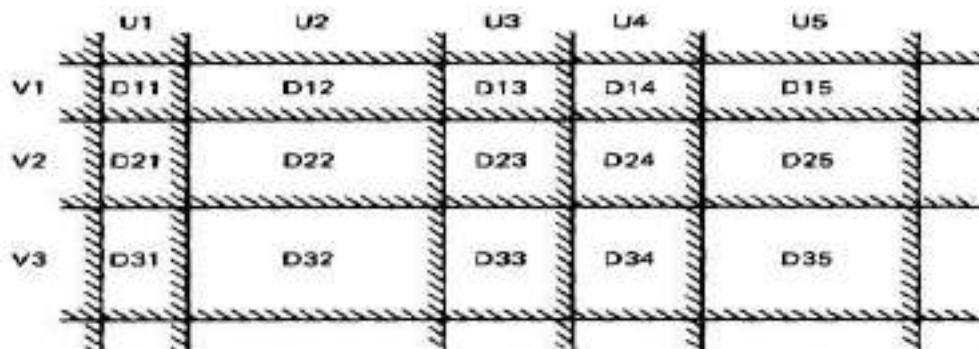


Figure 4.3: Nice Two-Dimensional Domains.

- **LINEAR AND NON LINEAR BOUNDARIES:**

- Nice domain boundaries are defined by linear inequalities or equations.
- The impact on testing stems from the fact that it takes only two points to determine a straight line and three points to determine a plane and in general $n+1$ point to determine an n -dimensional hyper plane.
- In practice more than 99.99% of all boundary predicates are either linear or can be liberalized by simple variable transformations.

- **COMPLETE BOUNDARIES:**

- Nice domain boundaries are complete in that they span the number space from plus to minus infinity in all dimensions.
- Figure 4.4 shows some incomplete boundaries. Boundaries A and E have gaps.
- Such boundaries can come about because the path that hypothetically corresponds to them is unachievable, because inputs are constrained in such a way that such values can't exist, because of compound predicates that define a single boundary, or because redundant predicates convert such boundary values into a null set.
- The advantage of complete boundaries is that one set of tests is needed to confirm the boundary no matter how many domains it bounds.
- If the boundary is chopped up and has holes in it, then every segment of that boundary must be tested for every domain it bounds.

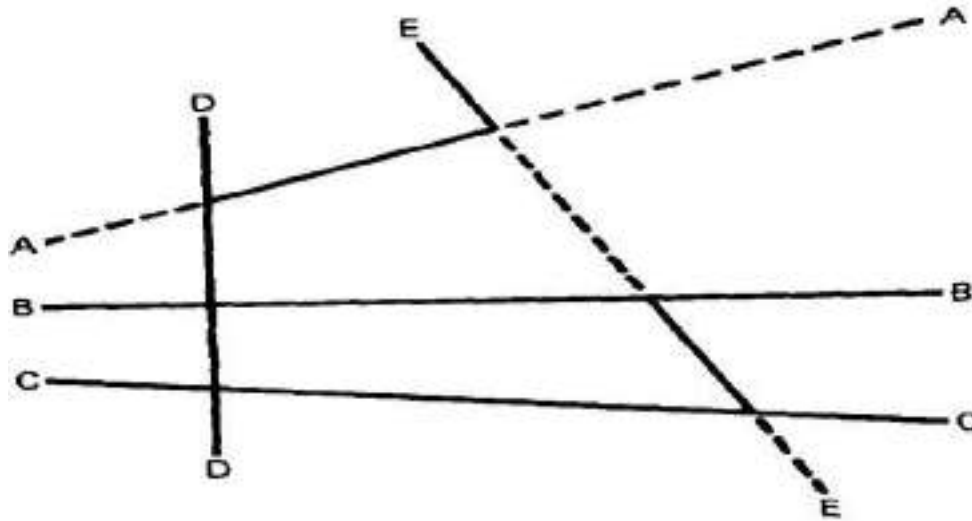


Figure 4.4: Incomplete Domain Boundaries.

- **SYSTEMATIC BOUNDARIES:**

- Systematic boundary means that boundary inequalities related by a simple function such as a constant.

In Figure 4.3 for example, the domain boundaries for u and v differ only by a constant.

$$\begin{array}{ll}
 f_1(X) \geq k_1 & \text{or } f_1(X) \geq g(1,c) \\
 f_1(X) \geq k_2 & f_2(X) \geq g(2,c) \\
 \dots\dots\dots & \dots\dots\dots \\
 f_i(X) \geq k_i & f_i(X) \geq g(i,c)
 \end{array}$$

- where f_i is an arbitrary linear function, X is the input vector, k_i and c are constants, and $g(i,c)$ is a decent function over i and c that yields a constant, such as $k + ic$.
- The first example is a set of parallel lines, and the second example is a set of systematically (e.g., equally) spaced parallel lines (such as the spokes of a wheel, if equally spaced in angles, systematic).
- If the boundaries are systematic and if you have one tied down and generate tests for it, the tests for the rest of the boundaries in that set can be automatically generated.

- **ORTHOGONAL BOUNDARIES:**

- Two boundary sets U and V (See Figure 4.3) are said to be orthogonal if every inequality in V is perpendicular to every inequality in U .
- If two boundary sets are orthogonal, then they can be tested independently
- In Figure 4.3 we have six boundaries in U and four in V . We can confirm the boundary properties in a number of tests proportional to $6 + 4 = 10$ ($O(n)$). If we tilt the boundaries to get Figure 4.5,
- we must now test the intersections. We've gone from a linear number of cases to a quadratic: from $O(n)$ to $O(n^2)$.

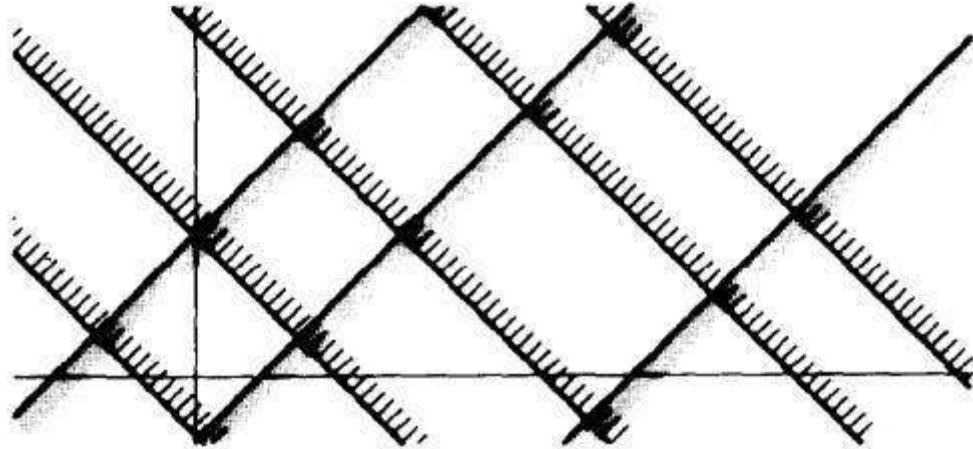


Figure 4.5: Tilted Boundaries.

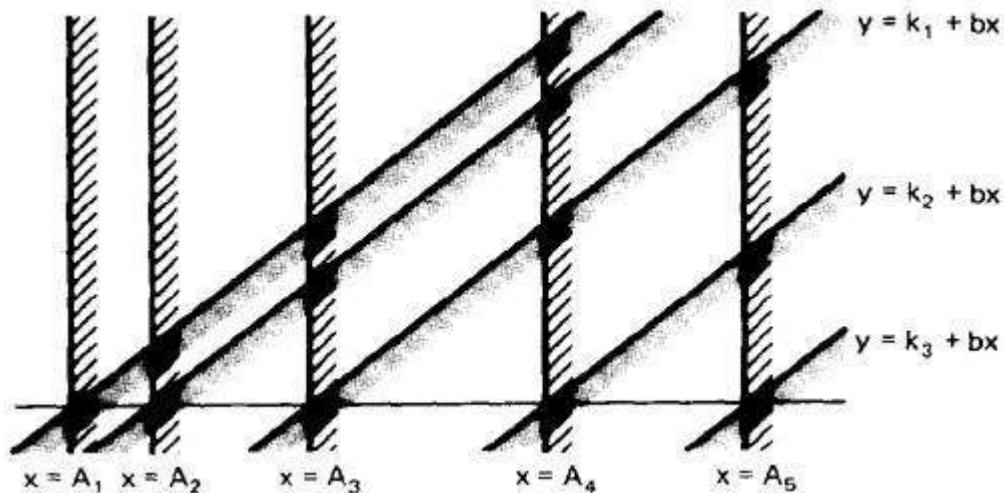


Figure 4.6: Linear, Non-orthogonal Domain Boundaries.

- Actually, there are two different but related orthogonality conditions. Sets of boundaries can be orthogonal to one another but not orthogonal to the coordinate axes (condition 1), or boundaries can be orthogonal to the coordinate axes (condition 2).
- **CLOSURE CONSISTENCY:**
 - Figure 4.6 shows another desirable domain property: boundary closures are consistent and systematic.
 - The shaded areas on the boundary denote that the boundary belongs to the domain in which the shading lies - e.g., the boundary lines belong to the domains on the right.
 - Consistent closure means that there is a simple pattern to the closures - for example, using the same relational operator for all boundaries of a set of parallel boundaries.
- **CONVEX:**
 - A geometric figure (in any number of dimensions) is convex if you can take two arbitrary points on any two different boundaries, join them by a line and all points on that line lie within the figure.
 - Nice domains are convex; dirty domains aren't.
 - You can smell a suspected concavity when you see phrases such as: ". . . except if . . .," "However . . .," ". . . but not. . . ." In programming, it's often the buts in the specification that kill you.
- **SIMPLY CONNECTED:**
 - Nice domains are simply connected; that is, they are in one piece rather than pieces all over the place interspersed with other domains.
 - Simple connectivity is a weaker requirement than convexity; if a domain is convex it is simply connected, but not vice versa.
 - Consider domain boundaries defined by a compound predicate of the (Boolean) form ABC. Say that the input space is divided into two domains, one defined by ABC and, therefore, the other defined by its negation.
 - For example, suppose we define valid numbers as those lying between 10 and 17 inclusive. The invalid numbers are the disconnected domain consisting of numbers less than 10 and greater than 17.
 - Simple connectivity, especially for default cases, may be impossible.
- **UGLY DOMAINS:**
 - Some domains are born ugly and some are uglified by bad specifications.
 - Every simplification of ugly domains by programmers can be either good or bad.
 - Programmers in search of nice solutions will "simplify" essential complexity out of existence. Testers in search of brilliant insights will be blind to essential complexity and therefore miss important cases.
 - If the ugliness results from bad specifications and the programmer's simplification is harmless, then the programmer has made ugly good.

- But if the domain's complexity is essential (e.g., the income tax code), such "simplifications" constitute bugs.
- Nonlinear boundaries are so rare in ordinary programming that there's no information on how programmers might "correct" such boundaries if they're essential.
- **AMBIGUITIES AND CONTRADICTIONS:**
 - Domain ambiguities are holes in the input space.
 - The holes may lie within the domains or in cracks between domains.
 - Two kinds of contradictions are possible: overlapped domain specifications and overlapped closure specifications
 - Figure 4.7c shows overlapped domains and Figure 4.7d shows dual closure assignment.

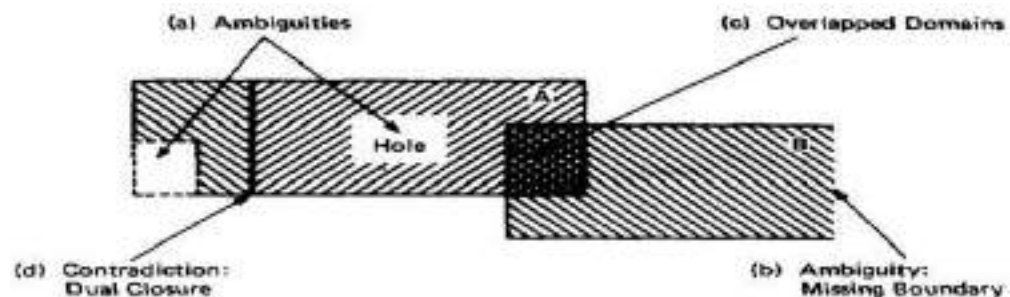


Figure 4.7: Domain Ambiguities and Contradictions.

- **SIMPLIFYING THE TOPOLOGY:**
 - The programmer's and tester's reaction to complex domains is the same - simplify
 - There are three generic cases: **concavities**, **holes** and **disconnected pieces**.
 - Programmers introduce bugs and testers misdesign test cases by: smoothing out concavities (Figure 4.8a), filling in holes (Figure 4.8b), and joining disconnected pieces (Figure 4.8c).

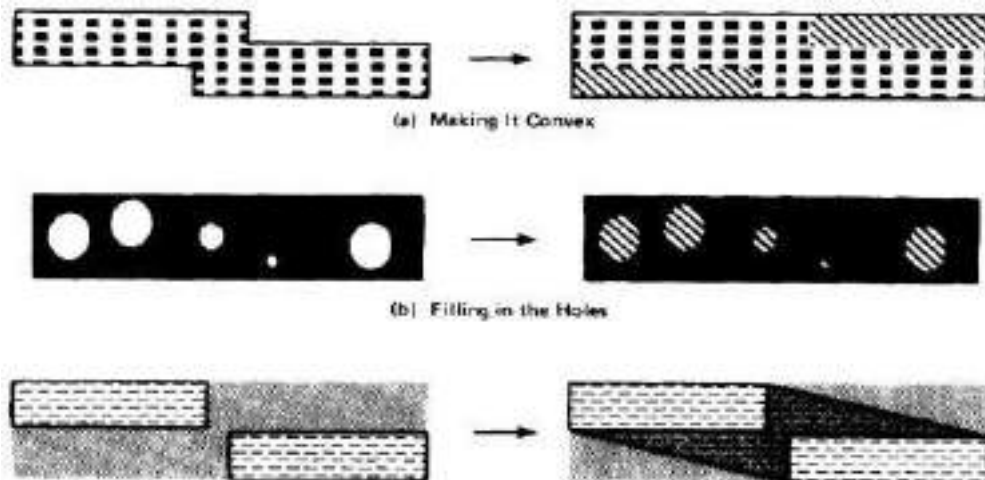


Figure 4.8: Simplifying the topology.

- **RECTIFYING BOUNDARY CLOSURES:**

- If domain boundaries are parallel but have closures that go every which way (left, right, left . . .) the natural reaction is to make closures go the same way (see Figure 4.9).

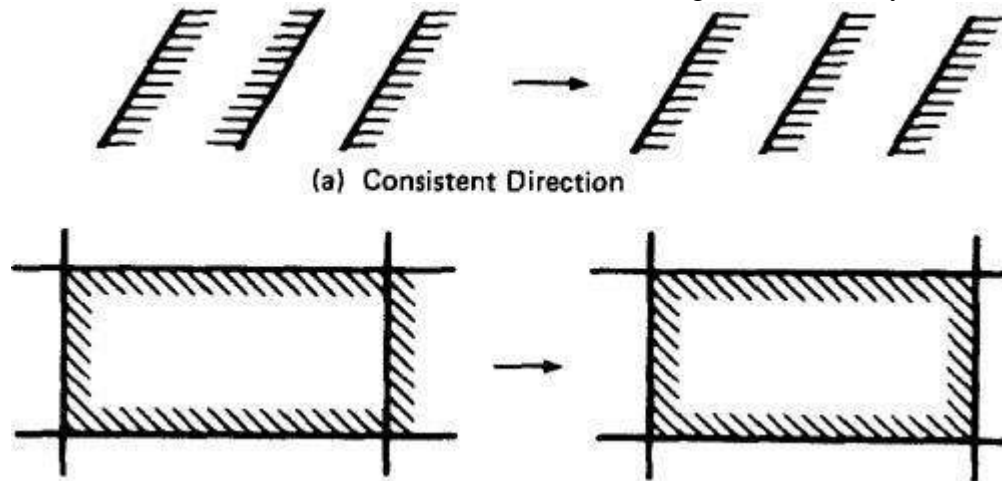


Figure 4.9: Forcing Closure Consistency.

DOMAIN TESTING:

- **DOMAIN TESTING STRATEGY:** The domain-testing strategy is simple, although possibly tedious (slow).
 - Domains are defined by their boundaries; therefore, domain testing concentrates test points on or near boundaries.
 - Classify what can go wrong with boundaries, then define a test strategy for each case. Pick enough points to test for all recognized kinds of boundary errors.
 - Because every boundary serves at least two different domains, test points used to check one domain can also be used to check adjacent domains. Remove redundant test points.
 - Run the tests and by posttest analysis (the tedious part) determine if any boundaries are faulty and if so, how.
 - Run enough tests to verify every boundary of every domain.
- **DOMAIN BUGS AND HOW TO TEST FOR THEM:**
 - An **interior point** (Figure 4.10) is a point in the domain such that all points within an arbitrarily small distance (called an epsilon neighborhood) are also in the domain.
 - A **boundary point** is one such that within an epsilon neighborhood there are points both in the domain and not in the domain.
 - An **extreme point** is a point that does not lie between any two other arbitrary but distinct points of a (convex) domain.

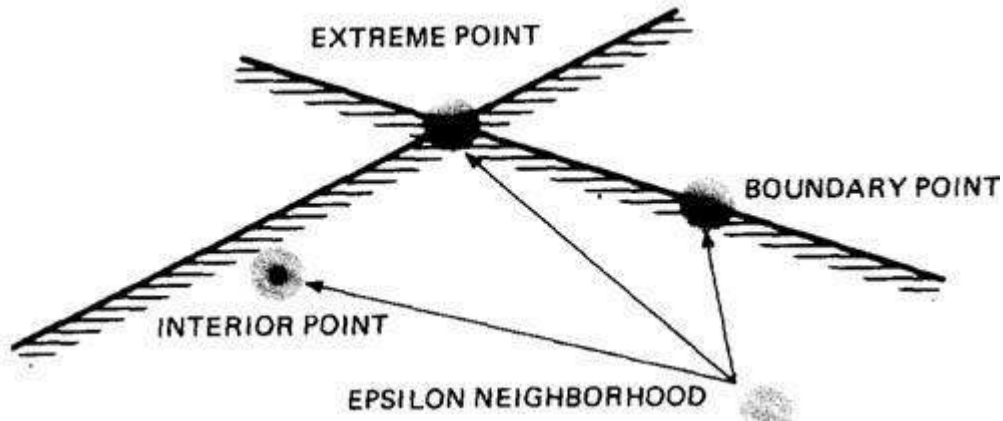


Figure 4.10: Interior, Boundary and Extreme points.

- An **on point** is a point on the boundary.
- If the domain boundary is closed, an **off point** is a point near the boundary but in the adjacent domain.
- If the boundary is open, an off point is a point near the boundary but in the domain being tested; see Figure 4.11. You can remember this by the acronym COOOOI: Closed Off Outside, Open Off Inside.

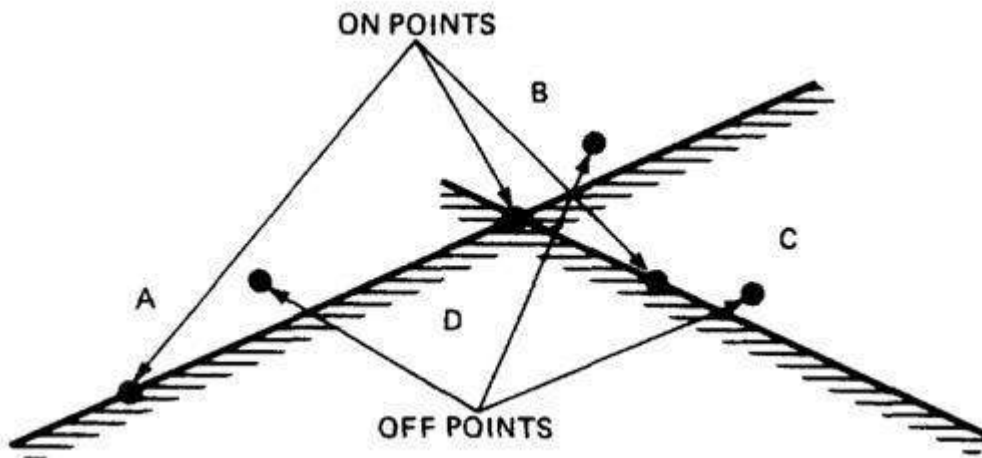


Figure 4.11: On points and Off points.

- Figure 4.12 shows generic domain bugs: closure bug, shifted boundaries, tilted boundaries, extra boundary, missing boundary.

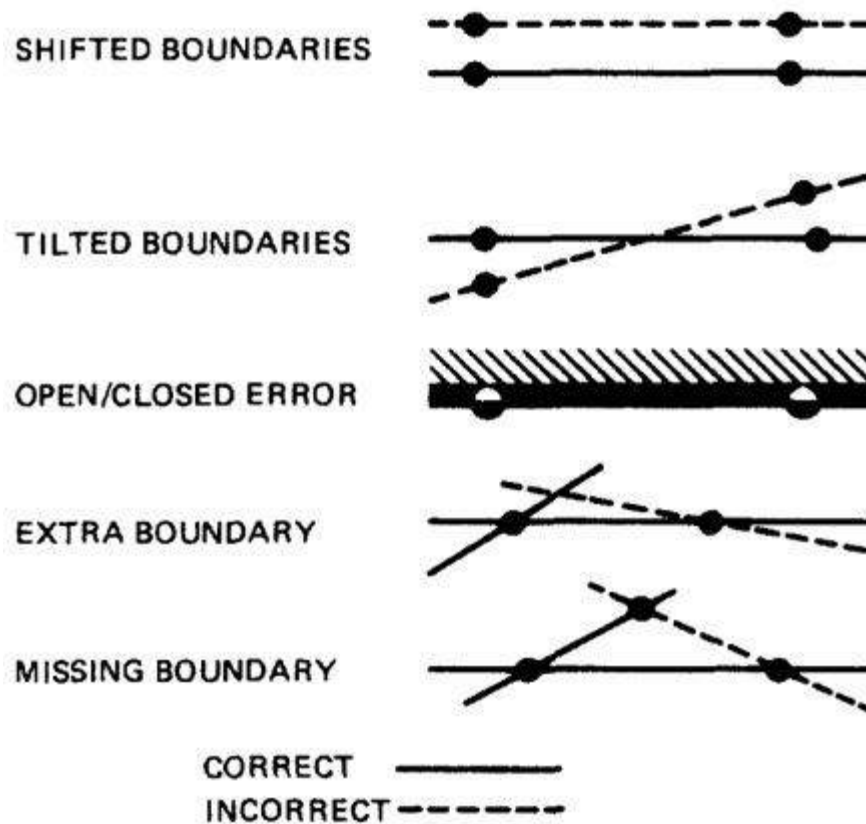


Figure 4.12: Generic Domain Bugs.

TESTING ONE DIMENSIONAL DOMAIN:

The closure can be wrong (i.e., assigned to the wrong domain) or the boundary (a point in this case) can be shifted one way or the other, we can be missing a boundary, or we can have an extra boundary.

1. Figure 4.13 shows possible domain bugs for a one-dimensional open domain boundary.
2. In Figure 4.13a we assumed that the boundary was to be open for A. The bug we're looking for is a closure error, which converts $>$ to \geq or $<$ to \leq (Figure 4.13b). One test (marked x) on the boundary point detects this bug because processing for that point will go to domain A rather than B.
3. In Figure 4.13c we've suffered a boundary shift to the left. The test point we used for closure detects this bug because the bug forces the point from the B domain, where it should be, to A processing. Note that we can't distinguish between a shift and a closure error, but we do know that we have a bug.

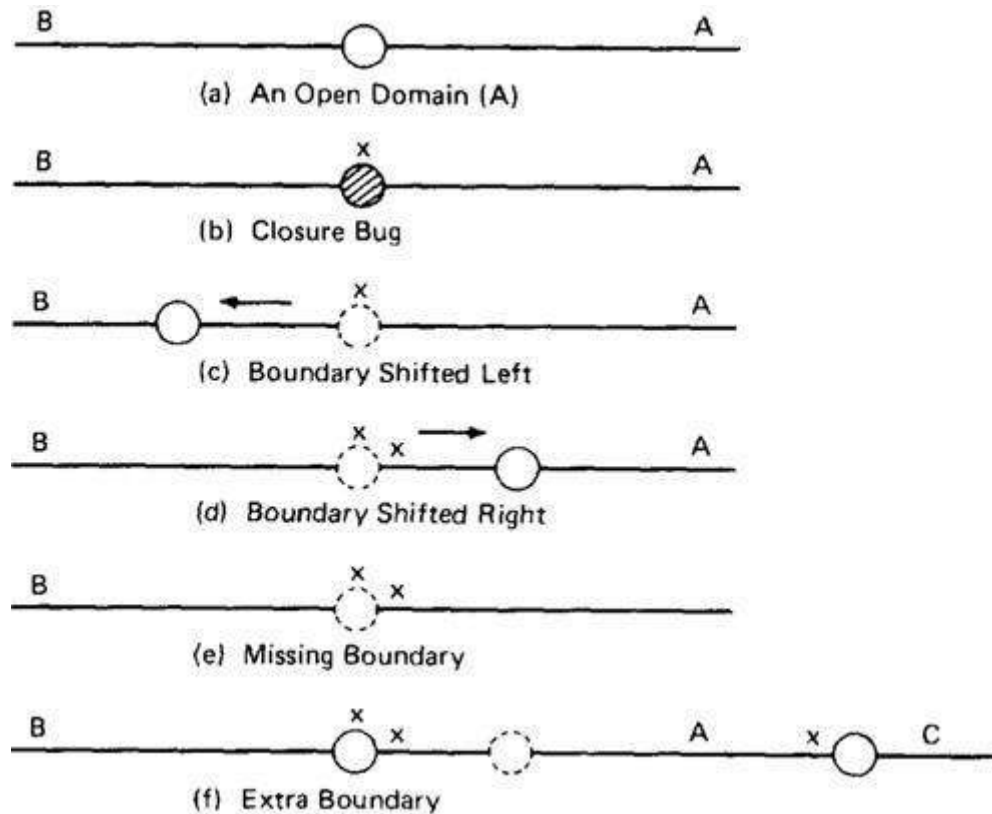


Figure 4.13: One Dimensional Domain Bugs, Open Boundaries.

4. Figure 4.13d shows a shift the other way. The on point doesn't tell us anything because the boundary shift doesn't change the fact that the test point will be processed in B. To detect this shift we need a point close to the boundary but within A. The boundary is open, therefore by definition, the off point is in A (Open Off Inside).
5. The same open off point also suffices to detect a missing boundary because what should have been processed in A is now processed in B.
6. To detect an extra boundary we have to look at two domain boundaries. In this context an extra boundary means that A has been split in two. The two off points that we selected before (one for each boundary) does the job. If point C had been a closed boundary, the on test point at C would do it.
7. For closed domains look at Figure 4.14. As for the open boundary, a test point on the boundary detects the closure bug. The rest of the cases are similar to the open boundary, except now the strategy requires off points just outside the domain.

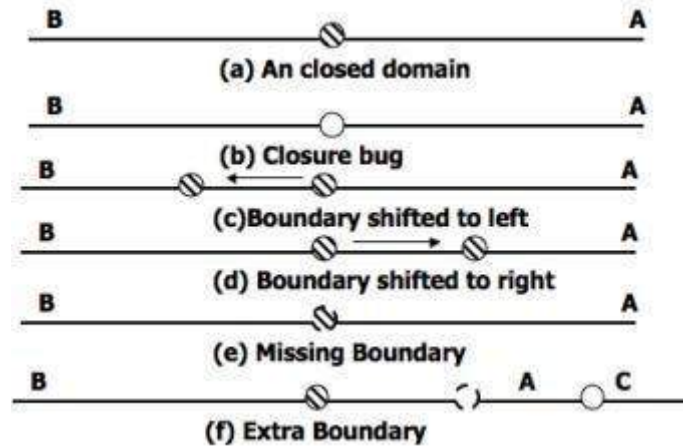


Figure 4.14: One Dimensional Domain Bugs, Closed Boundaries.

- **TESTING TWO DIMENSIONAL DOMAINS:**

1. Figure 4.15 shows possible domain boundary bugs for a two-dimensional domain.
2. A and B are adjacent domains and the boundary is closed with respect to A, which means that it is open with respect to B.

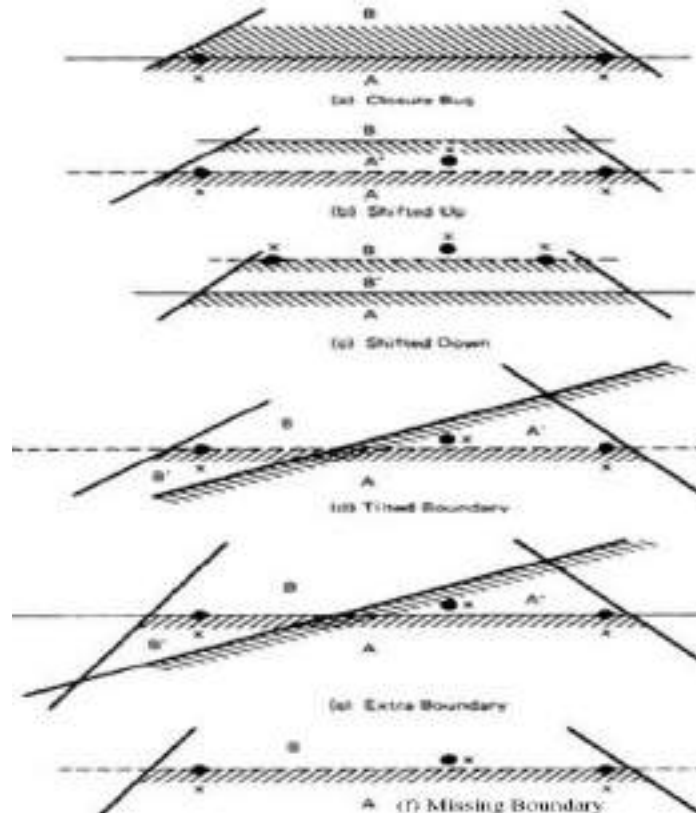


Figure 4.15: Two Dimensional Domain Bugs.

3. **For Closed Boundaries:**

Closure Bug: Figure 4.15a shows a faulty closure, such as might be caused by using a wrong operator (for example, $x \geq k$ when $x > k$ was intended, or vice

versa). The two on points detect this bug because those values will get B rather than A processing.

1. **Shifted Boundary:** In Figure 4.15b the bug is a shift up, which converts part of domain B into A processing, denoted by A'. This result is caused by an incorrect constant in a predicate, such as $x + y \geq 17$ when $x + y \geq 7$ was intended. The off point (closed off outside) catches this bug. Figure 4.15c shows a shift down that is caught by the two on points.
 2. **Tilted Boundary:** A tilted boundary occurs when coefficients in the boundary inequality are wrong. For example, $3x + 7y > 17$ when $7x + 3y > 17$ was intended. Figure 4.15d has a tilted boundary, which creates erroneous domain segments A' and B'. In this example the bug is caught by the left on point.
 3. **Extra Boundary:** An extra boundary is created by an extra predicate. An extra boundary will slice through many different domains and will therefore cause many test failures for the same bug. The extra boundary in Figure 4.15e is caught by two on points, and depending on which way the extra boundary goes, possibly by the off point also.
 4. **Missing Boundary:** A missing boundary is created by leaving a boundary predicate out. A missing boundary will merge different domains and will cause many test failures although there is only one bug. A missing boundary, shown in Figure 4.15f, is caught by the two on points because the processing for A and B is the same - either A or B processing.
- **PROCEDURE FOR TESTING:** The procedure is conceptually is straight forward. It can be done by hand for two dimensions and for a few domains and practically impossible for more than two variables.
 - 1 Identify input variables.
 - 2 Identify variable which appear in domain defining predicates, such as control flow predicates.
 - 3 Interpret all domain predicates in terms of input variables.
 - 4 For p binary predicates, there are at most 2^p combinations of TRUE-FALSE values and therefore, at most 2^p domains. Find the set of all non null domains. The result is a boolean expression in the predicates consisting a set of AND terms joined by OR's. For example $ABC+DEF+GHI.....$ Where the capital letters denote predicates. Each product term is a set of linear inequality that defines a domain or a part of a multiply connected domains.
 - 5 Solve these inequalities to find all the extreme points of each domain using any of the linear programming methods.

DOMAIN AND INTERFACE TESTING

• INTRODUCTION:

- Recall that we defined integration testing as testing the correctness of the interface between two otherwise correct components.

- Components A and B have been demonstrated to satisfy their component tests, and as part of the act of integrating them we want to investigate possible inconsistencies across their interface.
 - Interface between any two components is considered as a subroutine call.
 - We're looking for bugs in that "call" when we do interface testing.
 - Let's assume that the call sequence is correct and that there are no type incompatibilities.
 - For a single variable, the domain span is the set of numbers between (and including) the smallest value and the largest value. For every input variable we want (at least): compatible domain spans and compatible closures (Compatible but need not be Equal).
- **DOMAINS AND RANGE:**
 - The set of output values produced by a function is called the **range** of the function, in contrast with the **domain**, which is the set of input values over which the function is defined.
 - For most testing, our aim has been to specify input values and to predict and/or confirm output values that result from those inputs.
 - Interface testing requires that we select the output values of the calling routine *i.e.* caller's range must be compatible with the called routine's domain.
 - An interface test consists of exploring the correctness of the following mappings: caller domain --> caller range (caller unit test)
caller range --> called domain (integration test) called domain --> called range (called unit test)
 - **CLOSURE COMPATIBILITY:**
 - Assume that the caller's range and the called domain spans the same numbers - for example, 0 to 17.
 - Figure 4.16 shows the four ways in which the caller's range closure and the called's domain closure can agree.
 - The thick line means closed and the thin line means open. Figure 4.16 shows the four cases consisting of domains that are closed both on top (17) and bottom (0), open top and closed bottom, closed top and open bottom, and open top and bottom.

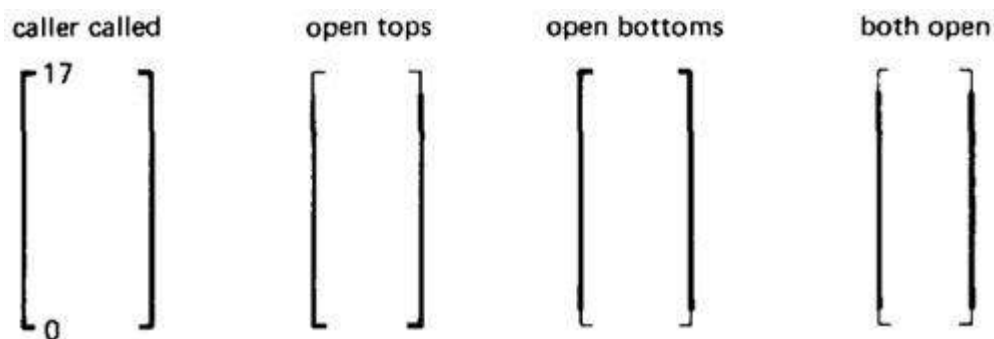


Figure 4.16: Range / Domain Closure Compatibility.

- Figure 4.17 shows the twelve different ways the caller and the called can disagree about closure. Not all of them are necessarily bugs. The four cases in which a

caller boundary is open and the called is closed (marked with a "?") are probably not buggy. It means that the caller will not supply such values but the called can accept them.

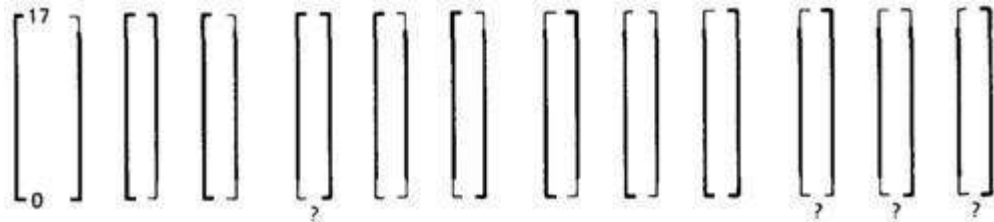


Figure 4.17: Equal-Span Range / Domain Compatibility Bugs.

- **SPAN COMPATIBILITY:**

- Figure 4.18 shows three possibly harmless span incompatibilities.

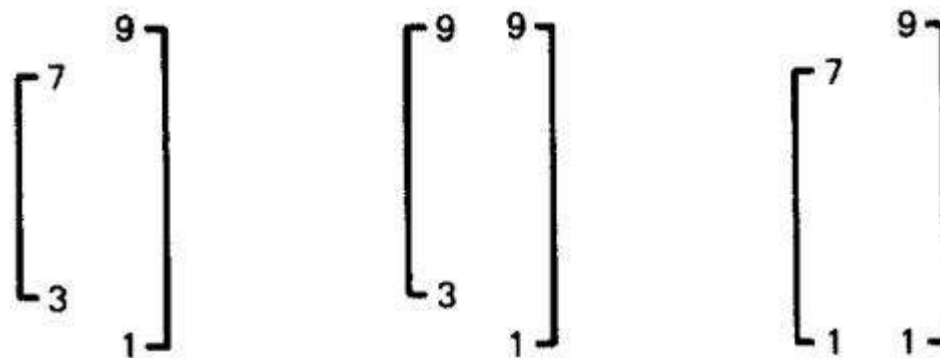


Figure 4.18: Harmless Range / Domain Span incompatibility bug (Caller Span is smaller than Called).

- In all cases, the caller's range is a subset of the called's domain. That's not necessarily a bug.
- The routine is used by many callers; some require values inside a range and some don't. This kind of span incompatibility is a bug only if the caller expects the called routine to validate the called number for the caller.
- Figure 4.19a shows the opposite situation, in which the called routine's domain has a smaller span than the caller expects. All of these examples are buggy.

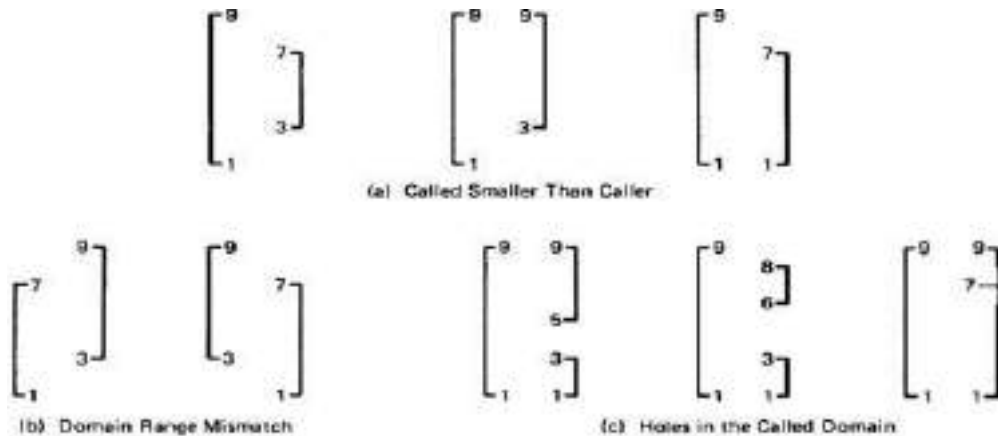


Figure 4.19: Buggy Range / Domain Mismatches

- In Figure 4.19b the ranges and domains don't line up; hence good values are rejected, bad values are accepted, and if the called routine isn't robust enough, we have crashes.
 - Figure 4.19c combines these notions to show various ways we can have holes in the domain: these are all probably buggy.
- **INTERFACE RANGE / DOMAIN COMPATIBILITY TESTING:**
 - For interface testing, bugs are more likely to concern single variables rather than peculiar combinations of two or more variables.
 - Test every input variable independently of other input variables to confirm compatibility of the caller's range and the called routine's domain span and closure of every domain defined for that variable.
 - There are two boundaries to test and it's a one-dimensional domain; therefore, it requires one on and one off point per boundary or a total of two on points and two off points for the domain - pick the off points appropriate to the closure (COOOOI).
 - Start with the called routine's domains and generate test points in accordance to the domain-testing strategy used for that routine in component testing.
 - Unless you're a mathematical whiz you won't be able to do this without tools for more than one variable at a time.

UNIT IV

PATHS, PATH PRODUCTS AND REGULAR EXPRESSIONS

PATH PRODUCTS AND PATH EXPRESSION:

- **MOTIVATION:**

- Flow graphs are being an abstract representation of programs.
- Any question about a program can be cast into an equivalent question about an appropriate flowgraph.
- Most software development, testing and debugging tools use flow graphs analysis techniques.

- **PATH PRODUCTS:**

- Normally flow graphs used to denote only control flow connectivity.
- The simplest weight we can give to a link is a name.
- Using link names as weights, we then convert the graphical flow graph into an equivalent algebraic like expressions which denotes the set of all possible paths from entry to exit for the flow graph.
- Every link of a graph can be given a name.
- The link name will be denoted by lower case italic letters In tracing a path or path segment through a flow graph, you traverse a succession of link names.
- The name of the path or path segment that corresponds to those links is expressed naturally by concatenating those link names.
- For example, if you traverse links a,b,c and d along some path, the name for that path segment is abcd. This path name is also called a **path product**. Figure 5.1 shows some examples:

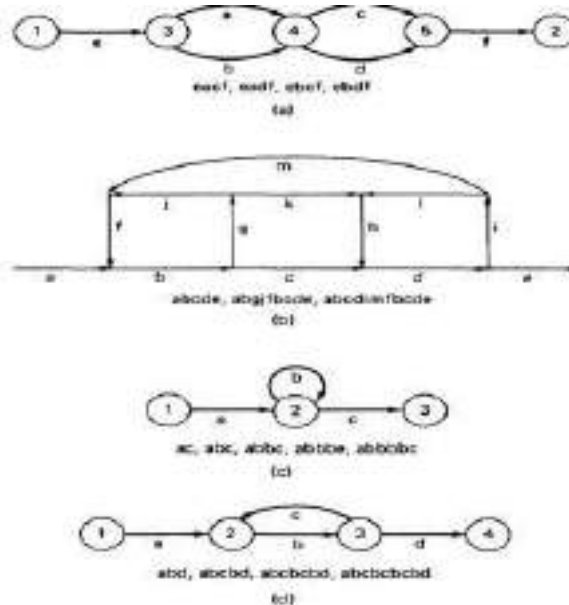


Figure 5.1: Examples of paths.

- **PATH EXPRESSION:**

- Consider a pair of nodes in a graph and the set of paths between those node.
- Denote that set of paths by Upper case letter such as X,Y. From Figure 5.1c, the members of the path set can be listed as follows:
ac, abc, abbc, abbbc, abbbbc.....
- Alternatively, the same set of paths can be denoted by :
ac+abc+abbc+abbbc+abbbbc+.....
- The + sign is understood to mean "or" between the two nodes of interest, paths ac, or abc, or abbc, and so on can be taken.
- Any expression that consists of path names and "OR"s and which denotes a set of paths between two nodes is called a "**Path Expression**".

- **PATH PRODUCTS:**

- The name of a path that consists of two successive path segments is conveniently expressed by the concatenation or **Path Product** of the segment names.
- For example, if X and Y are defined as X=abcde,Y=fghij,then the path corresponding to X followed by Y is denoted by
XY=abcdefghij
- Similarly, YX=fghijabcde aX=aabcde Xa=abcdea XaX=abcdeaabcde
- If X and Y represent sets of paths or path expressions, their product represents the set of paths that can be obtained by following every element of X by any element of Y in all possible ways. For example,
- X = abc + def + ghi
- Y = uvw + z Then,
XY = abcuvw + defuvw + ghiuvw + abcz + defz + ghiz
- If a link or segment name is repeated, that fact is denoted by an exponent. The exponent's value denotes the number of repetitions:
- $a^1 = a$; $a^2 = aa$; $a^3 = aaa$; $a^n = aaaa \dots n$ times. Similarly, if X = abcde then

$$X^1 = abcde$$

$$X^2 = abcdeabcde = (abcde)^2$$

$$X^3 = abcdeabcdeabcde = (abcde)^2abcde \\ = abcde(abcde)^2 = (abcde)^3$$

- The path product is not commutative (that is $XY \neq YX$).
- The path product is Associative.

$$\text{RULE 1: } A(BC) = (AB)C = ABC$$

where A,B,C are path names, set of path names or path expressions.

- The zeroth power of a link name, path product, or path expression is also needed for completeness. It is denoted by the numeral "1" and denotes the "path" whose length is zero - that is, the path that doesn't have any links.

- $a^0 = 1$
- $X^0 = 1$

- **PATH SUMS:**

- The "+" sign was used to denote the fact that path names were part of the same set of paths.
- The "PATH SUM" denotes paths in parallel between nodes.
- Links a and b in Figure 5.1a are parallel paths and are denoted by $a + b$. Similarly, links c and d are parallel paths between the next two nodes and are denoted by $c + d$.
- The set of all paths between nodes 1 and 2 can be thought of as a set of parallel paths and denoted by $eacf+eadf+ebcf+ebdf$.
- If X and Y are sets of paths that lie between the same pair of nodes, then $X+Y$ denotes the UNION of those set of paths. For example, in Figure 5.2:

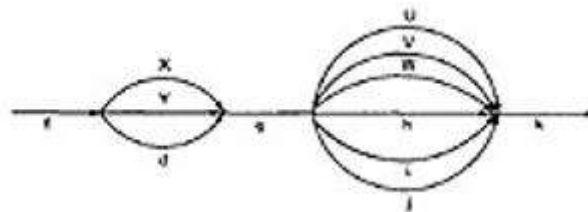


Figure 5.2: Examples of path sums.

The first set of parallel paths is denoted by $X + Y + d$ and the second set by $U + V + W + h + i + j$. The set of all paths in this flowgraph is $f(X + Y + d)g(U + V + W + h + i + j)k$

- The path is a set union operation, it is clearly Commutative and Associative.
- RULE 2: $X+Y=Y+X$
- RULE 3: $(X+Y)+Z=X+(Y+Z)=X+Y+Z$

- **DISTRIBUTIVE LAWS:**

- The product and sum operations are distributive, and the ordinary rules of multiplication apply; that is
RULE 4: $A(B+C)=AB+AC$ and $(B+C)D=BD+CD$
- Applying these rules to the below Figure 5.1a yields
 $e(a+b)(c+d)f=e(ac+ad+bc+bd)f = eacf+eadf+ebcf+ebdf$

- **ABSORPTION RULE:**

- If X and Y denote the same set of paths, then the union of these sets is unchanged; consequently,
RULE 5: $X+X=X$ (Absorption Rule)
 - If a set consists of paths names and a member of that set is added to it, the "new" name, which is already in that set of names, contributes nothing and can be ignored.
 - For example,
if $X=a+aa+abc+abcd+def$ then
 $X+a = X+aa = X+abc = X+abcd = X+def = X$
- It follows that any arbitrary sum of identical path expressions reduces to the same path expression.

- **LOOPS:**

Loops can be understood as an infinite set of parallel paths. Say that the loop consists of a single link b . then the set of all paths through that loop point is $b^0 + b^1 + b^2 + b^3 + b^4 + b^5 + \dots$

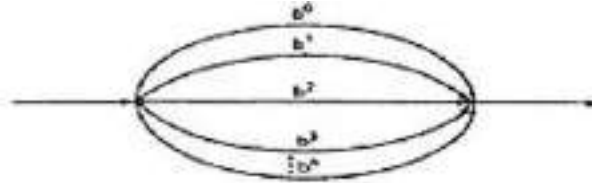


Figure 5.3: Examples of path loops.

This potentially infinite sum is denoted by b^* for an individual link and by X^*



Figure 5.4: Another example of path loops.

- The path expression for the above figure is denoted by the notation:
 $ab^*c = ac + abc + abbc + abbbc + \dots$
- Evidently,
 $aa^* = a^*a = a^+$ and $XX^* = X^*X = X^+$
- It is more convenient to denote the fact that a loop cannot be taken more than a certain, say n , number of times.
- A bar is used under the exponent to denote the fact as follows: $X^n = X^0 + X^1 + X^2 + X^3 + X^4 + X^5 + \dots + X^n$

RULES 6 - 16:

- The following rules can be derived from the previous rules:
- **RULE 6:** $X^n + X^m = X^n$ if $n > m$ **RULE 6:** $X^n + X^m = X^m$ if $m > n$ **RULE 7:** $X^n X^m = X^{n+m}$
- RULE 8:** $X^n X^* = X^* X^n = X^*$ **RULE 9:** $X^n X^+ = X^+ X^n = X^+$ **RULE 10:** $X^* X^+ = X^+ X^* = X^+$
- RULE 11:** $1 + 1 = 1$
- RULE 12:** $1X = X1 = X$
- Following or preceding a set of paths by a path of zero length does not change the set.
- RULE 13:** $1^n = 1^0 = 1^* = 1^+ = 1$
- No matter how often you traverse a path of zero length, It is a path of zero length.
- RULE 14:** $1^+ + 1 = 1^* = 1$

The null set of paths is denoted by the numeral 0. it obeys the following rules:

RULE 15: $X + 0 = 0 + X = X$

RULE 16: $0X = X0 = 0$

If you block the paths of a graph for or aft by a graph that has no paths , there won't be any paths.

REDUCTION PROCEDURE:

• REDUCTION PROCEDURE ALGORITHM:

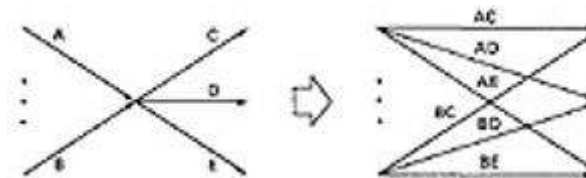
- This section presents a reduction procedure for converting a flowgraph whose links are labeled with names into a path expression that denotes the set of all entry/exit paths in that flowgraph. The procedure is a node-by-node removal algorithm.
- The steps in Reduction Algorithm are as follows:
 1. Combine all serial links by multiplying their path expressions.
 2. Combine all parallel links by adding their path expressions.
 3. Remove all self-loops (from any node to itself) by replacing them with a link of the form X^* , where X is the path expression of the link in that loop.

STEPS 4 - 8 ARE IN THE ALGORITHM'S LOOP:

4. Select any node for removal other than the initial or final node. Replace it with a set of equivalent links whose path expressions correspond to all the ways you can form a product of the set of inlinks with the set of outlinks of that node.
 5. Combine any remaining serial links by multiplying their path expressions.
 6. Combine all parallel links by adding their path expressions.
 7. Remove all self-loops as in step 3.
 8. Does the graph consist of a single link between the entry node and the exit node? If yes, then the path expression for that link is a path expression for the original flowgraph; otherwise, return to step 4.
- A flowgraph can have many equivalent path expressions between a given pair of nodes; that is, there are many different ways to generate the set of all paths between two nodes without affecting the content of that set.
 - The appearance of the path expression depends, in general, on the order in which nodes are removed.

• CROSS-TERM STEP (STEP 4):

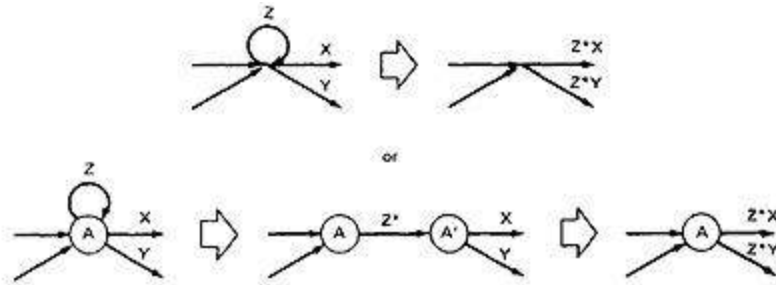
- The cross - term step is the fundamental step of the reduction algorithm.
- It removes a node, thereby reducing the number of nodes by one.
- Successive applications of this step eventually get you down to one entry and one exit node. The following diagram shows the situation at an arbitrary node that has been selected for removal:



- From the above diagram, one can infer:
- $(a + b)(c + d + e) = ac + ad + ae + bc + bd + be$

• LOOP REMOVAL OPERATIONS:

- There are two ways of looking at the loop-removal operation:



- In the first way, we remove the self-loop and then multiply all outgoing links by Z^* .
- In the second way, we split the node into two equivalent nodes, call them A and A' and put in a link between them whose path expression is Z^* . Then we remove node A' using steps 4 and 5 to yield outgoing links whose path expressions are Z^*X and Z^*Y .

• **A REDUCTION PROCEDURE - EXAMPLE:**

- Let us see by applying this algorithm to the following graph where we remove several nodes in order; that is

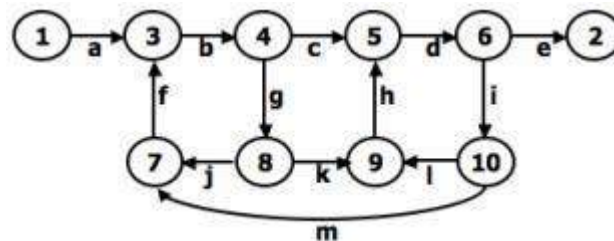
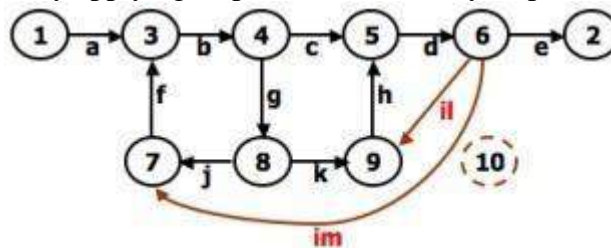
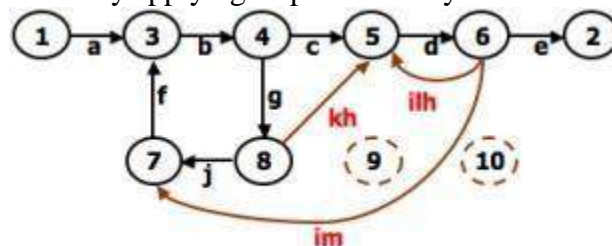


Figure 5.5: Example Flowgraph for demonstrating reduction procedure.

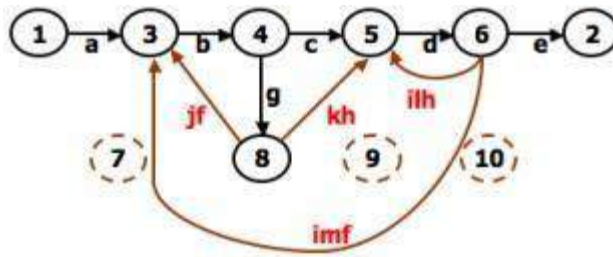
- Remove node 10 by applying step 4 and combine by step 5 to yield



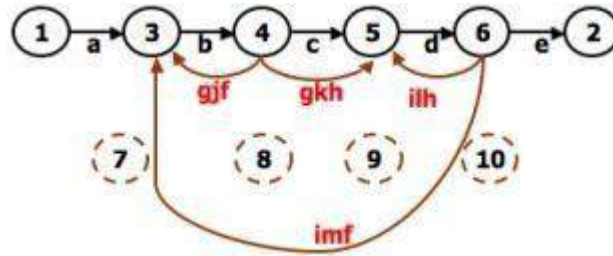
- Remove node 9 by applying step 4 and 5 to yield



- Remove node 7 by steps 4 and 5, as follows:

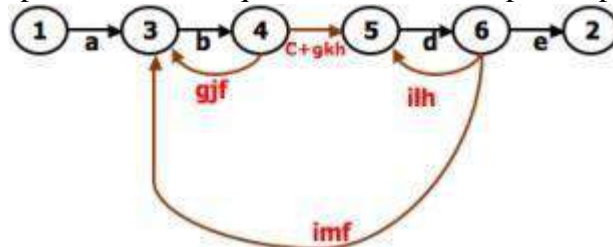


- Remove node 8 by steps 4 and 5, to obtain:



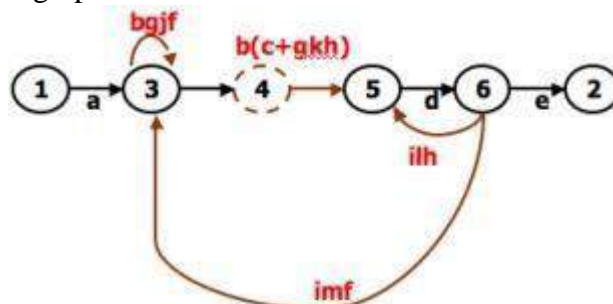
- **PARALLEL TERM (STEP 6):**

Removal of node 8 above led to a pair of parallel links between nodes 4 and 5. combine them to create a path expression for an equivalent link whose path expression is $c+gkh$; that is

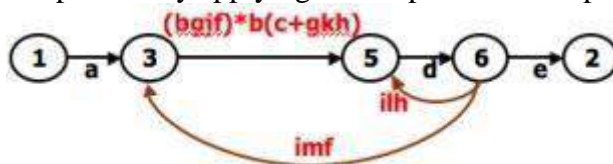


- **LOOP TERM (STEP 7):**

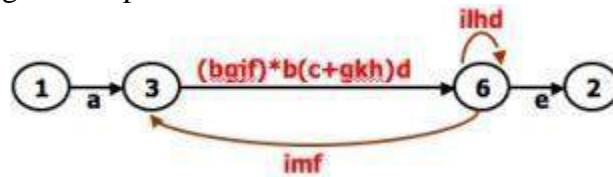
Removing node 4 leads to a loop term. The graph has now been replaced with the following equivalent simpler graph:



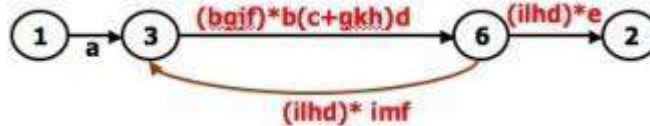
- Continue the process by applying the loop-removal step as follows:



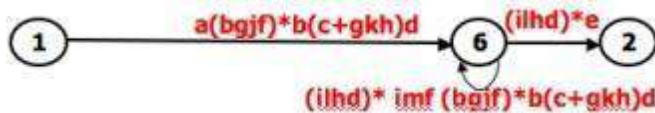
- Removing node 5 produces:



- Remove the loop at node 6 to yield:



- Remove node 3 to yield



- Removing the loop and then node 6 result in the following expression:
 $a(bgjf)*b(c+gkh)d((ilhd)*imf(bgjf)*b(c+gkh)d)*(ilhd)*e$
- You can practice by applying the algorithm on the following flowgraphs and generate their respective path expressions:

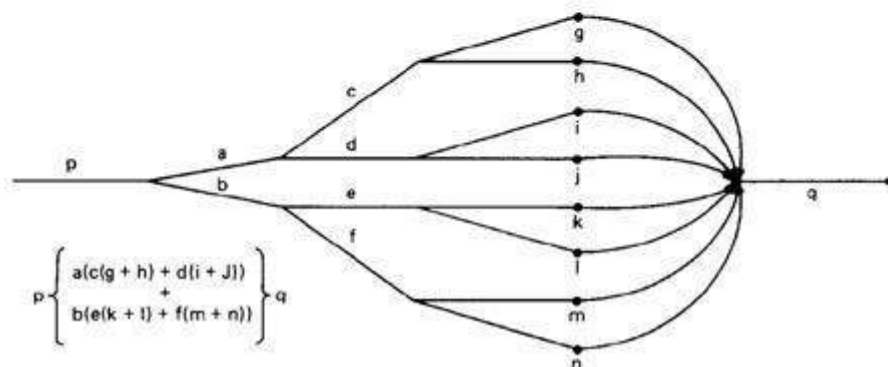
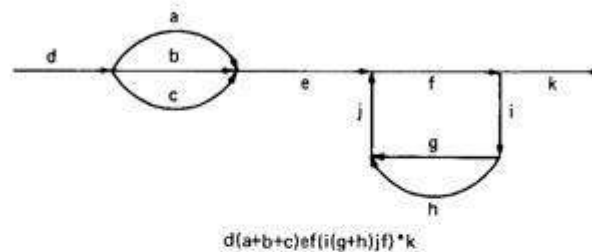
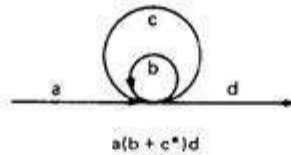


Figure 5.6: Some graphs and their path expressions.

APPLICATIONS:

- The purpose of the node removal algorithm is to present one very generalized concept- the path expression and way of getting it.
- Every application follows this common pattern:
 1. Convert the program or graph into a path expression.
 2. Identify a property of interest and derive an appropriate set of "arithmetic" rules that characterizes the property.

Replace the link names by the link weights for the property of interest. The path expression has now been converted to an expression in some algebra, such as

1. Ordinary algebra, regular expressions, or boolean algebra. This algebraic expression summarizes the property of interest over the set of all paths.
2. Simplify or evaluate the resulting "algebraic" expression to answer the question you asked.

• HOW MANY PATHS IN A FLOW GRAPH ?

- The question is not simple. Here are some ways you could ask it:
 1. What is the maximum number of different paths possible?
 2. What is the fewest number of paths possible?
 3. How many different paths are there really?
 4. What is the average number of paths?
- Determining the actual number of different paths is an inherently difficult problem because there could be unachievable paths resulting from correlated and dependent predicates.
- If we know both of these numbers (maximum and minimum number of possible paths) we have a good idea of how complete our testing is.
- Asking for "the average number of paths" is meaningless.

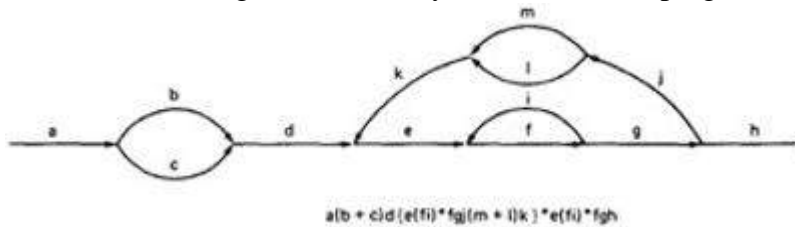
• MAXIMUM PATH COUNT ARITHMETIC:

- Label each link with a link weight that corresponds to the number of paths that link represents.
- Also mark each loop with the maximum number of times that loop can be taken. If the answer is infinite, you might as well stop the analysis because it is clear that the maximum number of paths will be infinite.
- There are three cases of interest: parallel links, serial links, and loops.

Case	Path expression	Weight expression
Parallels	$A+B$	W_A+W_B
Series	AB	$W_A W_B$
Loop	A^n	$\sum_{j=0}^n W_A^j$

- This arithmetic is an ordinary algebra. The weight is the number of paths in each set.
- **EXAMPLE:**

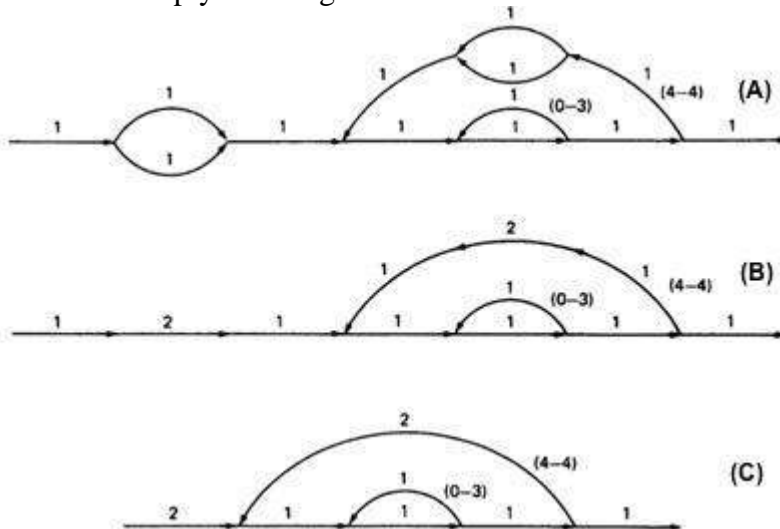
- The following is a reasonably well-structured program.



Each link represents a single link and consequently is given a weight of "1" to start. Let's say the outer loop will be taken exactly four times and inner Loop Can be taken zero or three times Its path expression, with a little work, is:

Path expression: $a(b+c)d\{e(fi)*fgj(m+l)k\}*e(fi)*fgh$

- **A:** The flow graph should be annotated by replacing the link name with the maximum of paths through that link (1) and also note the number of times for looping.
- **B:** Combine the first pair of parallel loops outside the loop and also the pair in the outer loop.
- **C:** Multiply the things out and remove nodes to clear the clutter.



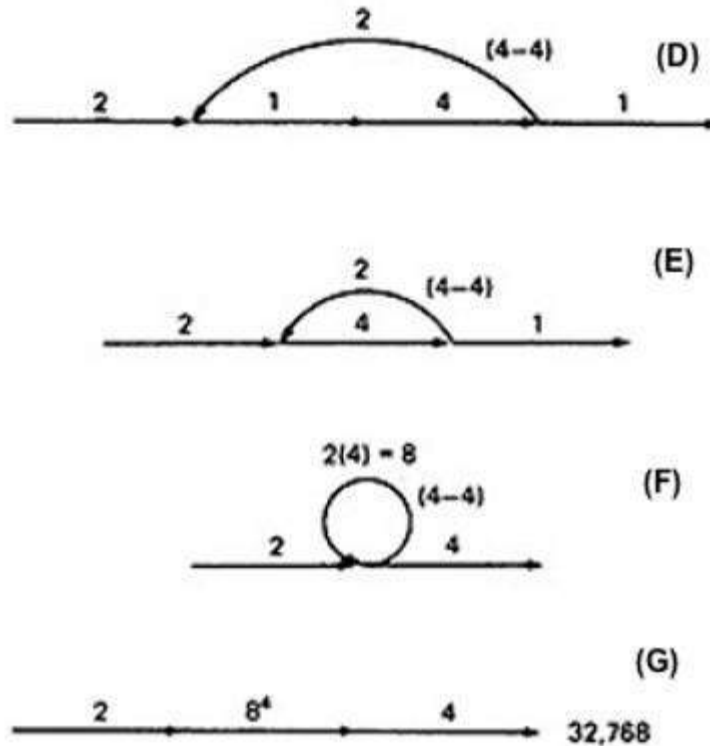
1. For the Inner Loop:

D: Calculate the total weight of inner loop, which can execute a min. of 0 times and max. of 3 times. So, it inner loop can be evaluated as follows:

$$1^3 = 1^0 + 1^1 + 1^2 + 1^3 = 1 + 1 + 1 + 1 = 4$$

- E:** Multiply the link weights inside the loop: $1 \times 4 = 4$
- F:** Evaluate the loop by multiplying the link weights: $2 \times 4 = 8$.
- G:** Simplifying the loop further results in the total maximum number of paths in the flowgraph:

$$2 \times 8^4 \times 2 = 32,768.$$



Alternatively, you could have substituted a "1" for each link in the path expression and then simplified, as follows:

$$\begin{aligned}
 & \mathbf{a(b+c)d\{e(fi)*fgj(m+l)k\}*e(fi)*fgh} \\
 &= 1(1 + 1)1(1(1 \times 1)^3 1 \times 1 \times 1(1 + 1)1)^4 1(1 \times 1)^3 1 \times 1 \times 1 \\
 &= 2(1^3 1 \times (2))^4 1^3 \\
 &= 2(4 \times 2)^4 \times 4 \\
 &= 2 \times 8^4 \times 4 = 32,768
 \end{aligned}$$

This is the same result we got graphically. Actually, the outer loop should be taken exactly four times. That doesn't mean it will be taken zero or four times. Consequently, there is a superfluous "4" on the outlink in the last step. Therefore the maximum number of different paths is 8192 rather than 32,768.

STRUCTURED FLOWGRAPH:

Structured code can be defined in several different ways that do not involve ad-hoc rules such as not using GOTOs.

A structured flowgraph is one that can be reduced to a single link by successive application of the transformations of Figure 5.7.

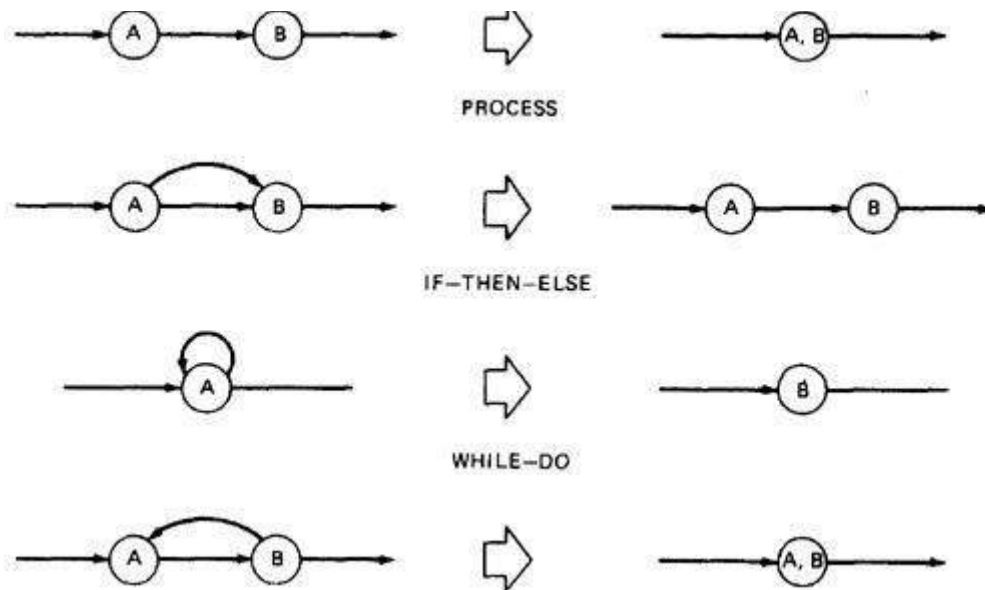


Figure 5.7: Structured Flowgraph Transformations.

The node-by-node reduction procedure can also be used as a test for structured code. Flow graphs that DO NOT contain one or more of the graphs shown below (Figure 5.8) as subgraphs are structured.

1. Jumping into loops
2. Jumping out of loops
3. Branching into decisions
4. Branching out of decisions

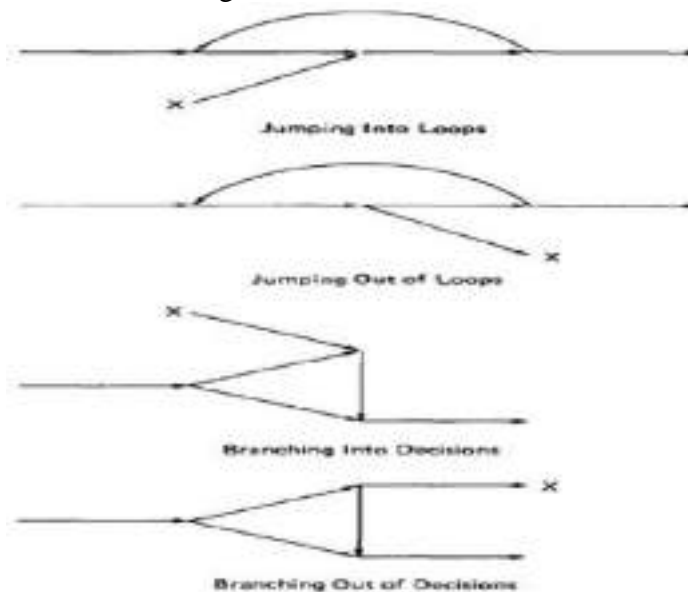


Figure 5.8: Un-structured sub-graphs.

LOWER PATH COUNT ARITHMETIC:

A lower bound on the number of paths in a routine can be approximated for structured flow graphs.

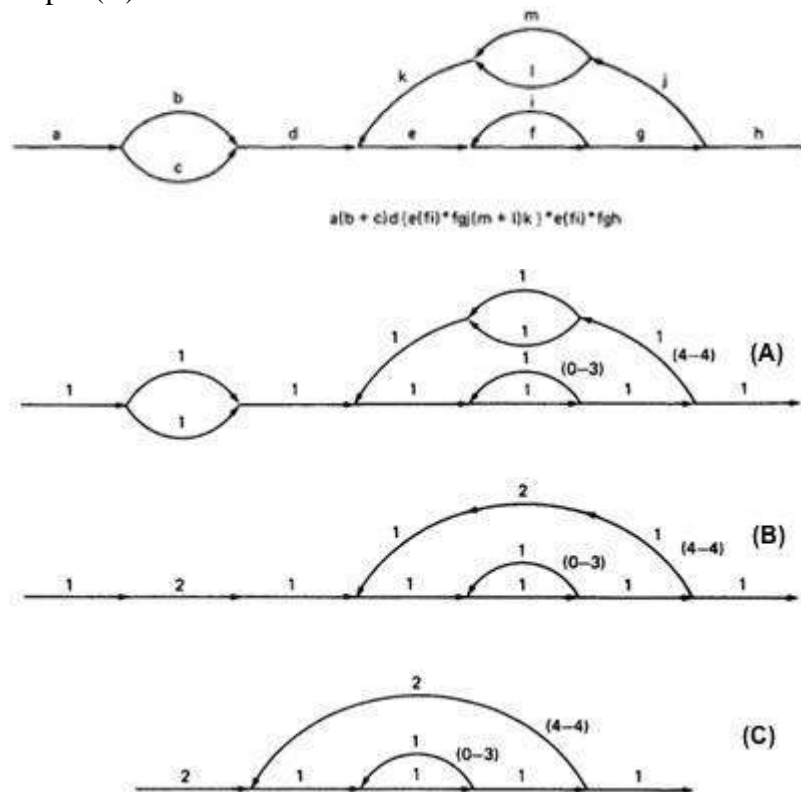
The arithmetic is as follows:

Case	Path expression	Weight expression
Parallels	$A+B$	W_A+W_B
Series	AB	$\max(W_A, W_B)$
Loop	A^n	$1, W_1$

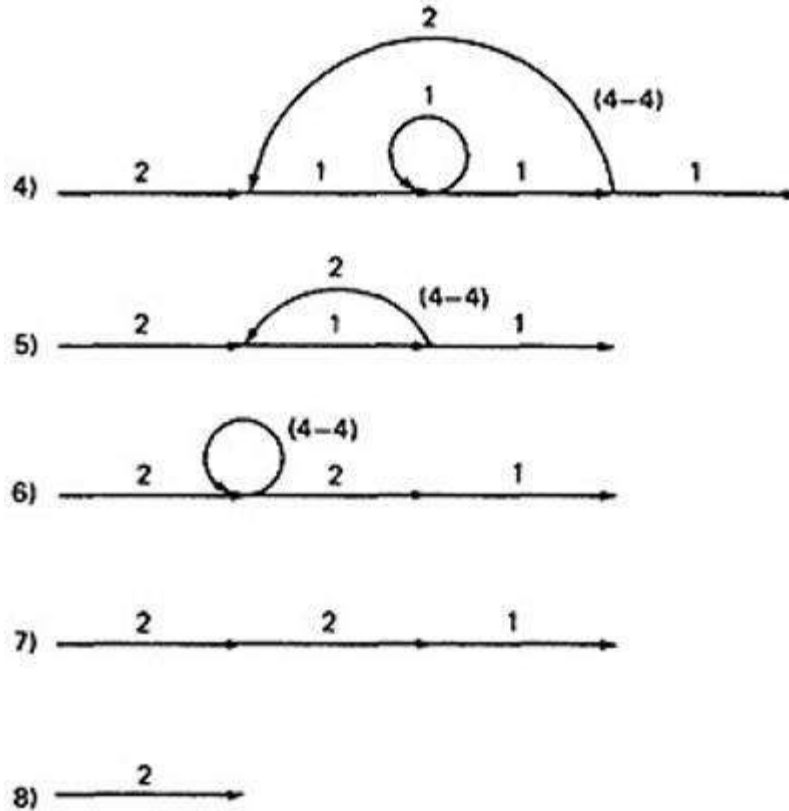
The values of the weights are the number of members in a set of paths.

EXAMPLE:

- Applying the arithmetic to the earlier example gives us the identical steps until step 3 (C) as below:



- From Step 4, the it would be different from the previous example:



- If you observe the original graph, it takes at least two paths to cover and that it can be done in two paths.
- If you have fewer paths in your test plan than this minimum you probably haven't covered. It's another check.

CALCULATING THE PROBABILITY:

Path selection should be biased toward the low - rather than the high-probability paths. This raises an interesting question:

What is the probability of being at a certain point in a routine?

This question can be answered under suitable assumptions primarily that all probabilities involved are independent, which is to say that all decisions are independent and uncorrelated. We use the same algorithm as before: node-by-node removal of uninteresting nodes.

Weights, Notations and Arithmetic:

- Probabilities can come into the act only at decisions (including decisions associated with loops).
- Annotate each outlink with a weight equal to the probability of going in that direction.
- Evidently, the sum of the outlink probabilities must equal 1
- For a simple loop, if the loop will be taken a mean of N times, the looping probability is $N/(N + 1)$ and the probability of not looping is $1/(N + 1)$.
- A link that is not part of a decision node has a probability of 1.
- The arithmetic rules are those of ordinary arithmetic.

Case	Path expression	Weight expression
Parallel	$A+B$	P_A+P_B
Series	AB	P_AP_B
Loop	A^*	$P_A / (1-P_L)$

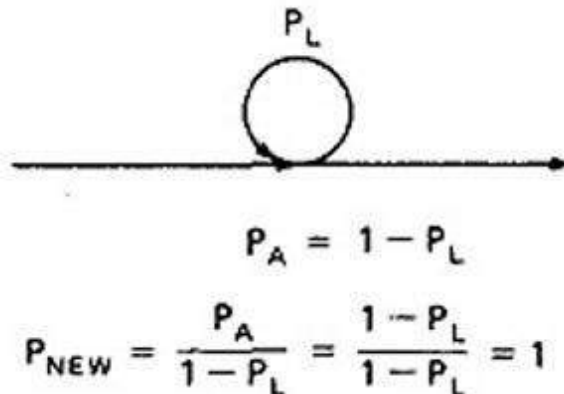
■ In this table, in case of a loop, P_A is the probability of the link leaving the loop and P_L is the probability of looping.

■ The rules are those of ordinary probability theory.

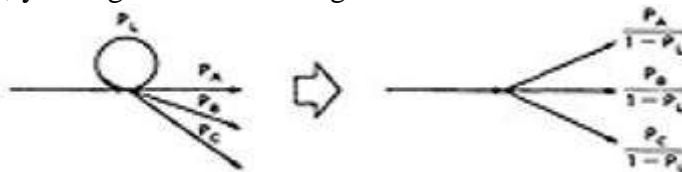
1. If you can do something either from column A with a probability of P_A or from column B with a probability P_B , then the probability that you do either is $P_A + P_B$.

2. For the series case, if you must do both things, and their probabilities are independent (as assumed), then the probability that you do both is the product of their probabilities.

For example, a loop node has a looping probability of P_L and a probability of not looping of P_A , which is obviously equal to $1 - P_L$.



Following the above rule, all we've done is replace the outgoing probability with 1 - so why the complicated rule? After a few steps in which you've removed nodes, combined parallel terms, removed loops and the like, you might find something like this:



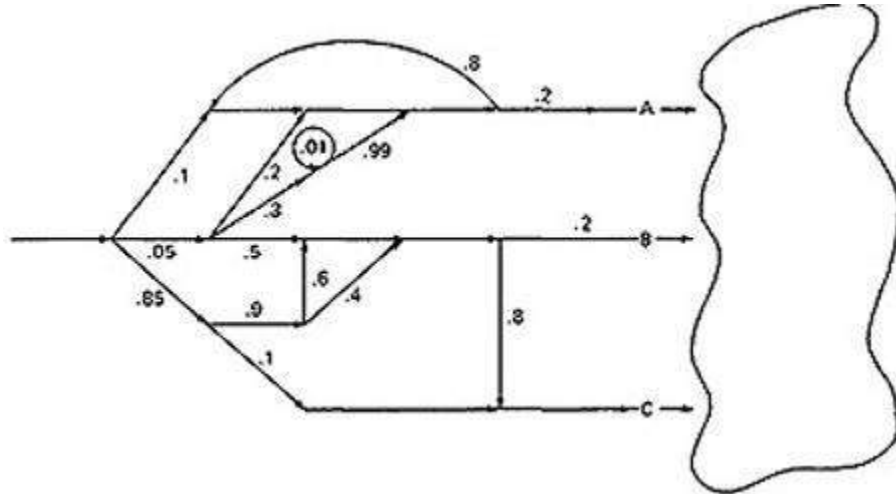
because $P_L + P_A + P_B + P_C = 1$, $1 - P_L = P_A + P_B + P_C$, and

$$\frac{P_A}{1 - P_L} + \frac{P_B}{1 - P_L} + \frac{P_C}{1 - P_L} = \frac{P_A + P_B + P_C}{1 - P_L} = 1$$

EXAMPL:

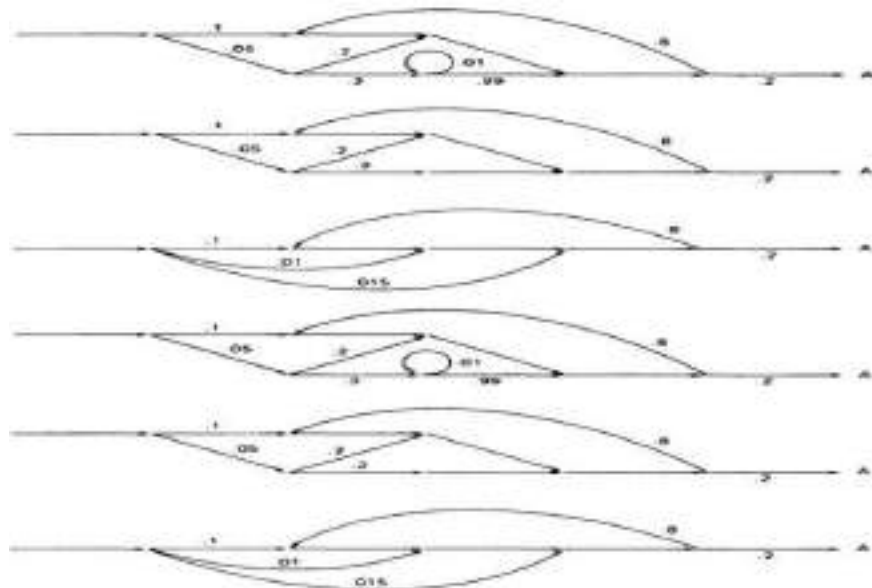
which is what we've postulated for any decision. In other words, division by $1 - P_L$ renormalizes the outlink probabilities so that their sum equals unity after the loop is removed.

- Here is a complicated bit of logic. We want to know the probability associated with cases A, B, and C.

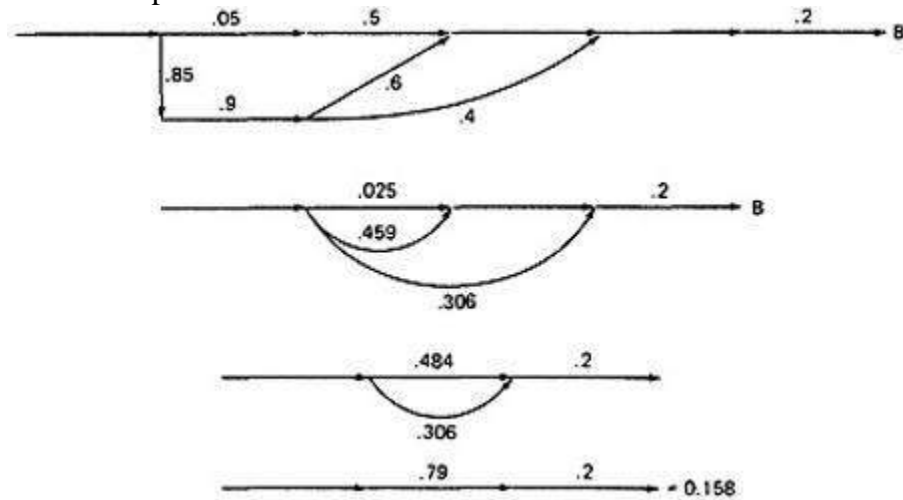


us do this in three parts, starting with case A. Note that the sum of the probabilities at each decision node is equal to 1. Start by throwing away anything that isn't on the way to case A, and then apply the reduction procedure. To avoid clutter, we usually leave out probabilities equal to 1.

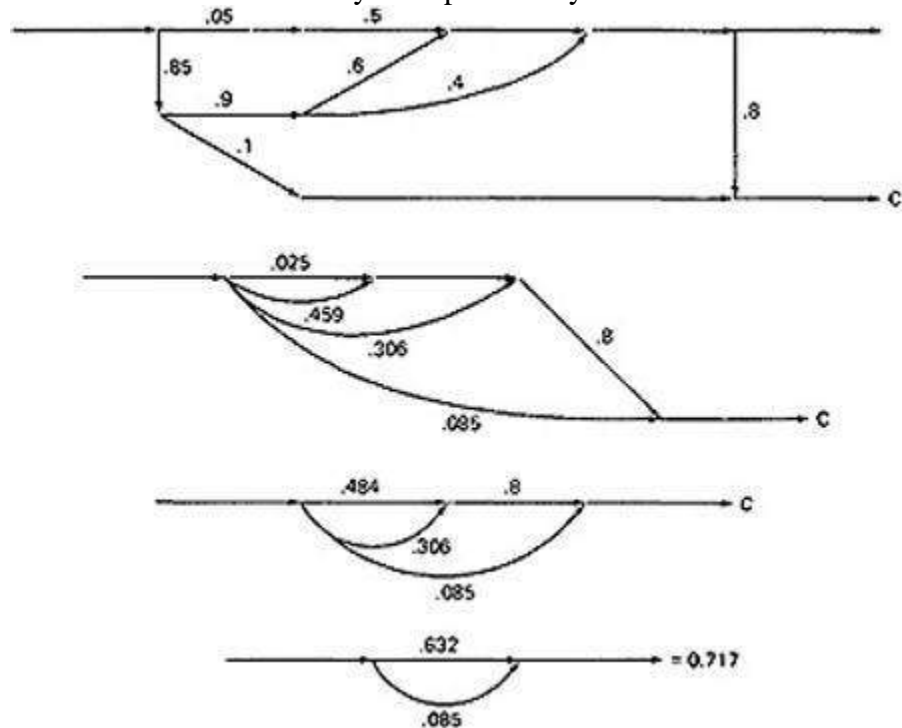
CASE A:



- Case B is simpler:



- Case C is similar and should yield a probability of $1 - 0.125 - 0.158 = 0.717$:



- These checks. It's a good idea when doing this sort of thing to calculate all the probabilities and to verify that the sum of the routine's exit probabilities does equal 1.
- If it doesn't, then you've made calculation error or, more likely, you've left out some bra How about path probabilities? That's easy. Just trace the path of interest and multiply the probabilities as you go.
- Alternatively, write down the path name and do the indicated arithmetic operation.
- Say that a path consisted of links a, b, c, d, e, and the associated probabilities were .2, .5, 1., .01, and I respectively. Path *abcbcbcddeabdddea* would have a probability of 5×10^{-10} .
- Long paths are usually improbable.

MEAN PROCESSING TIME OF A ROUTINE:

Given the execution time of all statements or instructions for every link in a flowgraph and the probability for each direction for all decisions are to find the mean processing time for the routine as a whole.

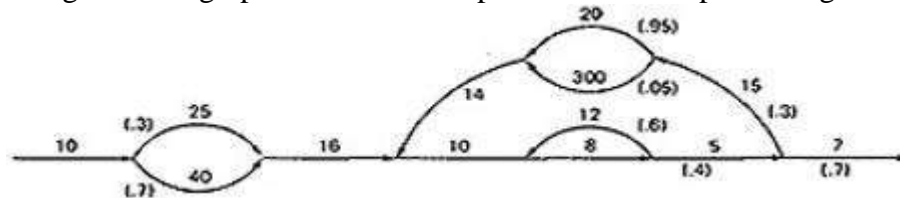
The model has two weights associated with every link: the processing time for that link, denoted by **T**, and the probability of that link **P**.

The arithmetic rules for calculating the mean time:

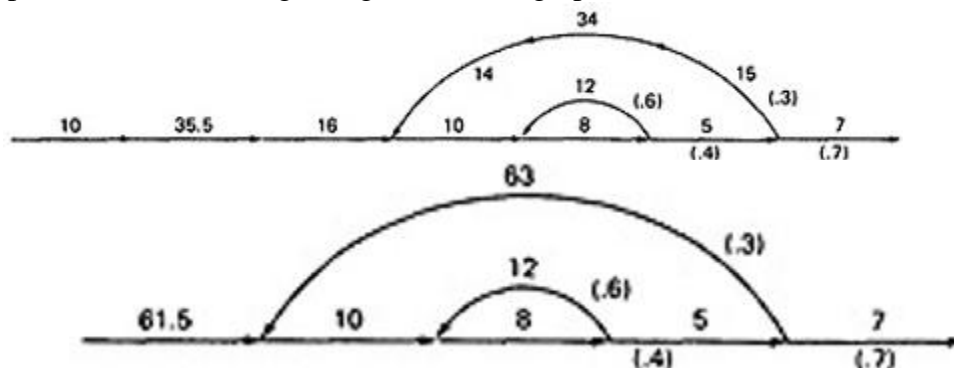
Case	Path expression	Weight expression
Parallel	$A+B$	$T_{A+B} = (P_A T_A + P_B T_B) / (P_A + P_B)$ $P_{A+B} = P_A + P_B$
Series	AB	$T_{AB} = T_A + T_B$ $P_{AB} = P_A P_B$
Loop	A^n	$T_A = T_A + T_L P_L / (1 - P_L)$ $P_A = P_A / (1 - P_L)$

EXAMPLE:

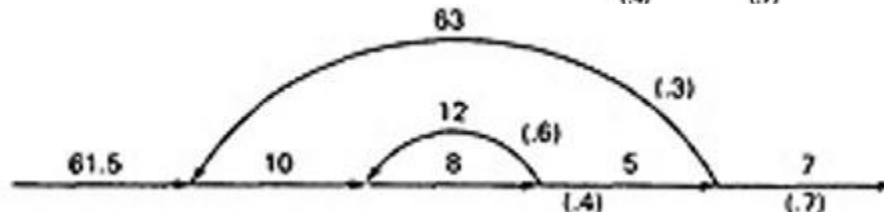
1. Start with the original flow graph annotated with probabilities and processing time.

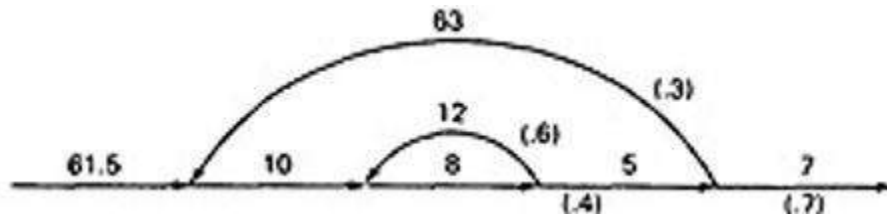


2. Combine the parallel links of the outer loop. The result is just the mean of the processing times for the links because there aren't any other links leaving the first node. Also combine the pair of links at the beginning of the flow graph.

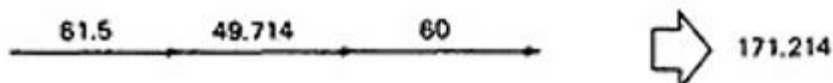
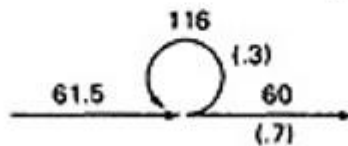
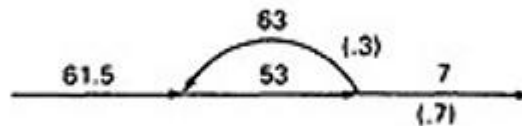
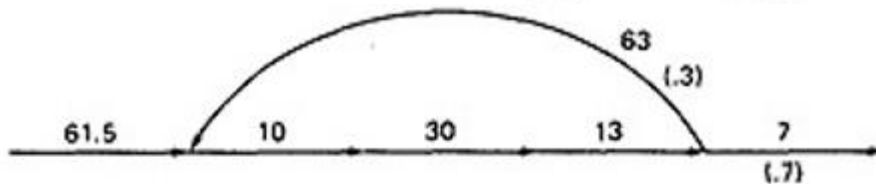
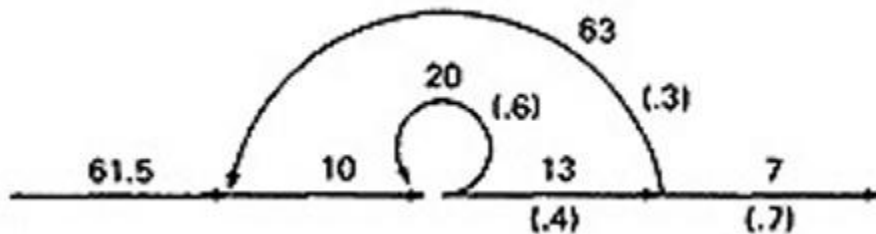


3. Combine as many serial links as you can.





4. Use the cross-term step to eliminate a node and to create the inner self - loop. 5.Finally, you can get the mean processing time, by using the arithmetic rules as follows:



PUSH/POP, GET/RETURN:

This model can be used to answer several different questions that can turn up in debugging. It can also help decide which test cases to design.

The question is:

Given a pair of complementary operations such as PUSH (the stack) and POP (the stack), considering the set of all possible paths through the routine, what is the net effect of the routine? PUSH or POP? How many times? Under what conditions?

Here are some other examples of complementary operations to which this model applies:

GET/RETURN a resource block.

OPEN/CLOSE a file.

START/STOP a device or process.

EXAMPLE 1 (PUSH / POP):

- Here is the Push/Pop Arithmetic:

Case	Path expression	Weight expression
Parallels	$A+B$	W_A+W_B
Series	AB	$W_A W_B$
Loop	A^*	W_A^*

- The numeral 1 is used to indicate that nothing of interest (neither PUSH nor POP) occurs on a given link.
- "H" denotes PUSH and "P" denotes POP. The operations are commutative, associative, and distributive.

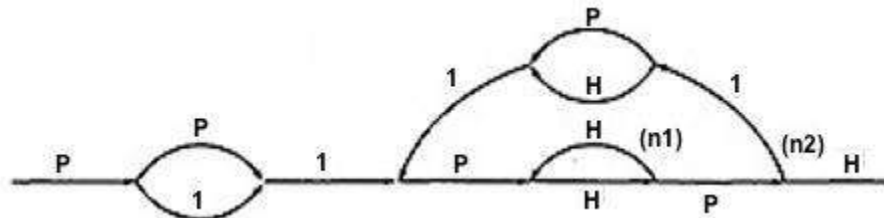
PUSH/POP MULTIPLICATION TABLE

X	H PUSH	P POP	1 NONE
H	H^2	1	H
P	1	P^2	P
1	H	P	1

PUSH/POP ADDITION TABLE

*	H PUSH	P POP	1 NONE
H	H	$P+H$	$H+1$
P	$P+H$	P	$P+1$
1	$H+1$	$P+1$	1

- Consider the following flow graph:



$$P(P+1)1\{P(HH)^{n1}HP1(P+H)1\}^{n2}P(HH)^{n1}HPH$$

▪
 simplifying by using the arithmetic tables,
 $= (P^2 + P)\{P(HH)^{n1}(P+H)\}^{n1}(HH)^{n1}$
 $= (P^2 + P)\{H^{2n1}(P^2 + 1)\}^{n2}H^{2n1}$

Si

Bel

Now Table 5.9 shows several combinations of values for the two looping terms - M1 is the number of times the inner loop will be taken and M2 the number of times the outer loop will be taken.

M_1	M_2	PUSH/POP
0	0	$P + P^2$
0	1	$P + P^2 + P^3 + P^4$
0	2	$\sum_{i=1}^6 P^i$
0	3	$\sum_{i=1}^8 P^i$
1	0	$1 + H$
1	1	$\sum_{i=0}^3 H^i$
1	2	$\sum_{i=0}^5 H^i$
1	3	$\sum_{i=0}^7 H^i$
2	0	$H^2 + H^3$
2	1	$\sum_{i=4}^7 H^i$
2	2	$\sum_{i=6}^{11} H^i$
2	3	$\sum_{i=8}^{15} H^i$

Figure 5.9: Result of the PUSH / POP Graph Analysis.

- These expressions state that the stack will be popped only if the inner loop is not taken.
- The stack will be left alone only if the inner loop is iterated once, but it may also be pushed.
- For all other values of the inner loop, the stack will only be pushed.

EXAMPLE 2 (GET / RETURN):

- Exactly the same arithmetic tables used for previous example are used for GET / RETURN a buffer block or resource, or, in fact, for any pair of

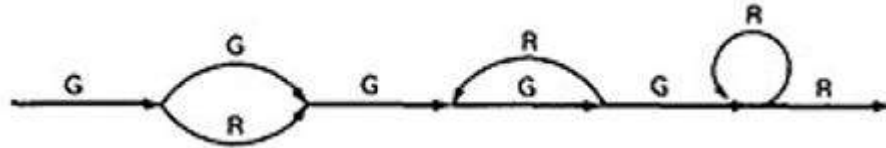
complementary operations in which the total number of operations in either direction is cumulative.

- The arithmetic tables for GET/RETURN are:

Multiplication Table				Addition Table			
X	G	R	1	+	G	R	1
G	G^2	1	G	G	G	$G+R$	$G+1$
R	1	R^2	R	R	$G+R$	R	$R+1$
1	G	R	1	1	$G+1$	$R+1$	1

"G" denotes GET and "R" denotes RETURN.

- Consider the following flowgraph:



$$\begin{aligned}
 & \cdot \quad G + R)G(GR)*GGR*R \quad G(\\
 & = G(G + R)G^3R*R \\
 & = (G + R)G^3R* \\
 & = (G^4 + G^2)R*
 \end{aligned}$$

This expression specifies the conditions under which the resources will be balanced on leaving the routine.

If the upper branch is taken at the first decision, the second loop must be taken four times.

If the lower branch is taken at the first decision, the second loop must be taken twice.

For any other values, the routine will not balance. Therefore, the first loop does not have to be instrumented to verify this behavior because its impact should be nil.

LIMITATIONS AND SOLUTIONS:

- The main limitation to these applications is the problem of unachievable paths.
- The node-by-node reduction procedure, and most graph-theory-based algorithms work well when all paths are possible, but may provide misleading results when some paths are unachievable.
- The approach to handling unachievable paths (for any application) is to partition the graph into subgraphs so that all paths in each of the subgraphs are achievable.
- The resulting subgraphs may overlap, because one path may be common to several different subgraphs.
- Each predicate's truth-functional value potentially splits the graph into two subgraphs. For n predicates, there could be as many as 2^n subgraphs.

REGULAR EXPRESSIONS AND FLOW ANOMALY DETECTION:

- **THE PROBLEM:**

- The generic flow-anomaly detection problem (note: not just data-flow anomalies, but any flow anomaly) is that of looking for a specific sequence of options considering all possible paths through a routine.
- Let the operations be SET and RESET, denoted by *s* and *r* respectively, and we want to know if there is a SET followed immediately a SET or a RESET followed immediately by a RESET (an *ss* or an *rr* sequence).
- Some more application examples:
 1. A file can be opened (*o*), closed (*c*), read (*r*), or written (*w*). If the file is read or written to after it's been closed, the sequence is nonsensical. Therefore, *cr* and *cw* are anomalous. Similarly, if the file is read before it's been written, just after opening, we may have a bug. Therefore, *or* is also anomalous. Furthermore, *oo* and *cc*, though not actual bugs, are a waste of time and therefore should also be examined.
 2. A tape transport can do a rewind (*d*), fast-forward (*f*), read (*r*), write (*w*), stop (*p*), and skip (*k*). There are rules concerning the use of the transport; for example, you cannot go from rewind to fast-forward without an intervening stop or from rewind or fast-forward to read or write without an intervening stop. The following sequences are anomalous: *df*, *dr*, *dw*, *fd*, and *fr*. Does the flowgraph lead to anomalous sequences on any path? If so, what sequences and under what circumstances?
 3. The data-flow anomalies discussed in Unit 4 requires us to detect the *dd*, *dk*, *kk*, and *ku* sequences. Are there paths with anomalous data flows?

- **THE METHOD:**

- Annotate each link in the graph with the appropriate operator or the null operator 1.
- Simplify things to the extent possible, using the fact that $a + a = a$ and $12 = 1$.
- You now have a regular expression that denotes all the possible sequences of operators in that graph. You can now examine that regular expression for the sequences of interest.
- **EXAMPLE:** Let *A*, *B*, *C*, be nonempty sets of character sequences whose smallest string is at least one character long. Let *T* be a two-character string of characters. Then if *T* is a substring of (i.e., if *T* appears within) AB^nC , then *T* will appear in AB^2C . (**HUANG's Theorem**)
As an example, let
- $A = pp \ B = srr \ C = rp \ T = ss$

The theorem states that *ss* will appear in $pp(srr)^n rp$ if it appears in $pp(srr)^2 rp$.

- However, let

$$A = p + pp + ps$$

$$B = psr + ps(r + ps) \quad C = rp$$

$$T = p^4$$

Is it obvious that there is a p^4 sequence in AB^nC ? The theorem states that we have only to look at

$$(p + pp + ps)[psr + ps(r + ps)]^2rp$$

Multiplying out the expression and simplifying shows that there is no p^4 Sequence.

- Incidentally, the above observation is an informal proof of the wisdom of looping twice discussed in Unit 2. Because data-flow anomalies are represented by two- character sequences, it follows the above theorem that looping twice is what you need to do to find such anomalies.

- **LIMITATIONS:**

- Huang's theorem can be easily generalized to cover sequences of greater length than two characters. Beyond three characters, though, things get complex and this method has probably reached its utilitarian limit for manual application.
- There are some nice theorems for finding sequences that occur at the beginnings and ends of strings but no nice algorithms for finding strings buried in an expression.
- Static flow analysis methods can't determine whether a path is or is not achievable. Unless the flow analysis includes symbolic execution or similar techniques, the impact of unachievable paths will not be included in the analysis.

The flow-anomaly application, for example, doesn't tell us that there will be a flow anomaly - it tells us that if the path is achievable, then there will be a flow anomaly. Such analytical problems go away, of course, if you take the trouble to design routines for which all paths are achievable.

LOGIC BASED TESTING

OVERVIEW OF LOGIC BASED TESTING:

- **INTRODUCTION:**

- The functional requirements of many programs can be specified by **decision tables**, which provide a useful basis for program and test design.
- Consistency and completeness can be analyzed by using boolean algebra, which can also be used as a basis for test design. Boolean algebra is trivialized by using **Karnaugh-Veitch charts**.
- "Logic" is one of the most often used words in programmers' vocabularies but one of their least used techniques.
- Boolean algebra is to logic as arithmetic is to mathematics. Without it, the tester or programmer is cut off from many test and design techniques and tools that incorporate those techniques.
- Logic has been, for several decades, the primary tool of hardware logic designers.
- Many test methods developed for hardware logic can be adapted to software logic testing. Because hardware testing automation is 10 to 15 years ahead of software testing automation, hardware testing methods and its associated theory is a fertile ground for software testing methods.
- As programming and test techniques have improved, the bugs have shifted closer to the process front end, to requirements and their specifications. These bugs range from 8% to 30% of the total and because they're first-in and last-out, they're the costliest of all.
- The trouble with specifications is that they're hard to express.
- Boolean algebra (also known as the sentential calculus) is the most basic of all logic systems.
- Higher-order logic systems are needed and used for formal specifications.
- Much of logical analysis can be and is embedded in tools. But these tools incorporate methods to simplify, transform, and check specifications, and the methods are to a large extent based on boolean algebra.

- **KNOWLEDGE BASED SYSTEM:**

- The **knowledge-based system** (also expert system, or "artificial intelligence" system) has become the programming construct of choice for many applications that were once considered very difficult.
- Knowledge-based systems incorporate knowledge from a knowledge domain such as medicine, law, or civil engineering into a database. The data can then be queried and interacted with to provide solutions to problems in that domain.
- One implementation of knowledge-based systems is to incorporate the expert's knowledge into a set of rules. The user can then provide data and ask questions based on that data.
- The user's data is processed through the rule base to yield conclusions (tentative or definite) and requests for more data. The processing is done by a program called the **inference engine**.
- Understanding knowledge-based systems and their validation problems requires an understanding of formal logic.
 - Decision tables are extensively used in business data processing; Decision-table preprocessors as extensions to COBOL are in common use; boolean algebra is embedded in the implementation of these processors.
 - Although programmed tools are nice to have, most of the benefits of boolean algebra can be reaped by wholly manual means if you have the right conceptual tool: the Karnaugh-Veitch diagram is that conceptual tool.

DECISION TABLES:

- Figure 6.1 is a limited - entry decision table. It consists of four areas called the condition stub, the condition entry, the action stub, and the action entry.
- Each column of the table is a rule that specifies the conditions under which the actions named in the action stub will take place.
- The condition stub is a list of names of conditions.

		CONDITION ENTRY			
CONDITION STUB		RULE 1	RULE 2	RULE 3	RULE 4
	CONDITION 1	YES	YES	NO	NO
	CONDITION 2	YES	I	NO	I
	CONDITION 3	NO	YES	NO	I
ACTION STUB	ACTION 1	YES	YES	NO	NO
	ACTION 2	NO	NO	YES	NO
	ACTION 3	NO	NO	NO	YES
		ACTION ENTRY			

Figure 6.1 : Examples of Decision Table.

- A more general decision table can be as below:

		Printer troubleshooter							
		Rules							
Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognised	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

Figure 6.2 : Another Examples of Decision Table.

- A rule specifies whether a condition should or should not be met for the rule to be satisfied. "YES" means that the condition must be met, "NO" means that the condition must not be met, and "I" means that the condition plays no part in the rule, or it is immaterial to that rule.

The action stub names the actions the routine will take or initiate if the rule is satisfied.

- If the action entry is "YES", the action will take place; if "NO", the action will not take place.

The table in Figure 6.1 can be translated as follows:

Action 1 will take place if conditions 1 and 2 are met and if conditions 3 and 4 are not met (rule 1) or if conditions 1, 3, and 4 are met (rule 2).

- "Condition" is another word for predicate.
- Decision-table uses "condition" and "satisfied" or "met". Let us use "predicate" and TRUE / FALSE.
- Now the above translations become:
 1. Action 1 will be taken if predicates 1 and 2 are true and if predicates 3 and 4 are false (rule 1), or if predicates 1, 3, and 4 are true (rule 2).
 2. Action 2 will be taken if the predicates are all false, (rule 3).
 3. Action 3 will take place if predicate 1 is false and predicate 4 is true (rule 4).
- In addition to the stated rules, we also need a **Default Rule** that specifies the default action to be taken when all other rules fail. The default rules for Table in Figure 6.1 is shown in Figure 6.3

	Rule 5	Rule 6	Rule 7	Rule 8
CONDITION 1	I	NO	YES	YES
CONDITION 2	I	YES	I	NO
CONDITION 3	YES	I	NO	NO
CONDITION 4	NO	NO	YES	I
DEFAULT ACTION	YES	YES	YES	YES

Figure 6.3 : The default rules of Table in Figure 6.1

• DECISION-TABLE PROCESSORS:

- Decision tables can be automatically translated into code and, as such, are a higher-order language
- If the rule is satisfied, the corresponding action takes place
- Otherwise, rule 2 is tried. This process continues until either a satisfied rule results in an action or no rule is satisfied and the default action is taken
- Decision tables have become a useful tool in the programmers kit, in business data processing.

DECISION-TABLES AS BASIS FOR TEST CASE DESIGN:

1. The specification is given as a decision table or can be easily converted into one.
2. The order in which the predicates are evaluated does not affect interpretation of the rules or the resulting action - i.e., an arbitrary permutation of the predicate order will not, or should not, affect which action takes place.
3. The order in which the rules are evaluated does not affect the resulting action - i.e., an arbitrary permutation of rules will not, or should not, affect which action takes place.
4. Once a rule is satisfied and an action selected, no other rule need be examined.
5. If several actions can result from satisfying a rule, the order in which the actions are executed doesn't matter.

DECISION-TABLES AND STRUCTURE:

- Decision tables can also be used to examine a program's structure.
- Figure 6.4 shows a program segment that consists of a decision tree.
- These decisions, in various combinations, can lead to actions 1, 2, or 3.

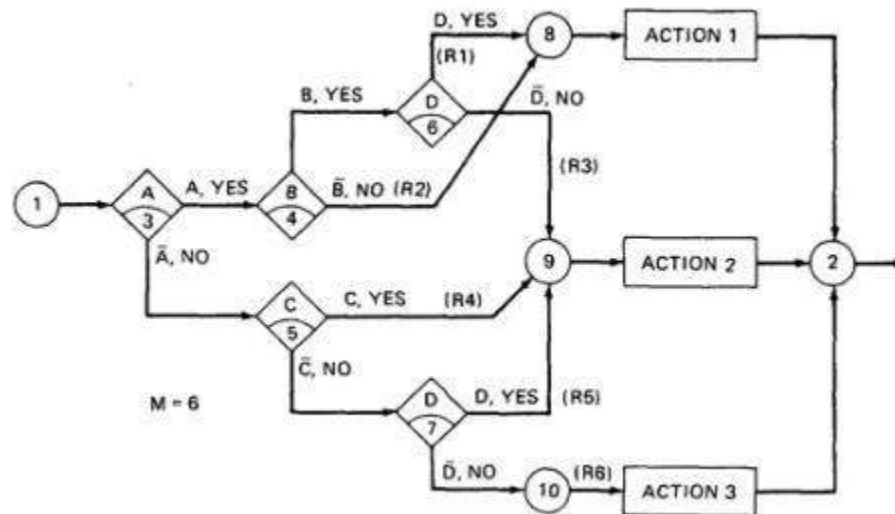


Figure 6.4 : A Sample Program

- If the decision appears on a path, put in a YES or NO as appropriate. If the decision does not appear on the path, put in an I, Rule 1 does not contain decision C, therefore its entries are: YES, YES, I, YES.
- The corresponding decision table is shown in Table 6.1

	RULE 1	RULE 2	RULE 3	RULE 4	RULE 5	RULE 6
CONDITION A						
CONDITION B	YES	YES	YES	NO	NO	NO
CONDITION C	YES	NO	YES	YES	NO	NO
CONDITION D	YES	I	NO		YES	NO
ACTION 1	YES	YES	NO	NO	NO	NO
ACTION 2	NO	NO	YES	YES	YES	NO
ACTION 3	NO	NO	NO	NO	NO	YES

Table 6.1: Decision Table corresponding to Figure 6.4

As an example, expanding the immaterial cases results as below:

	RULE 1	RULE 2		RULE 1.1	RULE 1.2	RULE 2.1	RULE 2.2
CONDITION 1	YES	YES		YES	YES	YES	YES
CONDITION 2	I	NO		YES	NO	NO	NO
CONDITION 3	YES	I		YES	YES	YES	NO
CONDITION 4	NO	NO		NO	NO	NO	NO
ACTION 1	YES	NO		YES	YES	NO	NO
ACTION 2	NO	YES		NO	NO	YES	YES

Table 6.2: Expansion of Table 6.1

- Similarly, If we expand the immaterial cases for the above Table 6.1, it results in Table 6.2 as below:

	R 1	RULE 2	R 3	RULE 4	R 5	R 6
CONDITION A	YY	YYYY	YY	NNNN	NN	NN
CONDITION B	YY	NNNN	YY	YYNN	NY	YN
CONDITION C	YN	NNYY	YN	YYYY	NN	NN
CONDITION D	YY	YNNY	NN	NYYN	YY	NN

1. Sixteen cases are represented in Table 6.1, and no case appears twice.
2. Consequently, the flowgraph appears to be complete and consistent.
3. As a first check, before you look for all sixteen combinations, count the number of Y's and N's in each row. They should be equal. We can find the bug that way.

• ANOTHER EXAMPLE - A TROUBLE SOME PROGRAM:

1. Consider the following specification whose putative flowgraph is shown in Figure 6.5:
 1. If condition A is met, do process A1 no matter what other actions are taken or what other conditions are met.

2. If condition B is met, do process A2 no matter what other actions are taken or what other conditions are met.
 3. If condition C is met, do process A3 no matter what other actions are taken or what other conditions are met.
 4. If none of the conditions is met, then do processes A1, A2, and A3.
 5. When more than one process is done, process A1 must be done first, then A2, and then A3. The only permissible cases are: (A1), (A2), (A3), (A1,A3), (A2,A3) and (A1,A2,A3).
2. Figure 6.5 shows a sample program with a bug.

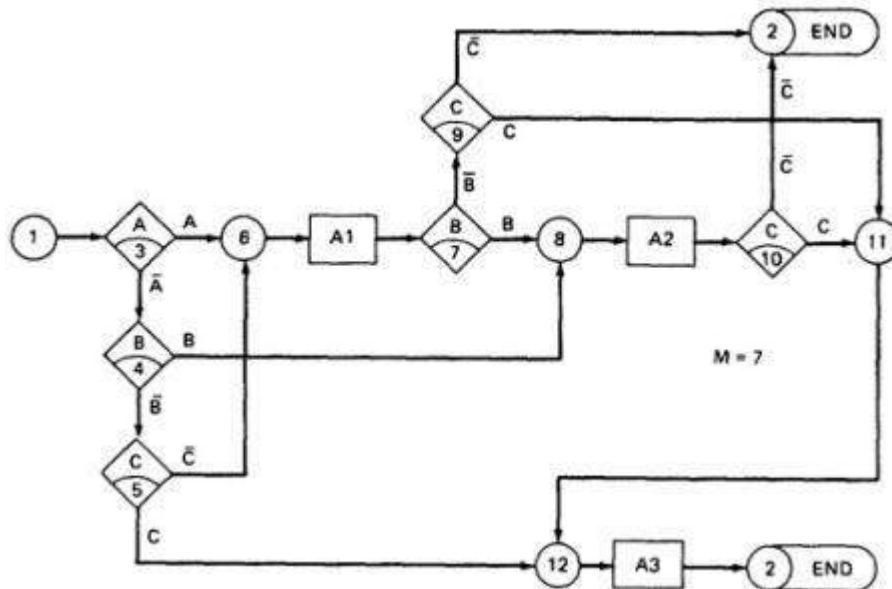


Figure 6.5 : A Troublesome Program

- The programmer tried to force all three processes to be executed for the $\bar{A}\bar{B}\bar{C}$ cases but forgot that the B and C predicates would be done again, thereby bypassing processes A2 and A3.
- Table 6.3 shows the conversion of this flow graph into a decision table after expansion.

RULES								
	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$	$\bar{A}B\bar{C}$	$\bar{A}BC$	$A\bar{B}\bar{C}$	$A\bar{B}C$	$AB\bar{C}$	ABC
CONDITION A	NO	NO	NO	NO	YES	YES	YES	YES
CONDITION B	NO	NO	YES	YES	YES	YES	NO	NO
CONDITION C	NO	YES	YES	NO	NO	YES	YES	NO
ACTION 1	YES	NO	NO	NO	YES	YES	YES	YES
ACTION 2	YES	NO	YES	YES	YES	YES	NO	NO
ACTION 3	YES	YES	YES	NO	NO	YES	YES	NO

Table 6.3: Decision Table for Figure 6.5

PATH EXPRESSIONS:

□ GENERAL:

- Logic-based testing is structural testing when it's applied to structure (e.g., control flow graph of an implementation); it's functional testing when it's applied to a specification.
- In logic-based testing we focus on the truth values of control flow predicates.

- A **predicate** is implemented as a process whose outcome is a truth-functional value.
- For our purpose, logic-based testing is restricted to binary predicates.
- We start by generating path expressions by path tracing as in Unit V, but this time, our purpose is to convert the path expressions into boolean algebra, using the predicates' truth values (e.g., A and \bar{A}) as weights.

□ **BOOLEAN ALGEBRA:**

○ **STEPS:**

1. Label each decision with an uppercase letter that represents the truth value of the predicate. The YES or TRUE branch is labeled with a letter (say A) and the NO or FALSE branch with the same letter overscored (say \bar{A}).
2. The truth value of a path is the product of the individual labels. Concatenation or products mean "AND". For example, the straight-through path of Figure 6.5, which goes via nodes 3, 6, 7, 8, 10, 11, 12, and 2, has a truth value of ABC. The path via nodes 3, 6, 7, 9 and 2 has a value of $AB\bar{C}$.
3. If two or more paths merge at a node, the fact is expressed by use of a plus sign (+) which means "OR".

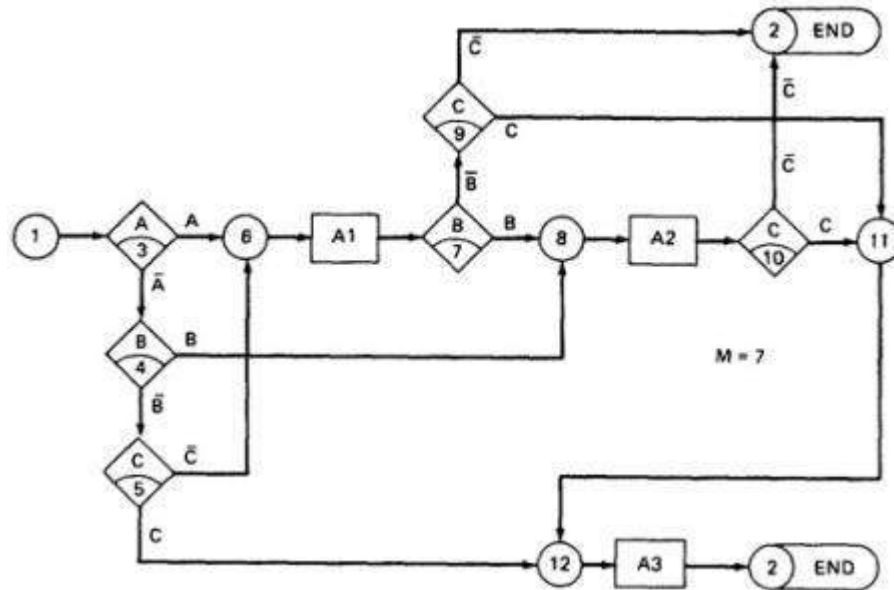


Figure 6.5: A Troublesome Program

- Using this convention, the truth-functional values for several of the nodes can be expressed in terms of segments from previous nodes. Use the node name to identify the point.

$$\begin{aligned}
 N6 &= A + \bar{A}\bar{B}\bar{C} \\
 N8 &= (N6)B + \bar{A}\bar{B} = AB + \bar{A}\bar{B}\bar{C}B + \bar{A}\bar{B} \\
 N11 &= (N8)C + (N6)\bar{B}C \\
 N12 &= N11 + \bar{A}\bar{B}C \\
 N2 &= N12 + (N8)\bar{C} + (N6)\bar{B}\bar{C}
 \end{aligned}$$

- There are only two numbers in boolean algebra: zero (0) and one (1). One means "always true" and zero means "always false".

○ **RULES OF BOOLEAN ALGEBRA:**

- Boolean algebra has three operators: X (AND), + (OR) and \bar{A} (NOT)
- **X** : meaning AND. Also called multiplication. A statement such as AB (A X B) means "A and B are both true". This symbol is usually left out as in ordinary algebra.
- **+** : meaning OR. "A + B" means "either A is true or B is true or both".
- \bar{A} meaning NOT. Also negation or complementation. This is read as either "not A" or "A bar". The entire expression under the bar is negated.
- The following are the laws of boolean algebra:

1. $\frac{A + A}{\bar{A} + \bar{A}}$	$= \frac{A}{\bar{A}}$	If something is true, saying it twice doesn't make it truer, ditto for falsehoods.
2. $A + 1$	$= 1$	If something is always true, then "either A or true or both" must also be universally true.
3. $A + 0$	$= A$	
4. $A + B$	$= B + A$	Commutative law.
5. $A + \bar{A}$	$= 1$	If either A is true or not-A is true, then the statement is always true.
6. $\frac{AA}{\bar{A}\bar{A}}$	$= \frac{A}{\bar{A}}$	
7. $A \times 1$	$= A$	
8. $A \times 0$	$= 0$	
9. AB	$= BA$	
10. $A\bar{A}$	$= 0$	A statement can't be simultaneously true and false.
11. $\bar{\bar{A}}$	$= A$	"You ain't not going" means you are. How about, "I ain't not never going to get this nohow."?
12. $\bar{0}$	$= 1$	
13. $\bar{1}$	$= 0$	
14. $\overline{A + B}$	$= \bar{A}\bar{B}$	Called "De Morgan's theorem or law."
15. \overline{AB}	$= \bar{A} + \bar{B}$	
16. $A(B + C)$	$= AB + AC$	Distributive law.
17. $(AB)C$	$= A(BC)$	Multiplication is associative.
18. $(A + B) + C$	$= A + (B + C)$	So is addition.
19. $A + \bar{A}B$	$= A + B$	Absorptive law.
20. $A + AB$	$= A$	

In all of the above, a letter can represent a single sentence or an entire boolean algebra expression.

Individual letters in a boolean algebra expression are called **Literals** (e.g. A,B) The product of several literals is called a **product term** (e.g., ABC, DE).

An arbitrary boolean expression that has been multiplied out so that it consists of the sum of products (e.g., ABC + DEF + GH) is said to be in **sum-of-products form**.

The result of simplifications (using the rules above) is again in the sum of product form and each product term in such a simplified version is called a **prime implicant**. For example, ABC + AB + DEF reduce by rule 20 to AB + DEF; that is, AB and DEF are prime implicants. The path expressions of Figure 6.5 can now be simplified by applying the rules.

The following are the laws of boolean algebra:

$$\begin{aligned}
N6 &= A + \overline{A} \overline{B} \overline{C} \\
&= A + \overline{B} \overline{C} && : \text{Use rule 19, with "B" = } \overline{B} \overline{C}. \\
N8 &= (N6)B + \overline{A} B \\
&= (A + \overline{B} \overline{C})B + \overline{A} B && : \text{Substitution.} \\
&= AB + \overline{B} \overline{C} B + \overline{A} B && : \text{Rule 16 (distributive law).} \\
&= AB + \overline{B} \overline{C} B + \overline{A} B && : \text{Rule 9 (commutative multiplication).} \\
&= AB + 0C + \overline{A} B && : \text{Rule 10.} \\
&= AB + 0 + \overline{A} B && : \text{Rule 8.} \\
&= AB + \overline{A} B && : \text{Rule 3.} \\
&= (A + \overline{A})B && : \text{Rule 16 (distributive law).} \\
&= 1 \times B && : \text{Rule 5.} \\
&= B && : \text{Rules 7, 9.}
\end{aligned}$$

Similarly,

$$\begin{aligned}
N11 &= (N8)C + (N6)\overline{B} \overline{C} \\
&= BC + (A + \overline{B} \overline{C})\overline{B} \overline{C} && : \text{Substitution.} \\
&= BC + A\overline{B} \overline{C} && : \text{Rules 16, 9, 10, 8, 3.} \\
&= C(B + \overline{B} \overline{A}) && : \text{Rules 9, 16.} \\
&= C(B + A) && : \text{Rule 19.} \\
&= AC + BC && : \text{Rules 16, 9, 9, 4.} \\
N12 &= N11 + \overline{A} \overline{B} \overline{C} \\
&= AC + BC + \overline{A} \overline{B} \overline{C} \\
&= C(B + \overline{A} \overline{B}) + AC \\
&= C(\overline{A} + B) + AC \\
&= C\overline{A} + AC + BC \\
&= C + BC \\
&= C \\
N2 &= N12 + (N8)\overline{C} + (N6)\overline{B} \overline{C} \\
&= C + \overline{B} \overline{C} + (A + \overline{B} \overline{C})\overline{B} \overline{C} \\
&= C + \overline{B} \overline{C} + \overline{B} \overline{C} \\
&= C + \overline{C}(B + \overline{B}) \\
&= C + \overline{C} \\
&= 1
\end{aligned}$$

The deviation from the specification is now clear. The functions should have been:

$$\begin{aligned}
N6 &= A + \overline{A} \overline{B} \overline{C} = A + \overline{B} \overline{C} && : \text{correct.} \\
N8 &= B + \overline{A} \overline{B} \overline{C} = B + \overline{A} \overline{C} && : \text{wrong, was just B.} \\
N12 &= C + \overline{A} \overline{B} \overline{C} = C + \overline{A} \overline{B} && : \text{wrong, was just C.}
\end{aligned}$$

Loops complicate things because we may have to solve a boolean equation to determine what predicate value combinations lead to where.

KV CHARTS:

□ INTRODUCTION:

- If you had to deal with expressions in four, five, or six variables, you could get bogged down in the algebra and make as many errors in designing test cases as there are bugs in the routine you're testing.
- **Karnaugh-Veitch chart** reduces boolean algebraic manipulations to graphical trivia.
- Beyond six variables these diagrams get cumbersome and may not be effective.

□ SINGLE VARIABLE:

- Figure 6.6 shows all the boolean functions of a single variable and their equivalent representation as a KV chart.

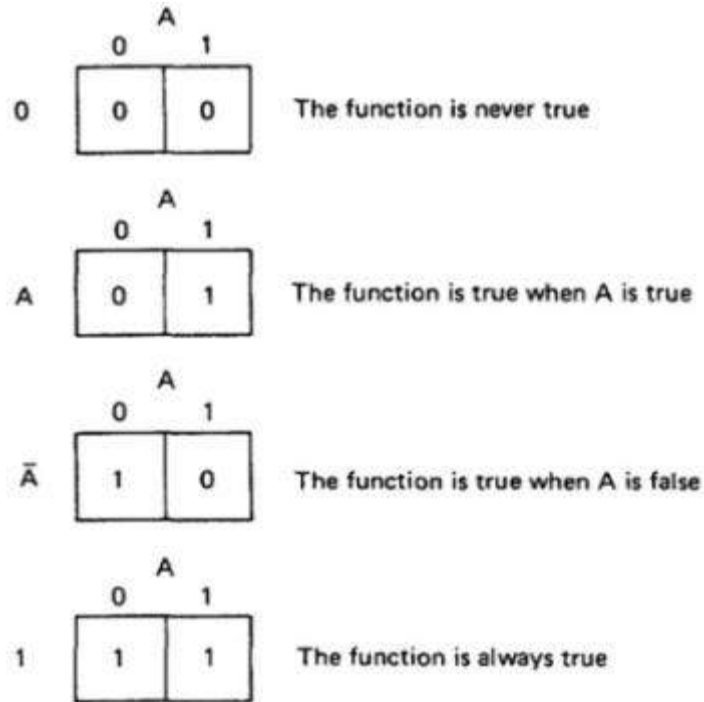


Figure 6.6 : KV Charts for Functions of a Single Variable.

- The charts show all possible truth values that the variable A can have.
 - A "1" means the variable's value is "1" or TRUE. A "0" means that the variable's value is 0 or FALSE.
 - The entry in the box (0 or 1) specifies whether the function that the chart represents is true or false for that value of the variable.
 - We usually do not explicitly put in 0 entries but specify only the conditions under which the function is true.
- **TWO VARIABLES:**
- Figure 6.7 shows eight of the sixteen possible functions of two variables.

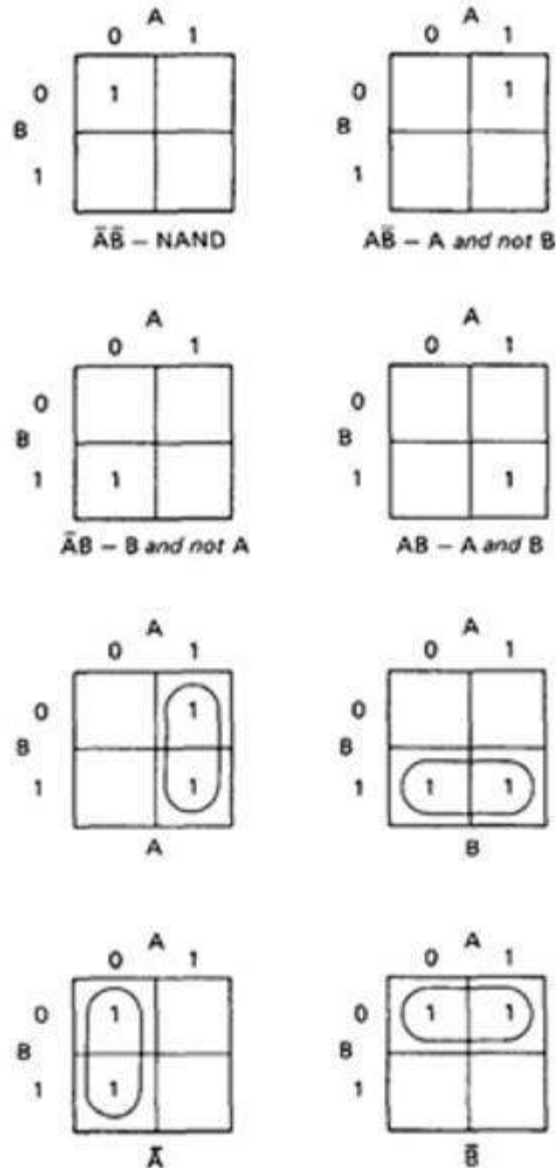


Figure 6.7: KV Charts for Functions of Two Variables.

- Each box corresponds to the combination of values of the variables for the row and column of that box.
- A pair may be adjacent either horizontally or vertically but not diagonally.
- Any variable that changes in either the horizontal or vertical direction does not appear in the expression.
- In the fifth chart, the B variable changes from 0 to 1 going down the column, and because the A variable's value for the column is 1, the chart is equivalent to a simple A.
- Figure 6.8 shows the remaining eight functions of two variables.

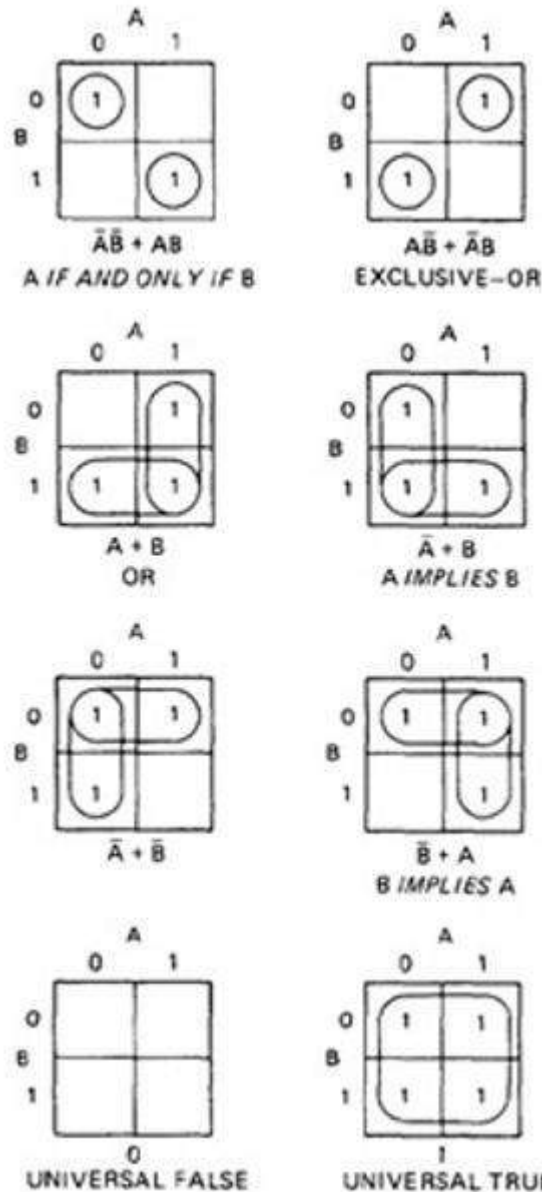
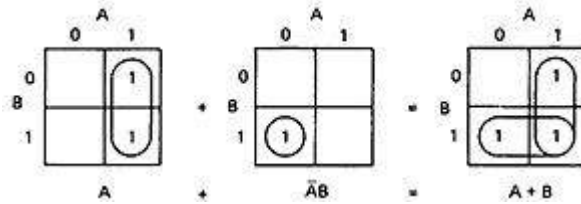


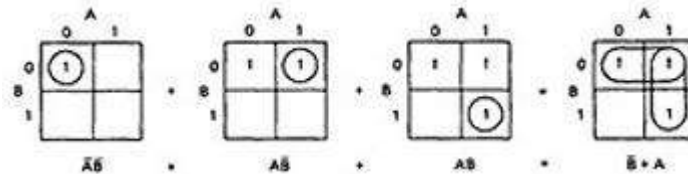
Figure 6.8: More Functions of Two Variables.

- The first chart has two 1's in it, but because they are not adjacent, each must be taken separately.
- They are written using a plus sign.
- It is clear now why there are sixteen functions of two variables.
- Each box in the KV chart corresponds to a combination of the variables' values.
- That combination might or might not be in the function (i.e., the box corresponding to that combination might have a 1 or 0 entry).
- Since n variables lead to 2^n combinations of 0 and 1 for the variables, and each such combination (box) can be filled or not filled, leading to 2^{2^n} ways of doing this.
- Consequently for one variable there are $2^{2^1} = 4$ functions, 16 functions of 2 variables, 256 functions of 3 variables, 16,384 functions of 4 variables, and so on.

- Given two charts over the same variables, arranged the same way, their product is the term by term product, their sum is the term by term sum, and the negation of a chart is gotten by reversing all the 0 and 1 entries in the chart.

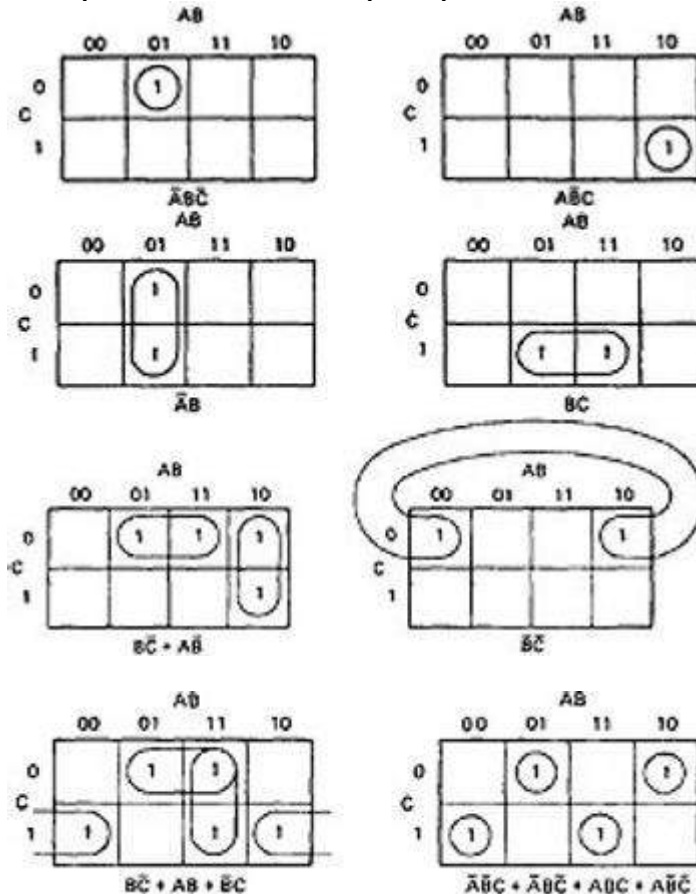


OR



THREE VARIABLES:

- KV charts for three variables are shown below.
- As before, each box represents an elementary term of three variables with a bar appearing or not appearing according to whether the row-column heading for that box is 0 or 1.
- A three-variable chart can have groupings of 1, 2, 4, and 8 boxes.
- A few examples will illustrate the principles:



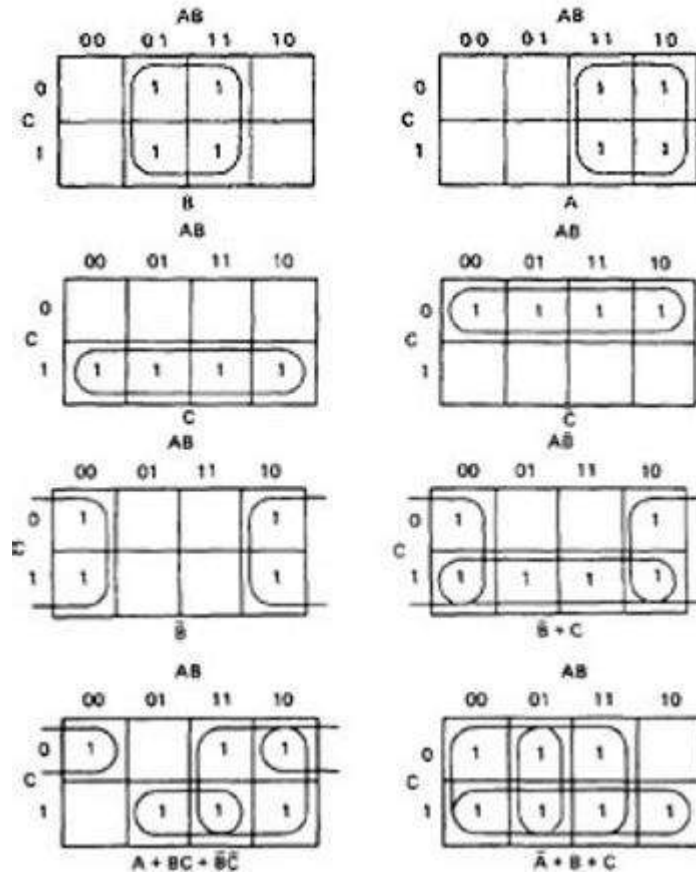


Figure 6.8: KV Charts for Functions of Three Variables.

- You'll notice that there are several ways to circle the boxes into maximum-sized covering groups.

UNIT-V

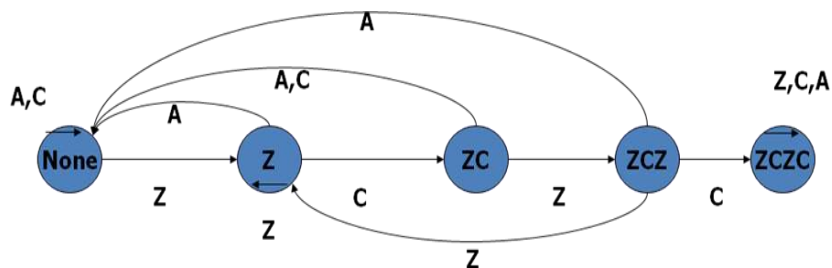
STATES, STATE GRAPHS, AND TRANSITION TESTING

Introduction

- ☐ The finite state machine is as fundamental to software engineering as boolean algebra to logic.
- ☐ State testing strategies are based on the use of finite state machine models for software structure, software behavior, or specifications of software behavior.
- ☐ Finite state machines can also be implemented as table-driven software, in which case they are a powerful design option.

State Graphs

- A state is defined as: “A combination of circumstances or attributes belonging for the time being to a person or thing.”
- ☐ For example, a moving automobile whose engine is running can have the following states with respect to its transmission.
 - Reverse gear
 - Neutral gear
 - First gear
 - Second gear
 - Third gear
 - Fourth gear
- For example, a program that detects the character sequence “ZCZC” can be in the following states.
 - ☐ Neither ZCZC nor any part of it has been detected.
 - Z has been detected.
 - ZC has been detected.
 - ZCZ has been detected.
 - ZCZC has been detected.



States are represented by Nodes. State are numbered or may identified by words or whatever else is convenient.

Inputs and Transitions

- ☐ Whatever is being modeled is subjected to inputs. As a result of those inputs, the state changes, or is said to have made a Transition.
- ☐ Transitions are denoted by links that join the states.
- ☐ The input that causes the transition are marked on the link; that is, the inputs are link weights.
- ☐ There is one out link from every state for every input.

- If several inputs in a state cause a transition to the same subsequent state, instead of drawing a bunch of parallel links we can abbreviate the notation by listing the several inputs as in: “input1, input2, input3.....”.

Finite State Machine

- A finite state machine is an abstract device that can be represented by a state graph having a finite number of states and a finite number of transitions between states.
 - Outputs
- An output can be associated with any link.
- Out puts are denoted by letters or words and are separated from inputs by a slash as follows: “input/output”.
- As always, output denotes anything of interest that’s observable and is not restricted to explicit outputs by devices.
- Outputs are also link weights.
- If every input associated with a transition causes the same output, then denoted it as:
 - “input1, input2, input3...../output”

State Tables

- Big state graphs are cluttered and hard to follow.
- It’s more convenient to represent the state graph as a table (the state table or state transition table) that specifies the states, the inputs, the transitions and the outputs.
- The following conventions are used:
 - Each row of the table corresponds to a state.
 - Each column corresponds to an input condition.
 - The box at the intersection of a row and a column specifies the next state (the transition) and the output, if any.

State Table-Example

STATE	inputs		
	Z	C	A
NONE	Z	NONE	NONE
Z	Z	ZC	NONE
ZC	ZCZ	NONE	NONE
ZCZ	Z	ZCZC	NONE
ZCZC	ZCZC	ZCZC	ZCZC

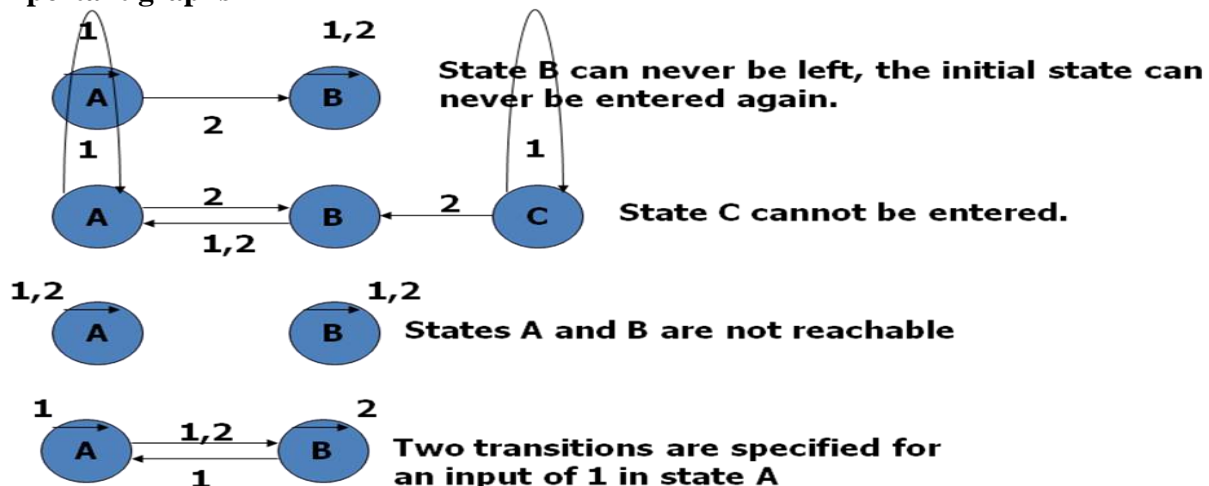
Time Versus Sequence

- State graphs don't represent time—they represent sequence.
 - A transition might take microseconds or centuries;
 - A system could be in one state for milliseconds and another for years- the state graph would be the same because it has no notion of time.
 - Although the finite state machines model can be elaborated to include notions of time in addition to sequence, such as time Petri Nets.
 - Software implementation
 - There is rarely a direct correspondence between programs and the behavior of a process described as a state graph.
 - The state graph represents, the total behavior consisting of the transport, the software, the executive, the status returns, interrupts, and so on.
 - There is no simple correspondence between lines of code and states. The state table forms the basis.

Good State Graphs and Bad

- What constitutes a good or a bad state graph is to some extent biased by the kinds of state graphs that are likely to be used in a software test design context.
- Here are some principles for judging.
 - The total number of states is equal to the product of the possibilities of factors that make up the state.
 - For every state and input there is exactly one transition specified to exactly one, possibly the same, state.
 - For every transition there is one output action specified. The output could be trivial, but at least one output does something sensible.
 - For every state there is a sequence of inputs that will drive the system back to the same state.

Important graphs



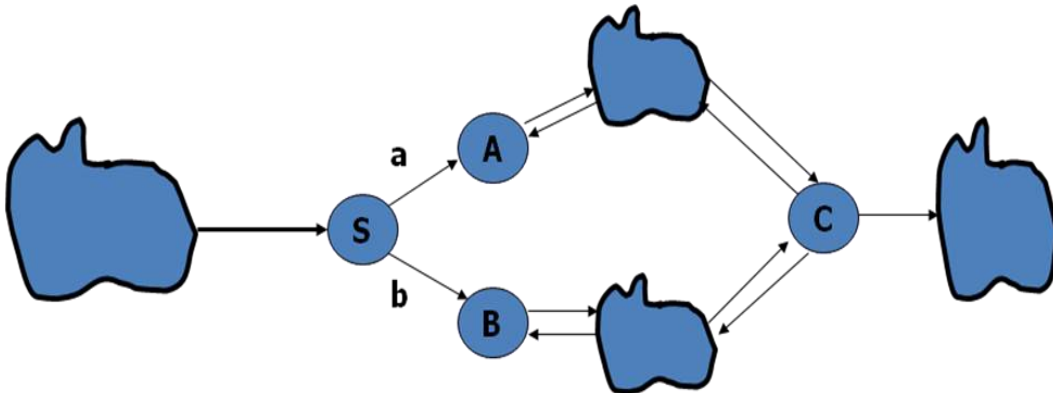
State Bugs-Number of States

- The number of states in a state graph is the number of states we choose to recognize or model.

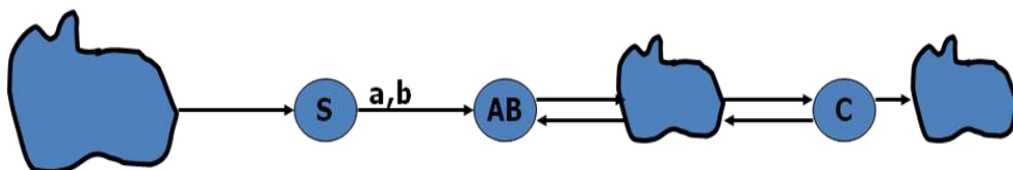
- The state is directly or indirectly recorded as a combination of values of variables that appear in the data base.
- For example, the state could be composed of the value of a counter whose possible values ranged from 0 to 9, combined with the setting of two bit flags, leading to a total of $2*2*10=40$ states.
- The number of states can be computed as follows:
 - Identify all the component factors of the state.
 - Identify all the allowable values for each factor.
 - The number of states is the product of the number of allowable values of all the factors.
- Before you do anything else, before you consider one test case, discuss the number of states you think there are with the number of states the programmer thinks there are.
- There is no point in designing tests intended to check the system's behavior in various states if there's no agreement on how many states there are.
 - Impossible States
- Some times some combinations of factors may appear to be impossible.
- The discrepancy between the programmer's state count and the tester's state count is often due to a difference of opinion concerning "impossible states".
- A robust piece of software will not ignore impossible states but will recognize them and invoke an illogical condition handler when they appear to have occurred.

Equivalent States

- Two states are Equivalent if every sequence of inputs starting from one state produces exactly the same sequence of outputs when started from the other state. This notion can also be extended to set of states.



Merging of Equivalent States



Recognizing Equivalent States

- ☐ Equivalent states can be recognized by the following procedures:
- ☐ The rows corresponding to the two states are identical with respect to input/output/next state but the name of the next state could differ.
- ☐ There are two sets of rows which, except for the state names, have identical state graphs with respect to transitions and outputs. The two sets can be merged.

Transition Bugs-

unspecified and contradictory Transitions

- ☐ Every input-state combination must have a specified transition.
- ☐ If the transition is impossible, then there must be a mechanism that prevents the input from occurring in that state.
- ☐ Exactly one transition must be specified for every combination of input and state.
- A program can't have contradictions or ambiguities.
- ☐ Ambiguities are impossible because the program will do something for every input. Even the state does not change, by definition this is a transition to the same state.

Unreachable States

- ☐ An unreachable state is like unreachable code.
- ☐ A state that no input sequence can reach.
- ☐ An unreachable state is not impossible, just as unreachable code is not impossible
- ☐ There may be transitions from unreachable state to other states; there usually because the state became unreachable as a result of incorrect transition.
- ☐ There are two possibilities for unreachable states:
 - There is a bug; that is some transitions are missing.
 - The transitions are there, but you don't know about it.

Dead States

- ☐ A dead state is a state that once entered cannot be left.
- ☐ This is not necessarily a bug but it is suspicious.

Output Errors

- ☐ The states, transitions, and the inputs could be correct, there could be no dead or unreachable states, but the output for the transition could be incorrect.
 - ☐ Output actions must be verified independently of states and transitions. State Testing
- Impact of Bugs
- ☐ If a routine is specified as a state graph that has been verified as correct in all details. Program code or table or a combination of both must still be implemented.
 - ☐ A bug can manifest itself as one of the following symptoms:
 - ☐ Wrong number of states.
 - ☐ Wrong transitions for a given state-input combination.
 - ☐ Wrong output for a given transition.
 - ☐ Pairs of states or sets of states that are inadvertently made equivalent.
 - ☐ States or set of states that are split to create in equivalent duplicates.

- ☐ States or sets of states that have become dead.
- ☐ States or sets of states that have become unreachable.

Principles of State Testing

- ☐ The strategy for state testing is analogous to that used for path testing flow graphs.
- Just as it's impractical to go through every possible path in a flow graph, it's impractical to go through every path in a state graph.
- ☐ The notion of coverage is identical to that used for flow graphs.
- Even though more state testing is done as a single case in a grand tour, it's impractical to do it that way for several reasons.
- ☐ In the early phases of testing, you will never complete the grand tour because of bugs.
- ☐ Later, in maintenance, testing objectives are understood, and only a few of the states and transitions have to be tested. A grand tour is waste of time.
- ☐ There is no much history in a long test sequence and so much has happened that verification is difficult.

Starting point of state testing

- ☐ Define a set of covering input sequences that get back to the initial state when starting from the initial state.
- ☐ For each step in each input sequence, define the expected next state, the expected transition, and the expected output code.
- ☐ A set of tests, then, consists of three sets of sequences:
 - Input sequences
 - Corresponding transitions or next-state names
 - Output sequences

Limitations and Extensions

- ☐ State transition coverage in a state graph model does not guarantee complete testing.
- ☐ How defines a hierarchy of paths and methods for combining paths to produce covers of state graphs.
- The simplest is called a "0 switch" which corresponds to testing each transition individually.
- The next level consists of testing transitions sequences consisting of two transitions called "1 switches".
- The maximum length switch is "n-1 switch" where there are n numbers of states.
 - Situations at which state testing is useful
- ☐ Any processing where the output is based on the occurrence of one or more sequences of events, such as detection of specified input sequences, sequential format validation, parsing, and other situations in which the order of inputs is important.
- ☐ Most protocols between systems, between humans and machines, between components of a system.
- ☐ Device drivers such as for tapes and discs that have complicated retry and recovery procedures if the action depends on the state.

Whenever a feature is directly and explicitly implemented as one or more state transition tables.

GRAPH MATRICES AND APPLICATIONS

Problem with Pictorial Graphs

- ☐ Graphs were introduced as an abstraction of software structure.
- ☐ Whenever a graph is used as a model, sooner or later we trace paths through it- to find a set of covering paths, a set of values that will sensitize paths, the logic function that controls the flow, the processing time of the routine, the equations that define the domain, or whether a state is reachable or not.
- Path is not easy, and it's subject to error. You can miss a link here and there or cover some links twice.
- One solution to this problem is to represent the graph as a matrix and to use matrix operations equivalent to path tracing. These methods are more methodical and mechanical and don't depend on your ability to see a path they are more reliable.

Tool Building

- ☐ If you build test tools or want to know how they work, sooner or later you will be implementing or investigating analysis routines based on these methods.
- ☐ It is hard to build algorithms over visual graphs so the properties or graph matrices are fundamental to tool building.

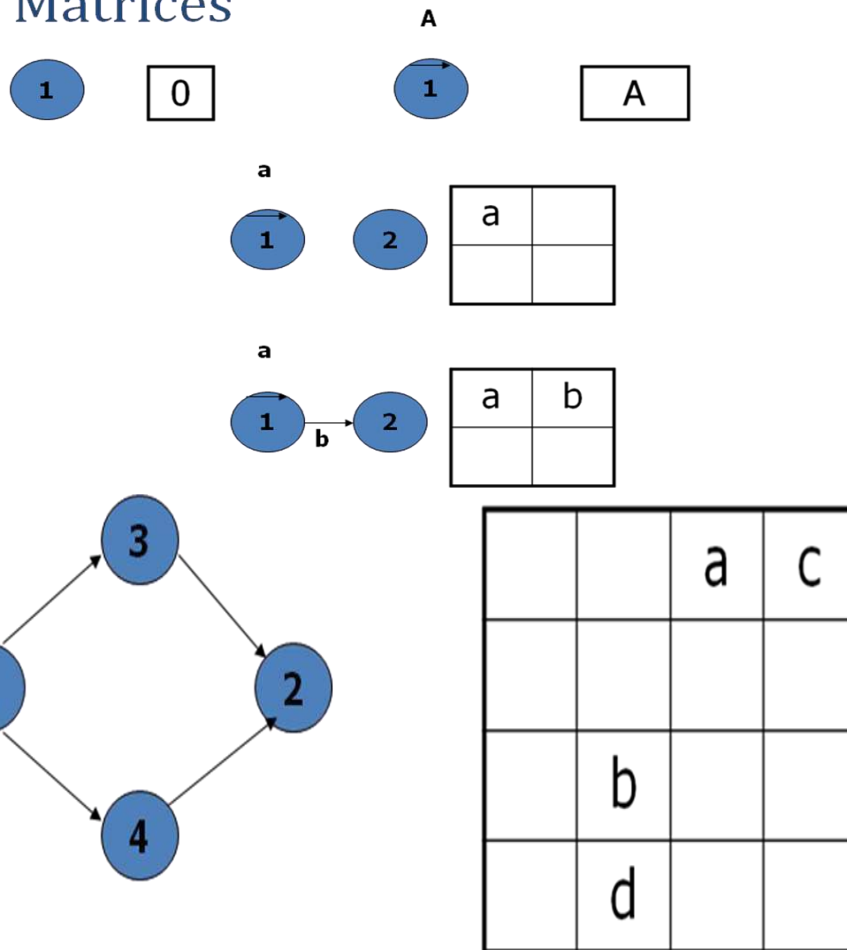
The Basic Algorithms

- ☐ The basic tool kit consists of:
- ☐ Matrix multiplication, which is used to get the path expression from every node to every other node.
- ☐ A partitioning algorithm for converting graphs with loops into loop free graphs or equivalence classes.
- ☐ A collapsing process which gets the path expression from any node to any other node.

The Matrix of a Graph

- A graph matrix is a square array with one row and one column for every node in the graph.
- Each row-column combination corresponds to a relation between the node corresponding to the row and the node corresponding to the column.
- The relation for example, could be as simple as the link name, if there is a link between the nodes.
- ☐ Some of the things to be observed:
- ☐ The size of the matrix equals the number of nodes.
- ☐ There is a place to put every possible direct connection or link between any and any other node.
- ☐ The entry at a row and column intersection is the link weight of the link that connects the two nodes in that direction.
- ☐ A connection from node i to j does not imply a connection from node j to node i.
- If there are several links between two nodes, then the entry is a sum; the "+" sign denotes parallel links as usual.

Some Graphs and their Matrices



A simple weight

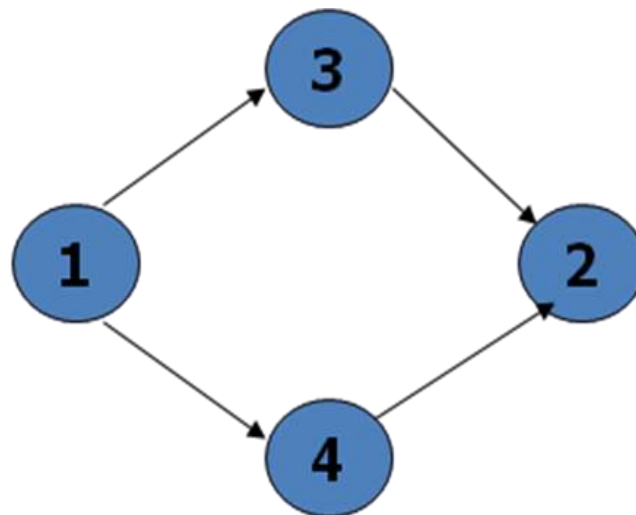
- A simplest weight we can use is to note that there is or isn't a connection. Let "1" mean that there is a connection and "0" mean that there isn't.
- The arithmetic rules are:
 - $1+1=1$ $1*1=1$
 - $1+0=1$ $1*0=0$
 - $0+0=0$ $0*0=0$
- A matrix defined like this is called connection matrix.

Connection matrix

- The connection matrix is obtained by replacing each entry with 1 if there is a link and 0 if there isn't.
- As usual we don't write down 0 entries to reduce the clutter.

		a	c
	b		
	d		

		1	1
	1		
	1		



Connection Matrix-continued

- ☐ Each row of a matrix denotes the out links of the node corresponding to that row.
- ☐ Each column denotes the in links corresponding to that node.
- ☐ A branch is a node with more than one nonzero entry in its row.
- ☐ A junction is node with more than one nonzero entry in its column.
- ☐ A self loop is an entry along the diagonal.

Cyclomatic Complexity

- The cyclomatic complexity obtained by subtracting 1 from the total number of entries in each row and ignoring rows with no entries, we obtain the equivalent number of decisions for each row. Adding these values and then adding 1 to the sum yields the graph's cyclomatic complexity.

		1	1	$2-1=1$
	1			$1-1=0$
	1			$1-1=0$

$1+1=2$ (cyclomatic complexity)

Relations

- ☐ A relation is a property that exists between two objects of interest.
- ☐ For example,
 - “Node a is connected to node b” or aRb where “R” means “is connected to”.
 - “ $a \geq b$ ” or aRb where “R” means greater than or equal”.
- ☐ A graph consists of set of abstract objects called nodes and a relation R between the nodes.
- ☐ If aRb , which is to say that a has the relation R to b, it is denoted by a link from a to b.
- ☐ For some relations we can associate properties called as link weights.

Transitive Relations

- ☐ A relation is transitive if aRb and bRc implies aRc .
- ☐ Most relations used in testing are transitive.
- ☐ Examples of transitive relations include: is connected to, is greater than or equal to, is less than or equal to, is a relative of, is faster than, is slower than, takes more time than, is a subset of, includes, shadows, is the boss of.
- ☐ Examples of intransitive relations include: is acquainted with, is a friend of, is a neighbor of, is lied to, has a du chain between.

Reflexive Relations

- ☐ A relation R is reflexive if, for every a, aRa .
- ☐ A reflexive relation is equivalent to a self loop at every node.
- ☐ Examples of reflexive relations include: equals, is acquainted with, is a relative of.
- ☐ Examples of irreflexive relations include: not equals, is a friend of, is on top of, is under.

Symmetric Relations

- ☐ A relation R is symmetric if for every a and b, aRb implies bRa .
- ☐ A symmetric relation mean that if there is a link from a to b then there is also a link from b to a.
- ☐ A graph whose relations are not symmetric are called directed graph.

- ☐ A graph over a symmetric relation is called an undirected graph.
- ☐ The matrix of an undirected graph is symmetric ($a_{ij}=a_{ji}$) for all i,j

Antisymmetric Relations

- ☐ A relation R is antisymmetric if for every a and b , if aRb and bRa , then $a=b$, or they are the same elements.
- ☐ Examples of antisymmetric relations: is greater than or equal to, is a subset of, time.
- ☐ Examples of nonantisymmetric relations: is connected to, can be reached from, is greater than, is a relative of, is a friend of

equivalence Relations

- ☐ An equivalence relation is a relation that satisfies the reflexive, transitive, and symmetric properties.
- ☐ Equality is the most familiar example of an equivalence relation.
- ☐ If a set of objects satisfy an equivalence relation, we say that they form an equivalence class over that relation.
- ☐ The importance of equivalence classes and relations is that any member of the equivalence class is, with respect to the relation, equivalent to any other member of that class.
- ☐ The idea behind partition testing strategies such as domain testing and path testing, is that we can partition the input space into equivalence classes.
- ☐ Testing any member of the equivalence class is as effective as testing them all.

Partial Ordering Relations

- ☐ A partial ordering relation satisfies the reflexive, transitive, and antisymmetric properties.
- ☐ Partial ordered graphs have several important properties: they are loop free, there is at least one maximum element, and there is at least one minimum element.

The Powers of a Matrix

- Each entry in the graph's matrix expresses a relation between the pair of nodes that corresponds to that entry.
- ☐ Squaring the matrix yields a new matrix that expresses the relation between each pair of nodes via one intermediate node under the assumption that the relation is transitive.
- ☐ The square of the matrix represents all path segments two links long.
- ☐ The third power represents all path segments three links long.

Matrix Powers and Products

- Given a matrix whose entries are a_{ij} , the square of that matrix is obtained by replacing every entry with
 - n
 - $a_{ij} = \sum_{k=1}^n a_{ik} a_{kj}$
 - $k=1$
- more generally, given two matrices A and B with entries a_{ik} and b_{kj} , respectively, their product is a new matrix C, whose entries are c_{ij} , where:
 - n
 - $C_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$
 - $k=1$

Partitioning Algorithm

- Consider any graph over a transitive relation. The graph may have loops.
- We would like to partition the graph by grouping nodes in such a way that every loop is contained within one group or another.
- Such a graph is partially ordered.
- There are many used for an algorithm that does that:
- We might want to embed the loops within a subroutine so as to have a resulting graph which is loop free at the top level.
- Many graphs with loops are easy to analyze if you know where to break the loops.
- While you and I can recognize loops, it's much harder to program a tool to do it unless you have a solid algorithm on which to base the tool.

Node Reduction Algorithm (General)

- The matrix powers usually tell us more than we want to know about most graphs.
 - In the context of testing, we usually interested in establishing a relation between two nodes- typically the entry and exit nodes.
 - In a debugging context it is unlikely that we would want to know the path expression between every node and every other node.
 - The advantage of matrix reduction method is that it is more methodical than the graphical method called as node by node removal algorithm.
1. Select a node for removal; replace the node by equivalent links that bypass that node and add those links to the links they parallel.
 2. Combine the parallel terms and simplify as you can.
 3. Observe loop terms and adjust the out links of every node that had a self loop to account for the effect of the loop.
 4. The result is a matrix whose size has been reduced by 1. Continue until only the two nodes of interest exist.

PYTHON PROGRAMMING

[R17A0554]

LECTURE NOTES

B.TECH III YEAR – II SEM (R17)
(2019-20)



DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING

**MALLA REDDY COLLEGE OF ENGINEERING &
TECHNOLOGY**

(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12 (B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015
Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, India

SYLLABUS

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

III Year B. Tech CSE -II SEM

L	T/P/D	C
3	- / - / -	3

OPEN ELECTIVE III (R17A0554) PYTHON PROGRAMMING

OBJECTIVES:

- To read and write simple Python programs.
- To develop Python programs with conditionals and loops.
- To define Python functions and call them.
- To use Python data structures — lists, tuples, dictionaries.
- To do input/output with files in Python.

UNIT I

INTRODUCTION DATA, EXPRESSIONS, STATEMENTS

Introduction to Python and installation, data types: Int, float, Boolean, string, and list; variables, expressions, statements, precedence of operators, comments; modules, functions--- function and its use, flow of execution, parameters and arguments.

UNIT II

CONTROL FLOW, LOOPS

Conditionals: Boolean values and operators, conditional (if), alternative (if-else), chained conditional (if-elif-else); Iteration: while, for, break, continue.

UNIT III

FUNCTIONS, ARRAYS

Fruitful functions: return values, parameters, local and global scope, function composition, recursion; Strings: string slices, immutability, string functions and methods, string module; Python arrays, Access the Elements of an Array, array methods.

UNIT IV

LISTS, TUPLES, DICTIONARIES

Lists: list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters, list comprehension; Tuples: tuple assignment, tuple as return value, tuple comprehension; Dictionaries: operations and methods, comprehension;

UNIT V

FILES, EXCEPTIONS, MODULES, PACKAGES

Files and exception: text files, reading and writing files, command line arguments, errors and exceptions, handling exceptions, modules (datetime, time, OS , calendar, math module), Explore packages.

OUTCOMES: Upon completion of the course, students will be able to

- Read, write, execute by hand simple Python programs.
- Structure simple Python programs for solving problems.
- Decompose a Python program into functions.
- Represent compound data using Python lists, tuples, dictionaries.
- Read and write data from/to files in Python Programs

TEXT BOOKS

1. Allen B. Downey, ``Think Python: How to Think Like a Computer Scientist``, 2nd edition, Updated for Python 3, Shroff/O'Reilly Publishers, 2016.
2. R. Nageswara Rao, "Core Python Programming", dreamtech
3. Python Programming: A Modern Approach, Vamsi Kurama, Pearson

REFERENCE BOOKS:

1. Core Python Programming, W.Chun, Pearson.
2. Introduction to Python, Kenneth A. Lambert, Cengage
3. Learning Python, Mark Lutz, Orielly

INDEX

UNIT	TOPIC	PAGE NO
I	INTRODUCTION DATA, EXPRESSIONS, STATEMENTS	1
	Introduction to Python and installation	1
	data types: Int	6
	float	7
	Boolean	8
	string	8
	List	10
	variables	11
	expressions	13
	statements	16
	precedence of operators	17
	comments	18
	modules	19
	functions ---- function and its use	20
	flow of execution	21
	parameters and arguments	26
II	CONTROL FLOW, LOOPS	35
	Conditionals: Boolean values and operators,	35
	conditional (if)	36
	alternative (if-else)	37
	chained conditional (if-elif-else)	39
	Iteration: while, for, break, continue.	41
III	FUNCTIONS, ARRAYS	55
	Fruitful functions: return values	55
	parameters	57
	local and global scope	59
	function composition	62
	recursion	63
	Strings: string slices	64
	immutability	66
	string functions and methods	67
	string module	72
	Python arrays	73
	Access the Elements of an Array	75
	Array methods	76

IV	LISTS, TUPLES, DICTIONARIES	78
	Lists	78
	list operations	79
	list slices	80
	list methods	81
	list loop	83
	mutability	85
	aliasing	87
	cloning lists	88
	list parameters	89
	list comprehension	90
	Tuples	91
	tuple assignment	94
	tuple as return value	95
	tuple comprehension	96
	Dictionaries	97
	operations and methods	97
	comprehension	102
V	FILES, EXCEPTIONS, MODULES, PACKAGES	103
	Files and exception: text files	103
	reading and writing files	104
	command line arguments	109
	errors and exceptions	112
	handling exceptions	114
	modules (datetime, time, OS , calendar, math module)	121
	Explore packages	134

UNIT – I**INTRODUCTION DATA, EXPRESSIONS, STATEMENTS**

Introduction to Python and installation, data types: Int, float, Boolean, string, and list; variables, expressions, statements, precedence of operators, comments; modules, functions--
- function and its use, flow of execution, parameters and arguments.

Introduction to Python and installation:

Python is a widely used general-purpose, high level programming language. It was initially designed by **Guido van Rossum in 1991** and developed by Python Software Foundation. It was mainly developed for emphasis on code readability, and its syntax allows programmers to express concepts in fewer lines of code.

Python is a programming language that lets you work quickly and integrate systems more efficiently.

There are two major Python versions- **Python 2 and Python 3**.

- On 16 October 2000, Python 2.0 was released with many new features.
- On 3rd December 2008, Python 3.0 was released with more testing and includes new features.

Beginning with Python programming:**1) Finding an Interpreter:**

Before we start Python programming, we need to have an interpreter to interpret and run our programs. There are certain online interpreters like <https://ide.geeksforgeeks.org/>, <http://ideone.com/> or <http://codepad.org/> that can be used to start Python without installing an interpreter.

Windows: There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from <http://python.org/downloads/>

2) Writing first program:

Script Begins

Statement1

Statement2

Statement3

Script Ends

Differences between scripting language and programming language:

SCRIPTING LANGUAGE	PROGRAMMING LANGUAGE
A programming language that supports scripts: programs written for a special run-time environment that automate the execution of tasks	A formal language, which comprises a set of instructions used to produce various kinds of output
Execution speed is slow	Compiler-based languages are executed much faster while interpreter-based languages are executed slower
Can be divided into client-side scripting languages and server-side scripting languages	Can be divided into high-level, low-level languages or compiler-based or interpreter-based languages
Easier to learn	Not as easy to learn
Ex: JavaScript, Perl, PHP, Python and Ruby	Ex: C, C++, and Assembly
Mostly used for web development	Used to develop various applications such as desktop, web, mobile, etc.

Why to use Python:

The following are the primary factors to use python in day-to-day life:

1. Python is object-oriented

Structure supports such concepts as polymorphism, operation overloading and multiple inheritance.

2. Indentation

Indentation is one of the greatest feature in python

3. It's free (open source)

Downloading python and installing python is free and easy

4. It's Powerful

- Dynamic typing
- Built-in types and tools
- Library utilities
- Third party utilities (e.g. Numeric, NumPy, sciPy)
- Automatic memory management

5. It's Portable

- Python runs virtually every major platform used today
- As long as you have a compatible python interpreter installed, python programs will run in exactly the same manner, irrespective of platform.

6. It's easy to use and learn

- No intermediate compile
- Python Programs are compiled automatically to an intermediate form called byte code, which the interpreter then reads.
- This gives python the development speed of an interpreter without the performance loss inherent in purely interpreted languages.
- Structure and syntax are pretty intuitive and easy to grasp.

7. Interpreted Language

Python is processed at runtime by python Interpreter

8. Interactive Programming Language

Users can interact with the python interpreter directly for writing the programs

9. Straight forward syntax

The formation of python syntax is simple and straight forward which also makes it popular.

Installation:

There are many interpreters available freely to run Python scripts like IDLE (Integrated Development Environment) which is installed when you install the python software from <http://python.org/downloads/>

Steps to be followed and remembered:

Step 1: Select Version of Python to Install.

Step 2: Download Python Executable Installer.

Step 3: Run Executable Installer.

Step 4: Verify Python Was Installed On Windows.

Step 5: Verify Pip Was Installed.

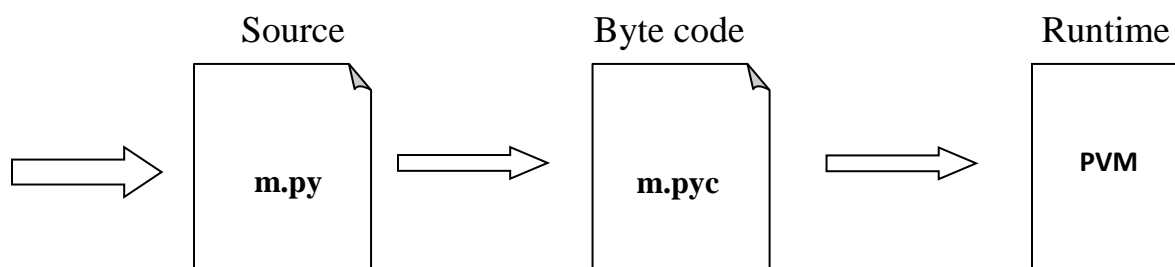
Step 6: Add Python Path to Environment Variables (Optional)



Working with Python

Python Code Execution:

Python's traditional runtime execution model: Source code you type is translated to byte code, which is then run by the Python Virtual Machine (PVM). Your code is automatically compiled, but then it is interpreted.



Source code extension is .py
Byte code extension is .pyc (Compiled python code)

There are two modes for using the Python interpreter:

- Interactive Mode
- Script Mode

Running Python in interactive mode:

Without passing python script file to the interpreter, directly execute code to Python prompt. Once you're inside the python interpreter, then you can start.

```
>>> print("hello world")
```

```
hello world
```

Relevant output is displayed on subsequent lines without the >>> symbol

```
>>> x=[0,1,2]
```

Quantities stored in memory are not displayed by default.

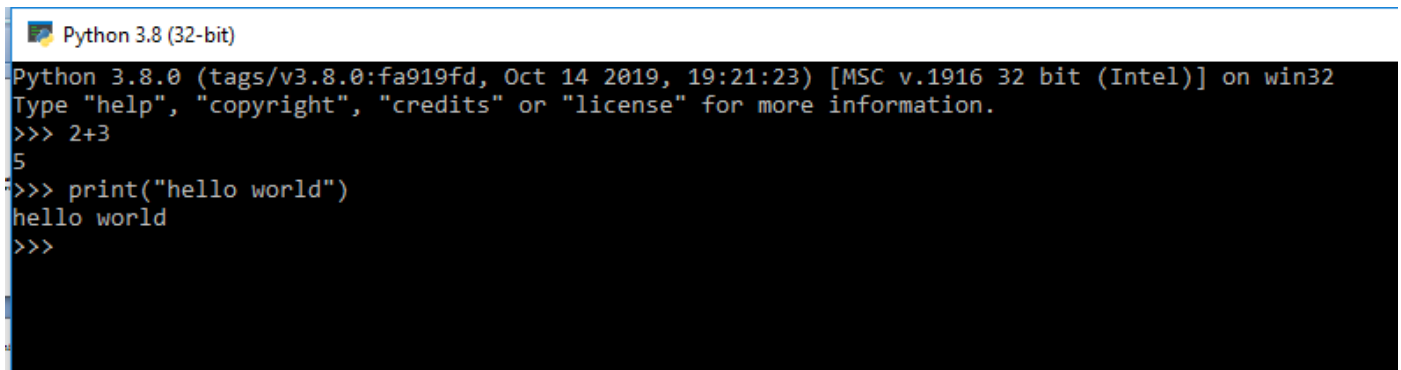
```
>>> x
```

#If a quantity is stored in memory, typing its name will display it.

```
[0, 1, 2]
```

```
>>> 2+3
```

```
5
```

A screenshot of a Python 3.8 (32-bit) interpreter window. The window title is "Python 3.8 (32-bit)". The command prompt shows the following text: "Python 3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)] on win32". Below this, it says "Type 'help', 'copyright', 'credits' or 'license' for more information." The user has entered the following commands and received the following output:

```
>>> 2+3
5
>>> print("hello world")
hello world
>>>
```

The chevron at the beginning of the 1st line, i.e., the symbol >>> is a prompt the python interpreter uses to indicate that it is ready. If the programmer types 2+6, the interpreter replies 8.

Running Python in script mode:

Alternatively, programmers can store Python script source code in a file with the .py extension, and use the interpreter to execute the contents of the file. To execute the script by the interpreter, you have to tell the interpreter the name of the file. For example, if you have a script name MyFile.py and you're working on Unix, to run the script you have to type:

python MyFile.py

Working with the interactive mode is better when Python programmers deal with small pieces of code as you can type and execute them immediately, but when the code is more than 2-4 lines, using the script for coding can help to modify and use the code in future.

Example:

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy>python e1.py
resource open
the no cant be divisibile zero division by zero
resource close
finished
```

Data types:

The data stored in memory can be of many types. For example, a student roll number is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Int:

Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.

```
>>> print(24656354687654+2)
```

```
24656354687656
```

```
>>> print(20)
```

```
20
```

```
>>> print(0b10)
```

```
2
```

```
>>> print(0B10)
```

```
2
```

```
>>> print(0X20)
```

```
32
```

```
>>> 20
```

```
20
```

```
>>> 0b10
```

```
2
```

```
>>> a=10
```

```
>>> print(a)
```

```
10
```

To verify the type of any object in Python, use the type() function:

```
>>> type(10)
```

```
<class 'int'>
```

```
>>> a=11
```

```
>>> print(type(a))
```

```
<class 'int'>
```

Float:

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Float can also be scientific numbers with an "e" to indicate the power of 10.

```
>>> y=2.8
```

```
>>> y
```

```
2.8
```

```
>>> y=2.8
```

```
>>> print(type(y))
```

```
<class 'float'>
```

```
>>> type(.4)
```

```
<class 'float'>
```

```
>>> 2.
```

2.0

Example:`x = 35e3``y = 12E4``z = -87.7e100``print(type(x))``print(type(y))``print(type(z))`**Output:**`<class 'float'>``<class 'float'>``<class 'float'>`**Boolean:**

Objects of Boolean type may have one of two values, True or False:

`>>> type(True)``<class 'bool'>``>>> type(False)``<class 'bool'>`**String:**

1. Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes.

- 'hello' is the same as "hello".
- Strings can be output to screen using the print function. **For example: print('hello').**

`>>> print("mrcet college")``mrcet college``>>> type("mrcet college")``<class 'str'>`

```
>>> print('mrcet college')
```

```
mrcet college
```

```
>>> " "
```

```
' '
```

If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type. If a string is to contain a single quote, delimit it with double quotes and vice versa:

```
>>> print("mrcet is an autonomous (') college")
```

```
mrcet is an autonomous (') college
```

```
>>> print('mrcet is an autonomous (") college')
```

```
mrcet is an autonomous (") college
```

Suppressing Special Character:

Specifying a backslash (\) in front of the quote character in a string “escapes” it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single quote character:

```
>>> print("mrcet is an autonomous (\') college")
```

```
mrcet is an autonomous (') college
```

```
>>> print('mrcet is an autonomous (\") college')
```

```
mrcet is an autonomous (") college
```

The following is a table of escape sequences which cause Python to suppress the usual special interpretation of a character in a string:

```
>>> print('a\
```

```
....b')
```

```
a....b
```

```
>>> print('a\
```

```
b\
```

```
c')
```

```
abc
>>> print('a \n b')
a
b
>>> print("mrcet \n college")
mrcet
college
```

Escape Sequence	Usual Interpretation of Character(s) After Backslash	“Escaped” Interpretation
\'	Terminates string with single quote opening delimiter	Literal single quote (') character
\"	Terminates string with double quote opening delimiter	Literal double quote (") character
\newline	Terminates input line	Newline is ignored
\\	Introduces escape sequence	Literal backslash (\) character

In Python (and almost all other common computer languages), a tab character can be specified by the escape sequence `\t`:

```
>>> print("a\tb")
a    b
```

List:

- It is a general purpose most widely used in data structures
- List is a collection which is ordered and changeable and allows duplicate members. (Grow and shrink as needed, sequence type, sortable).
- To use a list, you must declare it first. Do this using square brackets and separate values with commas.
- We can construct / create list in many ways.

Ex:

```
>>> list1=[1,2,3,'A','B',7,8,[10,11]]
>>> print(list1)
[1, 2, 3, 'A', 'B', 7, 8, [10, 11]]
```

```
-----  
>>> x=list()
```

```
>>> x
```

```
[]  
-----
```

```
>>> tuple1=(1,2,3,4)
```

```
>>> x=list(tuple1)
```

```
>>> x
```

```
[1, 2, 3, 4]
```

Variables:

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Assigning Values to Variables:

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable.

For example –

```
a= 100      # An integer assignment
```

```
b = 1000.0   # A floating point
```

```
c = "John"   # A string
```

```
print (a)
```

```
print (b)
```

```
print (c)
```

This produces the following result –

```
100
```

```
1000.0
```

```
John
```

Multiple Assignment:

Python allows you to assign a single value to several variables simultaneously.

For example :

```
a = b = c = 1
```

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables.

For example –

```
a,b,c = 1,2,"mrcet"
```

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

Output Variables:

The Python print statement is often used to output variables.

Variables do not need to be declared with any particular type and can even change type after they have been set.

```
x = 5          # x is of type int
x = "mrcet "   # x is now of type str
print(x)
```

Output: mrcet

To combine both text and a variable, Python uses the “+” character:

Example

```
x = "awesome"
print("Python is " + x)
```

Output

Python is awesome

You can also use the + character to add a variable to another variable:

Example

```
x = "Python is "
y = "awesome"
z = x + y
print(z)
```

Output:

Python is awesome

Expressions:

An expression is a combination of values, variables, and operators. An expression is evaluated using assignment operator.

Examples: $Y = x + 17$

```
>>> x=10
```

```
>>> z=x+20
```

```
>>> z
```

```
30
```



```
>>> x=10
>>> y=20
>>> c=x+y
>>> c
30
```

A value all by itself is a simple expression, and so is a variable.

```
>>> y=20
>>> y
20
```

Python also defines expressions only contain identifiers, literals, and operators. So,

Identifiers: Any name that is used to define a class, function, variable module, or object is an identifier.

Literals: These are language-independent terms in Python and should exist independently in any programming language. In Python, there are the string literals, byte literals, integer literals, floating point literals, and imaginary literals.

Operators: In Python you can implement the following operations using the corresponding tokens.

Operator	Token
add	+
subtract	-
multiply	*
Integer Division	/
remainder	%
Binary left shift	<<
Binary right shift	>>
and	&
or	\
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Check equality	==
Check not equal	!=

Some of the python expressions are:**Generator expression:**

Syntax: (compute(var) for var in iterable)

```
>>> x = (i for i in 'abc') #tuple comprehension
>>> x
<generator object <genexpr> at 0x033EEC30>

>>> print(x)
<generator object <genexpr> at 0x033EEC30>
```

You might expect this to print as ('a', 'b', 'c') but it prints as <generator object <genexpr> at 0x02AAD710> The result of a tuple comprehension is not a tuple: it is actually a generator. The only thing that you need to know now about a generator now is that you can iterate over it, but ONLY ONCE.

Conditional expression:

Syntax: true_value if Condition else false_value

```
>>> x = "1" if True else "2"

>>> x

'1'
```

Statements:

A statement is an instruction that the Python interpreter can execute. We have normally two basic statements, the assignment statement and the print statement. Some other kinds of statements that are if statements, while statements, and for statements generally called as control flows.

Examples:

An assignment statement creates new variables and gives them values:

```
>>> x=10
```

```
>>> college="mrcet"
```

An print statement is something which is an input from the user, to be printed / displayed on to the screen (or) monitor.

```
>>> print("mrcet colege")
```

```
mrcet college
```

Precedence of Operators:

Operator precedence affects how an expression is evaluated.

For example, $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first multiplies $3*2$ and then adds into 7.

Example 1:

```
>>> 3+4*2
```

```
11
```

Multiplication gets evaluated before the addition operation

```
>>> (10+10)*2
```

```
40
```

Parentheses () overriding the precedence of the arithmetic operators

Example 2:

```
a = 20
```

```
b = 10
```

```
c = 15
```

```
d = 5
```

```
e = 0
```

```
e = (a + b) * c / d    #( 30 * 15 ) / 5  
print("Value of (a + b) * c / d is ", e)
```

```
e = ((a + b) * c) / d    # (30 * 15 ) / 5  
print("Value of ((a + b) * c) / d is ", e)
```

```
e = (a + b) * (c / d);    # (30) * (15/5)
```

```
print("Value of (a + b) * (c / d) is ", e)
```

```
e = a + (b * c) / d;    # 20 + (150/5)
print("Value of a + (b * c) / d is ", e)
```

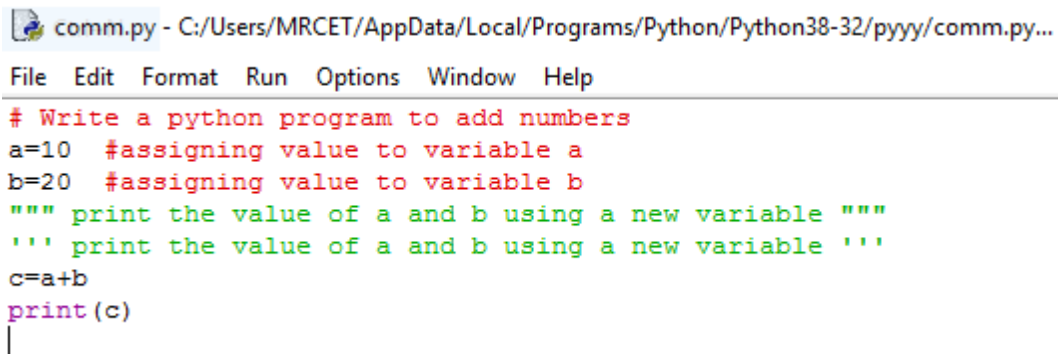
Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/opprec.py
Value of (a + b) * c / d is 90.0
Value of ((a + b) * c) / d is 90.0
Value of (a + b) * (c / d) is 90.0
Value of a + (b * c) / d is 50.0
```

Comments:

Single-line comments begins with a hash(#) symbol and is useful in mentioning that the whole line should be considered as a comment until the end of line.

A Multi line comment is useful when we need to comment on many lines. In python, triple double quote(“ “ “) and single quote(‘ ‘ ‘)are used for multi-line commenting.

Example:

```
comm.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/comm.py...
File Edit Format Run Options Window Help
# Write a python program to add numbers
a=10 #assigning value to variable a
b=20 #assigning value to variable b
""" print the value of a and b using a new variable """
''' print the value of a and b using a new variable '''
c=a+b
print(c)
|
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/comm.py
```

```
30
```

Modules:

Modules: Python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module. A module in Python provides us the flexibility to organize the code in a logical way. To use the functionality of one module into another, we must have to **import** the specific module.

Syntax:

```
import <module-name>
```

Every module has its own functions, those can be accessed with . (dot)

Note: In python we have help ()

Enter the name of any module, keyword, or topic to get help on writing Python programs and using Python modules. To quit this help utility and return to the interpreter, just type "quit".

Some of the modules like os, date, and calendar so on.....

```
>>> import sys
```

```
>>> print(sys.version)
```

```
3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)]
```

```
>>> print(sys.version_info)
```

```
sys.version_info(major=3, minor=8, micro=0, releaselevel='final', serial=0)
```

```
>>> print(calendar.month(2021,5))
```

```
    May 2021
Mo Tu We Th Fr Sa Su
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

```
>>> print(calendar.isleap(2020))
```

```
True
```

```
>>> print(calendar.isleap(2017))
```

```
False
```

Functions:

Functions and its use: Function is a group of related statements that perform a specific task. Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable. It avoids repetition and makes code reusable.

Basically, we can divide functions into the following two types:

1. **Built-in functions** - Functions that are built into Python.

Ex: abs(),all().ascii(),bool().....so on....

```
integer = -20
```

```
print('Absolute value of -20 is:', abs(integer))
```

Output:

Absolute value of -20 is: 20

2. **User-defined functions** - Functions defined by the users themselves.

```
def add_numbers(x,y):
```

```
    sum = x + y
```

```
    return sum
```

```
print("The sum is", add_numbers(5, 20))
```

Output:

The sum is 25

Flow of Execution:

1. The order in which statements are executed is called the flow of execution
2. Execution always begins at the first statement of the program.
3. Statements are executed one at a time, in order, from top to bottom.
4. Function definitions do not alter the flow of execution of the program, but remember that statements inside the function are not executed until the function is called.
5. Function calls are like a bypass in the flow of execution. Instead of going to the next statement, the flow jumps to the first line of the called function, executes all the statements there, and then comes back to pick up where it left off.

Note: When you read a program, don't read from top to bottom. Instead, follow the flow of execution. This means that you will read the def statements as you are scanning from top to bottom, but you should skip the statements of the function definition until you reach a point where that function is called.

Example:

#example for flow of execution

```
print("welcome")
for x in range(3):
    print(x)
print("Good morning college")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py

welcome

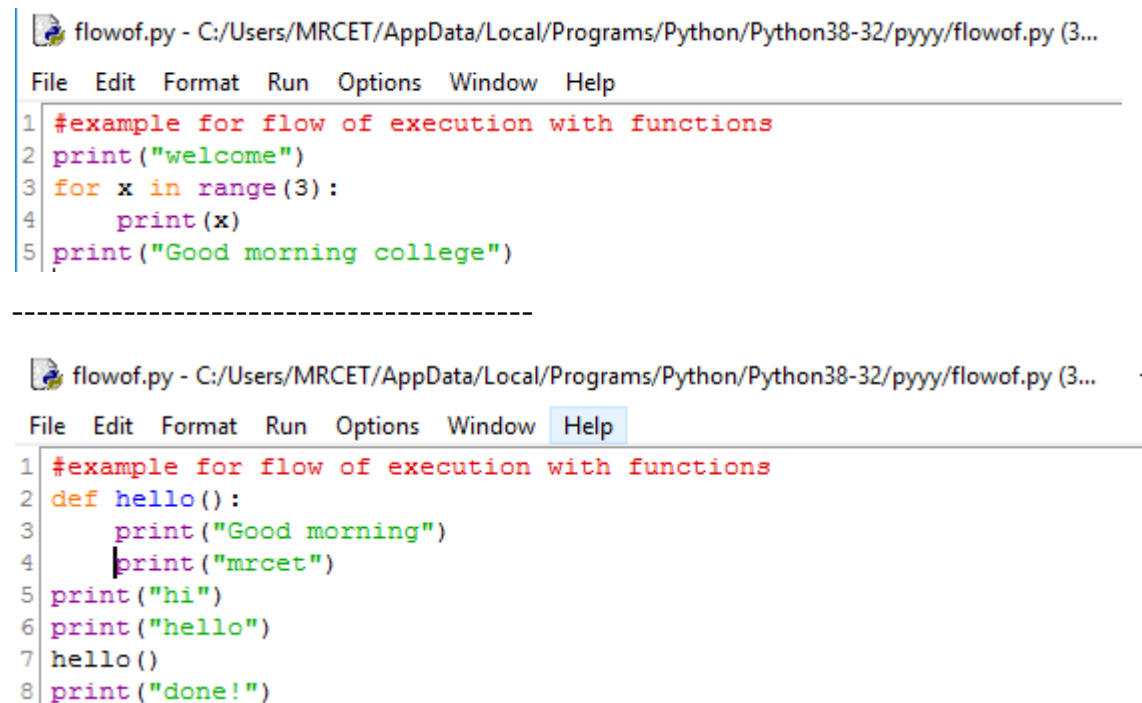
0

1

2

Good morning college

The flow/order of execution is: 2,3,4,3,4,3,4,5



```
flowof.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py (3...
File Edit Format Run Options Window Help
1 #example for flow of execution with functions
2 print("welcome")
3 for x in range(3):
4     print(x)
5 print("Good morning college")

-----

flowof.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py (3...
File Edit Format Run Options Window Help
1 #example for flow of execution with functions
2 def hello():
3     print("Good morning")
4     print("mrcet")
5 print("hi")
6 print("hello")
7 hello()
8 print("done!")
```


Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/flowof.py

hi

hello

Good morning

mrcet

done!

The flow/order of execution is: 2,5,6,7,2,3,4,7,8

Parameters and arguments:

Parameters are passed during the definition of function while Arguments are passed during the function call.

Example:

#here a and b are parameters

```
def add(a,b): ##function definition
    return a+b
```

#12 and 13 are arguments

#function call

```
result=add(12,13)
```

```
print(result)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/paraarg.py

25

There are three types of Python function arguments using which we can call a function.

1. Default Arguments
2. Keyword Arguments
3. Variable-length Arguments

Syntax:

```
def functionname():
```

statements

•
•
•

functionname()

Function definition consists of following components:

1. Keyword **def** indicates the start of function header.
2. A function name to uniquely identify it. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A **colon (:)** to mark the end of function header.
5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have same indentation level (usually 4 spaces).
7. An optional return statement to return a value from the function.

Example:

```
def hf():
```

```
    hello world
```

```
hf()
```

In the above example we are just trying to execute the program by calling the function. So it will not display any error and no output on to the screen but gets executed.

To get the statements of function need to be use print().

#calling function in python:

```
def hf():
```

```
    print("hello world")
```

```
hf()
```

Output:

```
hello world
```

```
-----
```

```
def hf():  
    print("hw")  
    print("gh kfjg 66666")
```

```
hf()
```

```
hf()
```

```
hf()
```

Output:

```
hw  
gh kfjg 66666  
hw  
gh kfjg 66666  
hw  
gh kfjg 66666
```

```
def add(x,y):
```

```
    c=x+y
```

```
    print(c)
```

```
add(5,4)
```

Output:

```
9
```

```
def add(x,y):
```

```
    c=x+y
```

```
    return c
```

```
print(add(5,4))
```

Output:

```
9
```

```
def add_sub(x,y):  
    c=x+y  
    d=x-y  
    return c,d  
print(add_sub(10,5))
```

Output:

(15, 5)

The **return** statement is used to exit a function and go back to the place from where it was called. This statement can contain expression which gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the **None** object.

```
def hf():  
    return "hw"  
print(hf())
```

Output:

hw

```
def hf():  
    return "hw"  
hf()
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu.py

>>>

```
def hello_f():  
    return "hellocollege"  
print(hello_f().upper())
```

Output:

HELLOCOLLEGE

Passing Arguments

```
def hello(wish):  
    return '{}'.format(wish)  
print(hello("mrcet"))
```

Output:

mrcet

Here, the function wish() has two parameters. Since, we have called this function with two arguments, it runs smoothly and we do not get any error. If we call it with different number of arguments, the interpreter will give errors.

```
def wish(name,msg):  
    """This function greets to  
    the person with the provided message"""  
    print("Hello",name + ' ' + msg)  
wish("MRCET","Good morning!")
```

Output:

Hello MRCET Good morning!

Below is a call to this function with one and no arguments along with their respective error messages.

```
>>> wish("MRCET")  # only one argument
TypeError: wish() missing 1 required positional argument: 'msg'
>>> wish()  # no arguments
TypeError: wish() missing 2 required positional arguments: 'name' and 'msg'
```

```
-----

def hello(wish,hello):

    return "hi" '{}{}'.format(wish,hello)

print(hello("mrcet","college"))
```

Output:

```
himrcet,college
```

#Keyword Arguments

When we call a function with some values, these values get assigned to the arguments according to their position.

Python allows functions to be called using keyword arguments. When we call functions in this way, the order (position) of the arguments can be changed.

(Or)

If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called **keyword arguments** - we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.

There are two *advantages* - one, using the function is easier since we do not need to worry about the order of the arguments. Two, we can give values to only those parameters which we want, provided that the other parameters have default argument values.

```
def func(a, b=5, c=10):
    print 'a is', a, 'and b is', b, 'and c is', c
```

```
func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Output:

```
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

Note:

The function named func has one parameter without default argument values, followed by two parameters with default argument values.

In the first usage, func(3, 7), the parameter a gets the value 3, the parameter b gets the value 5 and c gets the default value of 10.

In the second usage func(25, c=24), the variable a gets the value of 25 due to the position of the argument. Then, the parameter c gets the value of 24 due to naming i.e. keyword arguments. The variable b gets the default value of 5.

In the third usage func(c=50, a=100), we use keyword arguments completely to specify the values. Notice, that we are specifying value for parameter c before that for a even though a is defined before c in the function definition.

For example: if you define the function like below

```
def func(b=5, c=10,a): # shows error : non-default argument follows default argument
```

```
-----
def print_name(name1, name2):
```

```
    """ This function prints the name """
```

```
    print (name1 + " and " + name2 + " are friends")
```

```
#calling the function
```

```
print_name(name2 = 'A',name1 = 'B')
```

Output:

B and A are friends

#Default Arguments

Function arguments can have default values in Python.

We can provide a default value to an argument by using the assignment operator (=)

```
def hello(wish,name='you'):
    return '{},{ {}'.format(wish,name)

print(hello("good morning"))
```

Output:

good morning,you

```
def hello(wish,name='you'):
    return '{},{ {}'.format(wish,name)    //print(wish + ' ' + name)

print(hello("good morning","nirosha")) //hello("good morning","nirosha")
```

Output:

good morning,nirosha // good morning nirosha

Note: Any number of arguments in a function can have a default value. But once we have a default argument, all the arguments to its right must also have default values.

This means to say, non-default arguments cannot follow default arguments. For example, if we had defined the function header above as:

```
def hello(name='you', wish):
```

Syntax Error: non-default argument follows default argument

```
def sum(a=4, b=2): #2 is supplied as default argument
```



```
""" This function will print sum of two numbers

    if the arguments are not supplied

    it will add the default value """

print (a+b)

sum(1,2) #calling with arguments

sum( )  #calling without arguments
```

Output:

3

6

Variable-length arguments

Sometimes you may need more arguments to process function then you mentioned in the definition. If we don't know in advance about the arguments needed in function, we can use variable-length arguments also called arbitrary arguments.

For this an asterisk (*) is placed before a parameter in function definition which can hold non-keyworded variable-length arguments and a double asterisk (**) is placed before a parameter in function which can hold keyworded variable-length arguments.

If we use one asterisk (*) like *var, then all the positional arguments from that point till the end are collected as a tuple called 'var' and if we use two asterisks (**) before a variable like **var, then all the positional arguments from that point till the end are collected as a dictionary called 'var'.

```
def wish(*names):
    """This function greets all
    the person in the names tuple."""

    # names is a tuple with arguments
    for name in names:
        print("Hello",name)

wish("MRCET","CSE","SIR","MADAM")
```

Output:

Hello MRCET
Hello CSE
Hello SIR
Hello MADAM

#Program to find area of a circle using function use single return value function with argument.

```
pi=3.14
def areaOfCircle(r):

    return pi*r*r
r=int(input("Enter radius of circle"))

print(areaOfCircle(r))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
Enter radius of circle 3
28.259999999999998
```

#Program to write sum different product and using arguments with return value function.

```
def calculete(a,b):

    total=a+b

    diff=a-b

    prod=a*b

    div=a/b

    mod=a%b
```

```
    return total,diff,prod,div,mod

a=int(input("Enter a value"))
b=int(input("Enter b value"))

#function call
s,d,p,q,m = calculate(a,b)

print("Sum= ",s,"diff= ",d,"mul= ",p,"div= ",q,"mod= ",m)

#print("diff= ",d)

#print("mul= ",p)

#print("div= ",q)

#print("mod= ",m)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

Enter a value 5

Enter b value 6

Sum= 11 diff= -1 mul= 30 div= 0.8333333333333334 mod= 5

#program to find biggest of two numbers using functions.

```
def biggest(a,b):
    if a>b :
        return a
    else :
        return b

a=int(input("Enter a value"))
b=int(input("Enter b value"))
#function call
big= biggest(a,b)
print("big number= ",big)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

Enter a value 5

Enter b value-2

big number= 5

#program to find biggest of two numbers using functions. (nested if)

```
def biggest(a,b,c):
```

```
    if a>b :
```

```
        if a>c :
```

```
            return a
```

```
        else :
```

```
            return c
```

```
    else :
```

```
        if b>c :
```

```
            return b
```

```
        else :
```

```
            return c
```

```
a=int(input("Enter a value"))
```

```
b=int(input("Enter b value"))
```

```
c=int(input("Enter c value"))
```

```
#function call
```

```
big= biggest(a,b,c)
```

```
print("big number= ",big)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

Enter a value 5

Enter b value -6

Enter c value 7

big number= 7

#Writer a program to read one subject mark and print pass or fail use single return values function with argument.

```
def result(a):
```

```
    if a>40:
```

```
        return "pass"
```

```
else:  
    return "fail"  
a=int(input("Enter one subject marks"))  
  
print(result(a))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py  
Enter one subject marks 35  
fail
```

#Write a program to display mrcet cse dept 10 times on the screen. (while loop)

```
def usingFunctions():  
    count =0  
    while count<10:  
        print("mrcet cse dept",count)  
        count=count+1  
  
usingFunctions()
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py  
mrcet cse dept 0  
mrcet cse dept 1  
mrcet cse dept 2  
mrcet cse dept 3  
mrcet cse dept 4  
mrcet cse dept 5  
mrcet cse dept 6  
mrcet cse dept 7  
mrcet cse dept 8  
mrcet cse dept 9
```

UNIT – II**CONTROL FLOW, LOOPS**

Conditionals: Boolean values and operators, conditional (if), alternative (if-else), chained conditional (if-elif-else); Iteration: while, for, break, continue.

Control Flow, Loops:**Boolean Values and Operators:**

A boolean expression is an expression that is either true or false. The following examples use the operator `==`, which compares two operands and produces True if they are equal and False otherwise:

```
>>> 5 == 5
```

```
True
```

```
>>> 5 == 6
```

```
False
```

True and False are special values that belong to the type `bool`; they are not strings:

```
>>> type(True)
```

```
<class 'bool'>
```

```
>>> type(False)
```

```
<class 'bool'>
```

The `==` operator is one of the relational operators; the others are: `x != y` # x is not equal to y

`x > y` # x is greater than y `x < y` # x is less than y

`x >= y` # x is greater than or equal to y `x <= y` # x is less than or equal to y

Note:

All expressions involving relational and logical operators will evaluate to either true or false

Conditional (if):

The if statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.

Syntax:

if expression:

statement(s)

If the boolean expression evaluates to TRUE, then the block of statement(s) inside the if statement is executed. If boolean expression evaluates to FALSE, then the first set of code after the end of the if statement(s) is executed.

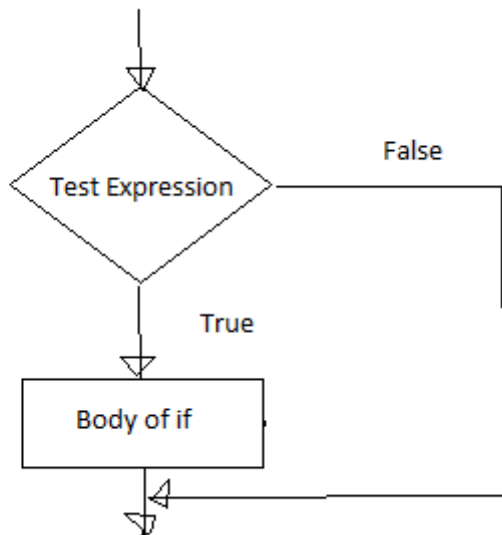
if Statement Flowchart:

Fig: Operation of if statement

Example: Python if Statement

```
a = 3
if a > 2:
    print(a, "is greater")
print("done")
```

```
a = -1
if a < 0:
    print(a, "a is smaller")
print("Finish")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/if1.py
3 is greater
done
-1 a is smaller
Finish
-----
```

```
a=10
```

```
if a>9:
```

```
    print("A is Greater than 9")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/if2.py
```

```
A is Greater than 9
```

Alternative if (If-Else):

An else statement can be combined with an if statement. An else statement contains the block of code (false block) that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

The else statement is an optional statement and there could be at most only one else Statement following if.

Syntax of if - else :

```
if test expression:
```

```
    Body of if stmts
```

```
else:
```

```
    Body of else stmts
```

If - else Flowchart :

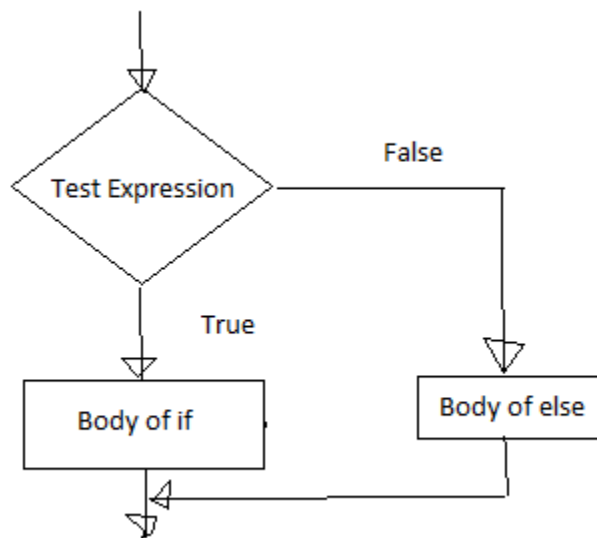


Fig: Operation of if – else statement

Example of if - else:

```
a=int(input('enter the number'))
if a>5:
    print("a is greater")
else:
    print("a is smaller than the input given")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py

enter the number 2

a is smaller than the input given

a=10

b=20

if a>b:

print("A is Greater than B")

else:

print("B is Greater than A")

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/if2.py

B is Greater than A

Chained Conditional: (If-elif-else):

The elif statement allows us to check multiple expressions for TRUE and execute a block of code as soon as one of the conditions evaluates to TRUE. Similar to the else, the elif statement is optional. However, unlike else, for which there can be at most one statement, there can be an arbitrary number of elif statements following an if.

Syntax of if – elif - else :

If test expression:

 Body of if stmts

elif test expression:

 Body of elif stmts

else:

 Body of else stmts

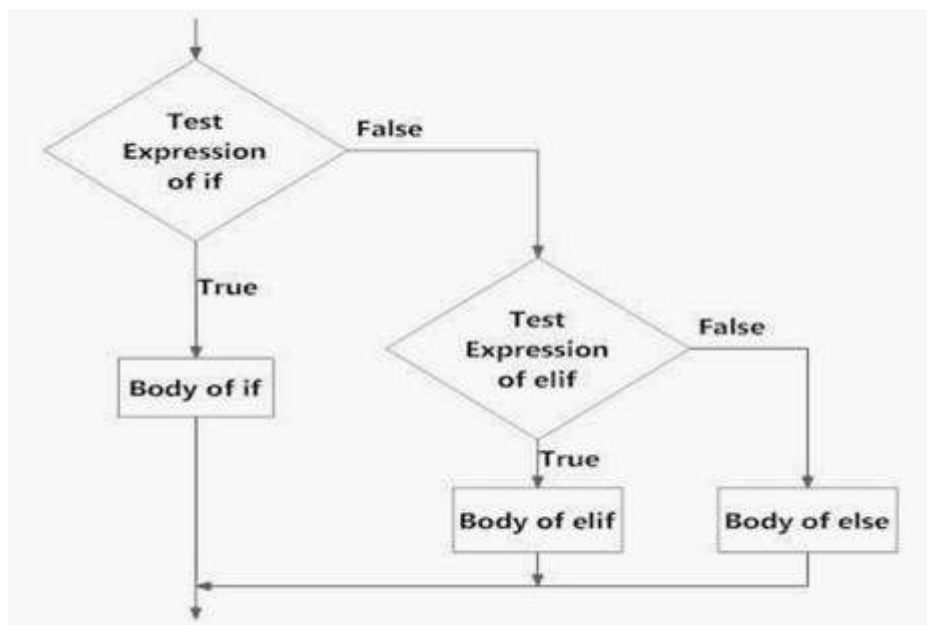
Flowchart of if – elif - else:

Fig: Operation of if – elif - else statement

Example of if - elif – else:

```
a=int(input('enter the number'))
b=int(input('enter the number'))
c=int(input('enter the number'))
if a>b:
```

```
print("a is greater")
elif b>c:
    print("b is greater")
else:
    print("c is greater")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py

enter the number5

enter the number2

enter the number9

a is greater

>>>

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ifelse.py

enter the number2

enter the number5

enter the number9

c is greater

var = 100

if var == 200:

print("1 - Got a true expression value")

print(var)

elif var == 150:

print("2 - Got a true expression value")

print(var)

elif var == 100:

print("3 - Got a true expression value")

print(var)

else:

print("4 - Got a false expression value")

print(var)

print("Good bye!")

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ifelif.py

3 - Got a true expression value

100

Good bye!

Iteration:

A loop statement allows us to execute a statement or group of statements multiple times as long as the condition is true. Repeated execution of a set of statements with the help of loops is called iteration.

Loops statements are used when we need to run same code again and again, each time with a different value.

Statements:

In Python Iteration (Loops) statements are of three types:

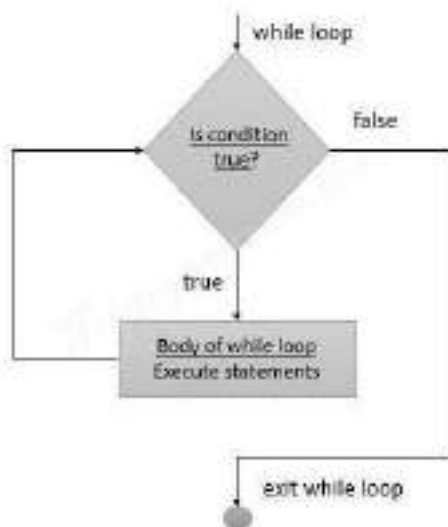
1. While Loop
2. For Loop
3. Nested For Loops

While loop:

- Loops are either infinite or conditional. Python while loop keeps reiterating a block of code defined inside it until the desired condition is met.
- The while loop contains a boolean expression and the code inside the loop is repeatedly executed as long as the boolean expression is true.
- The statements that are executed inside while can be a single line of code or a block of multiple statements.

Syntax:

```
while(expression):  
    Statement(s)
```

Flowchart:

Example Programs:

```
1. -----  
i=1  
while i<=6:  
    print("Mrcet college")  
    i=i+1
```

output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh1.py

Mrcet college

Mrcet college

Mrcet college

Mrcet college

Mrcet college

Mrcet college

```
2. -----  
i=1  
  
while i<=3:  
    print("MRCET",end=" ")  
    j=1  
    while j<=1:  
        print("CSE DEPT",end="")  
        j=j+1  
    i=i+1  
    print()
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh2.py

MRCET CSE DEPT

MRCET CSE DEPT

MRCET CSE DEPT

```
3. -----  
  
i=1
```

```
j=1
while i<=3:
    print("MRCET",end=" ")

    while j<=1:
        print("CSE DEPT",end="")
        j=j+1
    i=i+1
    print()
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh3.py

MRCET CSE DEPT
MRCET
MRCET

4. -----

```
i = 1
while (i < 10):
    print (i)
    i = i+1
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh4.py

1
2
3
4
5
6
7
8
9

2. -----

```
a = 1
b = 1
while (a<10):
    print ('Iteration',a)
    a = a + 1
    b = b + 1
```

```
if (b == 4):  
    break  
print ('While loop terminated')
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh5.py

Iteration 1

Iteration 2

Iteration 3

While loop terminated

count = 0

while (count < 9):

print("The count is:", count)

count = count + 1

print("Good bye!")

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/wh.py =

The count is: 0

The count is: 1

The count is: 2

The count is: 3

The count is: 4

The count is: 5

The count is: 6

The count is: 7

The count is: 8

Good bye!

For loop:

Python **for loop** is used for repeated execution of a group of statements for the desired number of times. It iterates over the items of lists, tuples, strings, the dictionaries and other iterable objects

Syntax: for var in sequence:

Statement(s)

↓
Holds the value of item
in sequence in each iteration

↓
A sequence of values assigned to var in each iteration

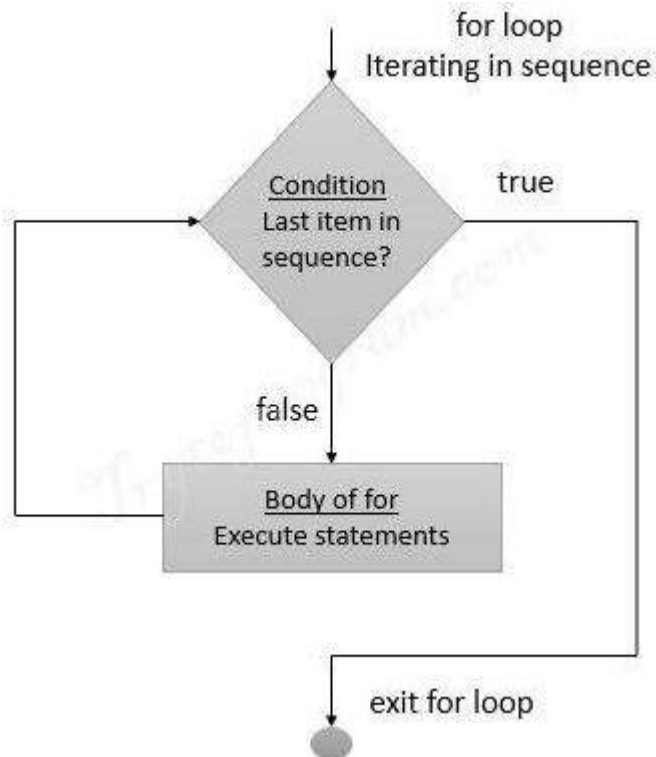
Sample Program:

```
numbers = [1, 2, 4, 6, 11, 20]
seq=0
for val in numbers:
    seq=val*val
    print(seq)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/fr.py

1
4
16
36
121
400

Flowchart:

Iterating over a list:

```
#list of items
list = ['M','R','C','E','T']
i = 1

#Iterating over the list
for item in list:
    print ('college ',i,' is ',item)
    i = i+1
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/lis.py
college 1 is M
college 2 is R
college 3 is C
college 4 is E
college 5 is T
```

Iterating over a Tuple:

```
tuple = (2,3,5,7)
print ('These are the first four prime numbers ')
#Iterating over the tuple
for a in tuple:
    print (a)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fr3.py
These are the first four prime numbers
2
3
5
7
```

Iterating over a dictionary:

```
#creating a dictionary
college = {"ces":"block1","it":"block2","ece":"block3"}

#Iterating over the dictionary to print keys
print ('Keys are:')
```

```
for keys in college:  
    print (keys)
```

```
#Iterating over the dictionary to print values  
print ('Values are:')  
for blocks in college.values():  
    print(blocks)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/dic.py

Keys are:

ces

it

ece

Values are:

block1

block2

block3

Iterating over a String:

```
#declare a string to iterate over  
college = 'MRCET'
```

```
#Iterating over the string  
for name in college:  
    print (name)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/strr.py

M

R

C

E

T

Nested For loop:

When one Loop defined within another Loop is called Nested Loops.

Syntax:

```
for val in sequence:
```

```
    for val in sequence:
```

statements

statements

Example 1 of Nested For Loops (Pattern Programs)

```
for i in range(1,6):  
    for j in range(0,i):  
        print(i, end=" ")  
    print("")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/nesforr.py

```
1  
2 2  
3 3 3  
4 4 4 4  
5 5 5 5 5
```

Example 2 of Nested For Loops (Pattern Programs)

```
for i in range(1,6):  
    for j in range(5,i-1,-1):  
        print(i, end=" ")  
    print("")
```

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/nesforr.py

Output:

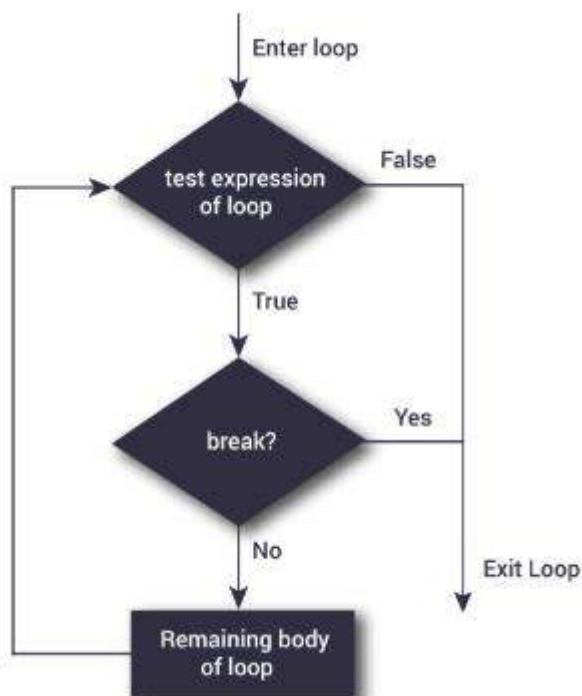
```
1 1 1 1 1  
2 2 2 2  
3 3 3  
4 4
```

Break and continue:

In Python, **break** and **continue** statements can alter the flow of a normal loop. Sometimes we wish to terminate the current iteration or even the whole loop without checking test expression. The break and continue statements are used in these cases.

Break:

The break statement terminates the loop containing it and control of the program flows to the statement immediately after the body of the loop. If break statement is inside a nested loop (loop inside another loop), break will terminate the innermost loop.

Flowchart:

The following shows the working of break statement in for and while loop:

for var in sequence:

 # code inside for loop

 If condition:

 break (if break condition satisfies it jumps to outside loop)

 # code inside for loop

code outside for loop

while test expression

 # code inside while loop

 If condition:

 break (if break condition satisfies it jumps to outside loop)

 # code inside while loop

code outside while loop

Example:

```
for val in "MRCET COLLEGE":
```

```
    if val == " ":
```

```
        break
```

```
    print(val)
```

```
print("The end")
```

Output:

M

R

C

E

T

The end

Program to display all the elements before number 88

```
for num in [11, 9, 88, 10, 90, 3, 19]:
```

```
    print(num)
```

```
    if(num==88):
```

```
        print("The number 88 is found")
```

```
        print("Terminating the loop")
```

```
        break
```

Output:

11

9

88

The number 88 is found

Terminating the loop

```
#-----  
for letter in "Python": # First Example  
    if letter == "h":  
        break  
    print("Current Letter :", letter )
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/br.py =

Current Letter : P

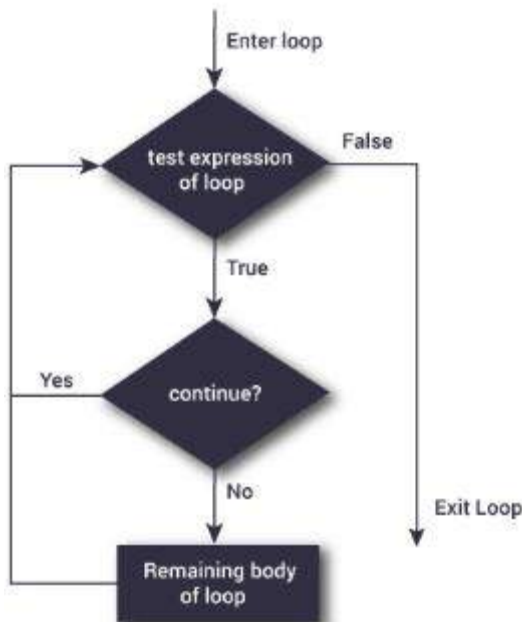
Current Letter : y

Current Letter : t

Continue:

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

Flowchart:



The following shows the working of break statement in for and while loop:

for var in sequence:

 # code inside for loop

 If condition:

 continue (if break condition satisfies it jumps to outside loop)

 # code inside for loop

code outside for loop

while test expression

 # code inside while loop

 If condition:

 continue (if break condition satisfies it jumps to outside loop)

 # code inside while loop

code outside while loop

Example:

Program to show the use of continue statement inside loops

```
for val in "string":
```

```
    if val == "i":
```

```
        continue
```

```
    print(val)
```

```
print("The end")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/cont.py
```

```
s
```

```
t
```

```
r
```

```
n
```

```
g
```

```
The end
```

program to display only odd numbers

```
for num in [20, 11, 9, 66, 4, 89, 44]:
```

```
# Skipping the iteration when number is even
if num%2 == 0:
    continue
# This statement will be skipped for all even numbers
print(num)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/cont2.py

11

9

89

#-----

for letter in "Python": # First Example

if letter == "h":

continue

print("Current Letter :", letter)

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/con1.py

Current Letter : P

Current Letter : y

Current Letter : t

Current Letter : o

Current Letter : n

Pass:

In Python programming, pass is a null statement. The difference between a comment and pass statement in Python is that, while the interpreter ignores a comment entirely, pass is not ignored.

pass is just a placeholder for functionality to be added later.

Example:

```
sequence = {'p', 'a', 's', 's'}
```

```
for val in sequence:
```

```
    pass
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fl.y.py


```
>>>
```

Similarly we can also write,

```
def f(arg): pass    # a function that does nothing (yet)
```

```
class C: pass      # a class with no methods (yet)
```

UNIT – III**FUNCTIONS, ARRAYS**

Fruitful functions: return values, parameters, local and global scope, function composition, recursion; Strings: string slices, immutability, string functions and methods, string module; Python arrays, Access the Elements of an Array, array methods.

Functions, Arrays:**Fruitful functions:**

We write functions that return values, which we will call fruitful functions. We have seen the return statement before, but in a fruitful function the return statement includes a return value. This statement means: "Return immediately from this function and use the following expression as a return value."

(or)

Any function that returns a value is called Fruitful function. A Function that does not return a value is called a void function

Return values:

The Keyword return is used to return back the value to the called function.

returns the area of a circle with the given radius:

```
def area(radius):  
    temp = 3.14 * radius**2  
    return temp  
print(area(4))
```

(or)

```
def area(radius):  
    return 3.14 * radius**2  
print(area(2))
```

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
def absolute_value(x):  
    if x < 0:
```

```
    return -x
else:
    return x
```

Since these return statements are in an alternative conditional, only one will be executed.

As soon as a return statement executes, the function terminates without executing any subsequent statements. Code that appears after a return statement, or any other place the flow of execution can never reach, is called dead code.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. For example:

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

This function is incorrect because if x happens to be 0, both conditions is true, and the function ends without hitting a return statement. If the flow of execution gets to the end of a function, the return value is None, which is not the absolute value of 0.

```
>>> print absolute_value(0)
None
```

By the way, Python provides a built-in function called abs that computes absolute values.

Write a Python function that takes two lists and returns True if they have at least one common member.

```
def common_data(list1, list2):
    for x in list1:
        for y in list2:
            if x == y:
                result = True
                return result
print(common_data([1,2,3,4,5], [1,2,3,4,5]))
print(common_data([1,2,3,4,5], [1,7,8,9,510]))
print(common_data([1,2,3,4,5], [6,7,8,9,10]))
```

Output:

C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\pyyy\fu1.py

```
True
True
None
```

```
#-----
```

```
def area(radius):
```

```
    b = 3.14159 * radius**2
```

```
    return b
```

Parameters:

Parameters are passed during the definition of function while Arguments are passed during the function call.

Example:

#here a and b are parameters

```
def add(a,b): #function definition
```

```
    return a+b
```

#12 and 13 are arguments

#function call

```
result=add(12,13)
```

```
print(result)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/paraarg.py
```

```
25
```

Some examples on functions:

To display vandemataram by using function use no args no return type

#function defination

```
def display():
```

```
    print("vandemataram")
```

```
print("i am in main")
```

```
#function call  
display()  
print("i am in main")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py  
i am in main  
vandemataram  
i am in main
```

#Type1 : No parameters and no return type

```
def Fun1() :  
    print("function 1")  
Fun1()
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py  
  
function 1
```

#Type 2: with param with out return type

```
def fun2(a) :  
    print(a)  
fun2("hello")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py  
  
Hello
```

#Type 3: without param with return type

```
def fun3():  
    return "welcome to python"  
print(fun3())
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
```

```
welcome to python
```

#Type 4: with param with return type

```
def fun4(a):  
    return a  
print(fun4("python is better then c"))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
```

```
python is better then c
```

Local and Global scope:**Local Scope:**

A variable which is defined inside a function is local to that function. It is accessible from the point at which it is defined until the end of the function, and exists for as long as the function is executing

Global Scope:

A variable which is defined in the main body of a file is called a global variable. It will be visible throughout the file, and also inside any file which imports that file.

- The variable defined inside a function can also be made global by using the global statement.

```
def function_name(args):  
    .....  
    global x    #declaring global variable inside a function  
    .....
```

create a global variable

```
x = "global"

def f():
    print("x inside :", x)

f()
print("x outside:", x)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

x inside : global

x outside: global

create a local variable

```
def f1():
    y = "local"
    print(y)

f1()
```

Output:

local

- If we try to access the local variable outside the scope for example,

```
def f2():
    y = "local"

f2()
print(y)
```

Then when we try to run it shows an error,

Traceback (most recent call last):

File "C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py", line 6, in <module>

```
print(y)
NameError: name 'y' is not defined
```

The output shows an error, because we are trying to access a local variable y in a global scope whereas the local variable only works inside f2() or local scope.

use local and global variables in same code

```
x = "global"

def f3():
    global x
    y = "local"
    x = x * 2
    print(x)
    print(y)

f3()
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py
globalglobal
local
```

- In the above code, we declare x as a global and y as a local variable in the f3(). Then, we use multiplication operator * to modify the global variable x and we print both x and y.
- After calling the f3(), the value of x becomes global global because we used the x * 2 to print two times global. After that, we print the value of local variable y i.e local.

use Global variable and Local variable with same name

```
x = 5

def f4():
    x = 10
    print("local x:", x)

f4()
print("global x:", x)
```


Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/fu1.py

local x: 10

global x: 5

Function Composition:

Having two (or more) functions where the output of one function is the input for another. So for example if you have two functions FunctionA and FunctionB you compose them by doing the following.

FunctionB(FunctionA(x))

Here x is the input for FunctionA and the result of that is the input for FunctionB.

Example 1:**#create a function compose2**

```
>>> def compose2(f, g):  
    return lambda x:f(g(x))
```

```
>>> def d(x):  
    return x*2
```

```
>>> def e(x):  
    return x+1
```

```
>>> a=compose2(d,e) # FunctionC = compose(FunctionB,FunctionA)  
>>> a(5)            # FunctionC(x)  
12
```

In the above program we tried to compose n functions with the main function created.

Example 2:

```
>>> colors=('red','green','blue')
```

```
>>> fruits=['orange','banana','cherry']
```

```
>>> zip(colors,fruits)
```

```
<zip object at 0x03DAC6C8>
```

```
>>> list(zip(colors,fruits))
```

```
[('red', 'orange'), ('green', 'banana'), ('blue', 'cherry')]
```

Recursion:

Recursion is the process of defining something in terms of itself.

Python Recursive Function

We know that in Python, a function can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

Factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is $1*2*3*4*5*6 = 720$.

Following is an example of recursive function to find the factorial of an integer.

```
# Write a program to factorial using recursion
```

```
def fact(x):  
    if x==0:  
        result = 1  
    else :  
        result = x * fact(x-1)  
    return result  
print("zero factorial",fact(0))  
print("five factorial",fact(5))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/rec.py
```

```
zero factorial 1
```

```
five factorial 120
```

```
-----  
def calc_factorial(x):  
    """This is a recursive function  
    to find the factorial of an integer"""  
  
    if x == 1:  
        return 1  
    else:  
        return (x * calc_factorial(x-1))
```

```
num = 4
print("The factorial of", num, "is", calc_factorial(num))
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/rec.py

The factorial of 4 is 24

Strings:

A string is a group/ a sequence of characters. Since Python has no provision for arrays, we simply use strings. This is how we declare a string. We can use a pair of single or double quotes. Every string object is of the type 'str'.

```
>>> type("name")
<class 'str'>
>>> name=str()
>>> name
''
>>> a=str('mrcet')
>>> a
'mrcet'
>>> a=str(mrcet)
>>> a[2]
'c'
>>> fruit = 'banana'
>>> letter = fruit[1]
```

The second statement selects character number 1 from fruit and assigns it to letter. The expression in brackets is called an index. The index indicates which character in the sequence we want

String slices:

A segment of a string is called a slice. Selecting a slice is similar to selecting a character:

Subsets of strings can be taken using the slice operator (**[] and [:]**) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

Slice out substrings, sub lists, sub Tuples using index.

Syntax:[Start: stop: steps]

- Slicing will start from index and will go up to **stop** in **step** of steps.
- Default value of start is 0,

- Stop is last index of list
- And for step default is 1

For example 1–

```
str = 'Hello World!'
```

```
print str # Prints complete string
```

```
print str[0] # Prints first character of the string
```

```
print str[2:5] # Prints characters starting from 3rd to 5th
```

```
print str[2:] # Prints string starting from 3rd character print
```

```
str * 2 # Prints string two times
```

```
print str + "TEST" # Prints concatenated string
```

Output:

Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST

Example 2:

```
>>> x='computer'
```

```
>>> x[1:4]
```

```
'omp'
```

```
>>> x[1:6:2]
```

```
'opt'
```

```
>>> x[3:]
```

```
'puter'
>>> x[:5]
'compu'
>>> x[-1]
'r'
>>> x[-3:]
'ter'
>>> x[:-2]
'comput'
>>> x[::-2]
'rtpo'
>>> x[::-1]
'retupmoc'
```

Immutability:

It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string.

For example:

```
>>> greeting='mrcet college!'
>>> greeting[0]='n'
```

TypeError: 'str' object does not support item assignment

The reason for the error is that strings are **immutable**, which means we can't change an existing string. The best we can do is creating a new string that is a variation on the original:

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> new_greeting
'Jello, world!'
```

Note: The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator

String functions and methods:

There are many methods to operate on String.

S.no	Method name	Description
1.	isalnum()	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
2.	isalpha()	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
3.	isdigit()	Returns true if string contains only digits and false otherwise.
4.	islower()	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
5.	isnumeric()	Returns true if a string contains only numeric characters and false otherwise.
6.	isspace()	Returns true if string contains only whitespace characters and false otherwise.
7.	istitle()	Returns true if string is properly “titlecased” and false otherwise.
8.	isupper()	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
9.	replace(old, new [, max])	Replaces all occurrences of old in string with new or at most max occurrences if max given.
10.	split()	Splits string according to delimiter str (space if not provided) and returns list of substrings;
11.	count()	Occurrence of a string in another string
12.	find()	Finding the index of the first occurrence of a string in another string
13.	swapcase()	Converts lowercase letters in a string to uppercase and viceversa
14.	startswith(str, beg=0, end=len(string))	Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.

Note:

All the string methods will be returning either true or false as the result

1. isalnum():

Isalnum() method returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.

Syntax:

String.isalnum()

Example:

```
>>> string="123alpha"
>>> string.isalnum() True
```

2. isalpha():

isalpha() method returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

Syntax:

String.isalpha()

Example:

```
>>> string="nikhil"
>>> string.isalpha()
True
```

3. isdigit():

isdigit() returns true if string contains only digits and false otherwise.

Syntax:

String.isdigit()

Example:

```
>>> string="123456789"
>>> string.isdigit()
True
```

4. islower():

Islower() returns true if string has characters that are in lowercase and false otherwise.

Syntax:

String.islower()

Example:

```
>>> string="nikhil"  
>>> string.islower()  
True
```

5. isnumeric():

isnumeric() method returns true if a string contains only numeric characters and false otherwise.

Syntax:

String.isnumeric()

Example:

```
>>> string="123456789"  
>>> string.isnumeric()  
True
```

6. isspace():

isspace() returns true if string contains only whitespace characters and false otherwise.

Syntax:

String.isspace()

Example:

```
>>> string=" "  
>>> string.isspace()  
True
```

7. istitle()

istitle() method returns true if string is properly “titlecased”(starting letter of each word is capital) and false otherwise

Syntax:

String.istitle()

Example:

```
>>> string="Nikhil Is Learning"
>>> string.istitle()
True
```

8. isupper()

isupper() returns true if string has characters that are in uppercase and false otherwise.

Syntax:

String.isupper()

Example:

```
>>> string="HELLO"
>>> string.isupper()
True
```

9. replace()

replace() method replaces all occurrences of old in string with new or at most max occurrences if max given.

Syntax:

String.replace()

Example:

```
>>> string="Nikhil Is Learning"
>>> string.replace('Nikhil','Neha')
'Neha Is Learning'
```

10.split()

split() method splits the string according to delimiter str (space if not provided)

Syntax:

String.split()

Example:

```
>>> string="Nikhil Is Learning"
>>> string.split()
```

```
['Nikhil', 'Is', 'Learning']
```

11.count()

count() method counts the occurrence of a string in another string Syntax:

String.count()

Example:

```
>>> string='Nikhil Is Learning'
>>> string.count('i')
3
```

12.find()

Find() method is used for finding the index of the first occurrence of a string in another string

Syntax:

String.find(„string“)

Example:

```
>>> string="Nikhil Is Learning"
>>> string.find('k')
2
```

13.swapcase()

converts lowercase letters in a string to uppercase and viceversa

Syntax:

String.find(„string“)

Example:

```
>>> string="HELLO"
>>> string.swapcase()
'hello'
```

14.startswith()

Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.

Syntax:

```
String.startswith(„string“)
```

Example:

```
>>> string="Nikhil Is Learning"
>>> string.startswith('N')
True
```

15.endswith()

Determines if string or a substring of string (if starting index beg and ending index end are given) ends with substring str; returns true if so and false otherwise.

Syntax:

```
String.endswith(„string“)
```

Example:

```
>>> string="Nikhil Is Learning"
>>> string.startswith('g')
True
```

String module:

This module contains a number of functions to process standard Python strings. In recent versions, most functions are available as string methods as well.

It's a built-in module and we have to **import** it before using any of its constants and classes

Syntax: import string

Note:

help(string) --- gives the information about all the variables ,functions, attributes and classes to be used in string module.

Example:

```
import string
print(string.ascii_letters)
print(string.ascii_lowercase)
print(string.ascii_uppercase)
print(string.digits)
```

```
print(string.hexdigits)
#print(string.whitespace)
print(string.punctuation)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/strrmodl.py

```
=====
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
0123456789abcdefABCDEF
!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~
```

Python String Module Classes

Python string module contains two classes – Formatter and Template.

Formatter

It behaves exactly same as str.format() function. This class becomes useful if you want to subclass it and define your own format string syntax.

Syntax: from string import Formatter

Template

This class is used to create a string template for simpler string substitutions

Syntax: from string import Template

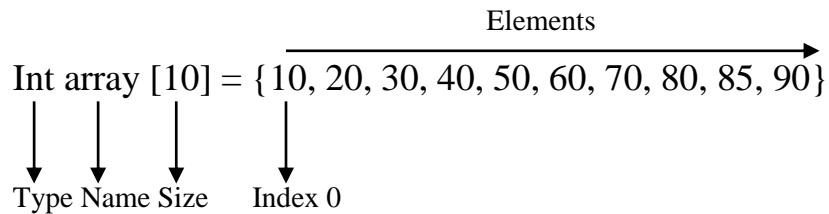
Python arrays:

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- **Element**– Each item stored in an array is called an element.
- **Index** – Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. Below is an illustration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 70

Basic Operations

Following are the basic operations supported by an array.

- Traverse – print all the array elements one by one.
- Insertion – Adds an element at the given index.
- Deletion – Deletes an element at the given index.
- Search – Searches an element using the given index or by the value.
- Update – Updates an element at the given index.

Array is created in Python by importing array module to the python program. Then the array is declared as shown below.

```
from array import *  
  
arrayName=array(typecode, [initializers])
```

Typecode are the codes that are used to define the type of value the array will hold. Some common typecodes used are:

Typecode	Value
b	Represents signed integer of size 1 byte
B	Represents unsigned integer of size 1 byte

c	Represents character of size 1 byte
i	Represents signed integer of size 2 bytes
l	Represents unsigned integer of size 2 bytes
f	Represents floating point of size 4 bytes
d	Represents floating point of size 8 bytes

Creating an array:

```
from array import *  
array1 = array('i', [10,20,30,40,50])  
for x in array1:  
    print(x)
```

Output:

```
>>>
```

RESTART: C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/arr.py

```
10  
20  
30  
40  
50
```

Access the elements of an Array:**Accessing Array Element**

We can access each element of an array using the index of the element.

```
from array import *  
array1 = array('i', [10,20,30,40,50])  
print (array1[0])  
print (array1[2])
```

Output:

RESTART: C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/arr2.py

10

30

Array methods:

Python has a set of built-in methods that you can use on lists/arrays.

Method	Description
<u>append()</u>	Adds an element at the end of the list
<u>clear()</u>	Removes all the elements from the list
<u>copy()</u>	Returns a copy of the list
<u>count()</u>	Returns the number of elements with the specified value
<u>extend()</u>	Add the elements of a list (or any iterable), to the end of the current list
<u>index()</u>	Returns the index of the first element with the specified value
<u>insert()</u>	Adds an element at the specified position
<u>pop()</u>	Removes the element at the specified position
<u>remove()</u>	Removes the first item with the specified value
<u>reverse()</u>	Reverses the order of the list
<u>sort()</u>	Sorts the list

Note: Python does not have built-in support for Arrays, but Python Lists can be used instead.

Example:

```
>>> college=["mrcet","it","cse"]

>>> college.append("autonomous")

>>> college

['mrcet', 'it', 'cse', 'autonomous']

>>> college.append("eee")

>>> college.append("ece")

>>> college

['mrcet', 'it', 'cse', 'autonomous', 'eee', 'ece']

>>> college.pop()

'ece'

>>> college

['mrcet', 'it', 'cse', 'autonomous', 'eee']

>>> college.pop(4)

'eee'

>>> college

['mrcet', 'it', 'cse', 'autonomous']

>>> college.remove("it")

>>> college

['mrcet', 'cse', 'autonomous']
```


UNIT – IV**LISTS, TUPLES, DICTIONARIES**

Lists: list operations, list slices, list methods, list loop, mutability, aliasing, cloning lists, list parameters, list comprehension; **Tuples:** tuple assignment, tuple as return value, tuple comprehension; **Dictionaries:** operations and methods, comprehension;

Lists, Tuples, Dictionaries:**List:**

- It is a general purpose most widely used in data structures
- List is a collection which is ordered and changeable and allows duplicate members. (Grow and shrink as needed, sequence type, sortable).
- To use a list, you must declare it first. Do this using square brackets and separate values with commas.
- We can construct / create list in many ways.

Ex:

```
>>> list1=[1,2,3,'A','B',7,8,[10,11]]
```

```
>>> print(list1)
```

```
[1, 2, 3, 'A', 'B', 7, 8, [10, 11]]
```

```
-----
```

```
>>> x=list()
```

```
>>> x
```

```
[]
```

```
-----
```

```
>>> tuple1=(1,2,3,4)
```

```
>>> x=list(tuple1)
```

```
>>> x
```

```
[1, 2, 3, 4]
```

List operations:

These operations include indexing, slicing, adding, multiplying, and checking for membership

Basic List Operations:

Lists respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1, 2, 3]: print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input –

L = ['mrcet', 'college', 'MRCET!']

Python Expression	Results	Description
L[2]	MRCET	Offsets start at zero

L[-2]	college	Negative: count from the right
L[1:]	['college', 'MRCET!']	Slicing fetches sections

List slices:

```
>>> list1=range(1,6)
>>> list1
range(1, 6)
>>> print(list1)
range(1, 6)
>>> list1=[1,2,3,4,5,6,7,8,9,10]
>>> list1[1:]
[2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list1[:1]
[1]
>>> list1[2:5]
[3, 4, 5]
>>> list1[:6]
[1, 2, 3, 4, 5, 6]
>>> list1[1:2:4]
[2]
>>> list1[1:8:2]
[2, 4, 6, 8]
```

List methods:

The list data type has some more methods. Here are all of the methods of list objects:

- Del()

- Append()
- Extend()
- Insert()
- Pop()
- Remove()
- Reverse()
- Sort()

Delete: Delete a list or an item from a list

```
>>> x=[5,3,8,6]
>>> del(x[1])      #deletes the index position 1 in a list
>>> x
[5, 8, 6]
-----
>>> del(x)
>>> x              # complete list gets deleted
```

Append: Append an item to a list

```
>>> x=[1,5,8,4]
>>> x.append(10)
>>> x
[1, 5, 8, 4, 10]
```

Extend: Append a sequence to a list.

```
>>> x=[1,2,3,4]
>>> y=[3,6,9,1]
>>> x.extend(y)
>>> x
[1, 2, 3, 4, 3, 6, 9, 1]
```

Insert: To add an item at the specified index, use the insert () method:

```
>>> x=[1,2,4,6,7]
```

```
>>> x.insert(2,10) #insert(index no, item to be inserted)
```

```
>>> x
```

```
[1, 2, 10, 4, 6, 7]
```

```
-----
```

```
>>> x.insert(4,['a',11])
```

```
>>> x
```

```
[1, 2, 10, 4, ['a', 11], 6, 7]
```

Pop: The pop() method removes the specified index, (or the last item if index is not specified) or simply pops the last item of list and returns the item.

```
>>> x=[1, 2, 10, 4, 6, 7]
```

```
>>> x.pop()
```

```
7
```

```
>>> x
```

```
[1, 2, 10, 4, 6]
```

```
-----
```

```
>>> x=[1, 2, 10, 4, 6]
```

```
>>> x.pop(2)
```

```
10
```

```
>>> x
```

```
[1, 2, 4, 6]
```

Remove: The remove() method removes the specified item from a given list.

```
>>> x=[1,33,2,10,4,6]
```

```
>>> x.remove(33)
```

```
>>> x
```

```
[1, 2, 10, 4, 6]
```

```
>>> x.remove(4)
```

```
>>> x
```

```
[1, 2, 10, 6]
```

Reverse: Reverse the order of a given list.

```
>>> x=[1,2,3,4,5,6,7]
```

```
>>> x.reverse()
```

```
>>> x
```

```
[7, 6, 5, 4, 3, 2, 1]
```

Sort: Sorts the elements in ascending order

```
>>> x=[7, 6, 5, 4, 3, 2, 1]
```

```
>>> x.sort()
```

```
>>> x
```

```
[1, 2, 3, 4, 5, 6, 7]
```

```
-----
```

```
>>> x=[10,1,5,3,8,7]
```

```
>>> x.sort()
```

```
>>> x
```

```
[1, 3, 5, 7, 8, 10]
```

List loop:

Loops are control structures used to repeat a given section of code a certain number of times or until a particular condition is met.

Method #1: For loop

#list of items

```
list = ['M','R','C','E','T']
```

```
i = 1
```

```
#Iterating over the list
for item in list:
    print ('college ',i,' is ',item)
    i = i+1
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/lis.py

```
college 1 is M
college 2 is R
college 3 is C
college 4 is E
college 5 is T
```

Method #2: For loop and range()

In case we want to use the traditional for loop which iterates from number x to number y.

Python3 code to iterate over a list

```
list = [1, 3, 5, 7, 9]
```

```
# getting length of list
length = len(list)
```

```
# Iterating the index
# same as 'for i in range(len(list))'
for i in range(length):
    print(list[i])
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/listloop.py

```
1
3
5
7
9
```

Method #3: using while loop

Python3 code to iterate over a list

```
list = [1, 3, 5, 7, 9]
```

```
# Getting length of list
```

```
length = len(list)
i = 0
```

```
# Iterating using while loop
while i < length:
    print(list[i])
    i += 1
```

Mutability:

A mutable object can be changed after it is created, and an immutable object can't.

Append: Append an item to a list

```
>>> x=[1,5,8,4]
>>> x.append(10)
>>> x
[1, 5, 8, 4, 10]
```

Extend: Append a sequence to a list.

```
>>> x=[1,2,3,4]
>>> y=[3,6,9,1]
>>> x.extend(y)
>>> x
```

Delete: Delete a list or an item from a list

```
>>> x=[5,3,8,6]
>>> del(x[1])      #deletes the index position 1 in a list
>>> x
[5, 8, 6]
```

Insert: To add an item at the specified index, use the insert () method:

```
>>> x=[1,2,4,6,7]
>>> x.insert(2,10) #insert(index no, item to be inserted)
```



```
>>> x
```

```
[1, 2, 10, 4, 6, 7]
```

```
-----
```

```
>>> x.insert(4,['a',11])
```

```
>>> x
```

```
[1, 2, 10, 4, ['a', 11], 6, 7]
```

Pop: The pop() method removes the specified index, (or the last item if index is not specified) or simply pops the last item of list and returns the item.

```
>>> x=[1, 2, 10, 4, 6, 7]
```

```
>>> x.pop()
```

```
7
```

```
>>> x
```

```
[1, 2, 10, 4, 6]
```

```
-----
```

```
>>> x=[1, 2, 10, 4, 6]
```

```
>>> x.pop(2)
```

```
10
```

```
>>> x
```

```
[1, 2, 4, 6]
```

Remove: The remove() method removes the specified item from a given list.

```
>>> x=[1,33,2,10,4,6]
```

```
>>> x.remove(33)
```

```
>>> x
```

```
[1, 2, 10, 4, 6]
```

```
>>> x.remove(4)
```

```
>>> x
```

```
[1, 2, 10, 6]
```

Reverse: Reverse the order of a given list.

```
>>> x=[1,2,3,4,5,6,7]
```

```
>>> x.reverse()
```

```
>>> x
```

```
[7, 6, 5, 4, 3, 2, 1]
```

Sort: Sorts the elements in ascending order

```
>>> x=[7, 6, 5, 4, 3, 2, 1]
```

```
>>> x.sort()
```

```
>>> x
```

```
[1, 2, 3, 4, 5, 6, 7]
```

```
-----
```

```
>>> x=[10,1,5,3,8,7]
```

```
>>> x.sort()
```

```
>>> x
```

```
[1, 3, 5, 7, 8, 10]
```

Aliasing:

1. An alias is a second name for a piece of data, often easier (and more useful) than making a copy.
2. If the data is immutable, aliases don't matter because the data can't change.
3. But if data can change, aliases can result in lot of hard – to – find bugs.
4. Aliasing happens whenever one variable's value is assigned to another variable.

For ex:

```
a = [81, 82, 83]
```

```
b = [81, 82, 83]
print(a == b)
print(a is b)
b = a
print(a == b)
print(a is b)
b[0] = 5
print(a)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/alia.py
True
False
True
True
[5, 82, 83]
```

Because the same list has two different names, a and b, we say that it is **aliased**. Changes made with one alias affect the other. In the example above, you can see that a and b refer to the same list after executing the assignment statement b = a.

Cloning Lists:

If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference. This process is sometimes called cloning, to avoid the ambiguity of the word copy.

The easiest way to clone a list is to use the slice operator. Taking any slice of a creates a new list. In this case the slice happens to consist of the whole list.

Example:

```
a = [81, 82, 83]

b = a[:]    # make a clone using slice

print(a == b)

print(a is b)

b[0] = 5

print(a)

print(b)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/clo.py

True

False

[81, 82, 83]

[5, 82, 83]

Now we are free to make changes to b without worrying about a

List parameters:

Passing a list as an argument actually passes a reference to the list, not a copy of the list. Since lists are mutable, changes made to the elements referenced by the parameter change the same list that the argument is referencing.

for example, the function below takes a list as an argument and multiplies each element in the list by 2:

```
def doubleStuff(List):
```

```
    """ Overwrite each element in aList with double its value. """
```

```
    for position in range(len(List)):
```

```
        List[position] = 2 * List[position]
```

```
things = [2, 5, 9]
```

```
print(things)
```

```
doubleStuff(things)
```

```
print(things)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/lipar.py ==

[2, 5, 9]

[4, 10, 18]

List comprehension:**List:**

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
>>> list1=[]

>>> for x in range(10):

    list1.append(x**2)

>>> list1

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

(or)

This is also equivalent to

```
>>> list1=list(map(lambda x:x**2, range(10)))

>>> list1

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

(or)

Which is more concise and readable.

```
>>> list1=[x**2 for x in range(10)]

>>> list1

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Similarly some examples:

```
>>> x=[m for m in range(8)]
>>> print(x)
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
>>> x=[z**2 for z in range(10) if z>4]
>>> print(x)
[25, 36, 49, 64, 81]
```

```
>>> x=[x ** 2 for x in range (1, 11) if x % 2 == 1]
>>> print(x)
[1, 9, 25, 49, 81]
```

```
>>> a=5
>>> table = [[a, b, a * b] for b in range(1, 11)]
>>> for i in table:
    print(i)
```

```
[5, 1, 5]
[5, 2, 10]
[5, 3, 15]
[5, 4, 20]
[5, 5, 25]
[5, 6, 30]
[5, 7, 35]
[5, 8, 40]
[5, 9, 45]
[5, 10, 50]
```

Tuples:

A tuple is a collection which is ordered and unchangeable. In Python tuples are written with round brackets.

- Supports all operations for sequences.
- Immutable, but member objects may be mutable.
- If the contents of a list shouldn't change, use a tuple to prevent items from

accidentally being added, changed, or deleted.

- Tuples are more efficient than list due to python's implementation.

We can construct tuple in many ways:

```
X=() #no item tuple
```

```
X=(1,2,3)
```

```
X=tuple(list1)
```

```
X=1,2,3,4
```

Example:

```
>>> x=(1,2,3)
```

```
>>> print(x)
```

```
(1, 2, 3)
```

```
>>> x
```

```
(1, 2, 3)
```

```
-----
```

```
>>> x=()
```

```
>>> x
```

```
()
```

```
-----
```

```
>>> x=[4,5,66,9]
```

```
>>> y=tuple(x)
```

```
>>> y
```

```
(4, 5, 66, 9)
```

```
-----
```

```
>>> x=1,2,3,4
```

```
>>> x
```

```
(1, 2, 3, 4)
```

Some of the operations of tuple are:

- Access tuple items
- Change tuple items
- Loop through a tuple
- Count()
- Index()
- Length()

Access tuple items: Access tuple items by referring to the index number, inside square brackets

```
>>> x=('a','b','c','g')
```

```
>>> print(x[2])
```

c

Change tuple items: Once a tuple is created, you cannot change its values. Tuples are unchangeable.

```
>>> x=(2,5,7,'4',8)
```

```
>>> x[1]=10
```

Traceback (most recent call last):

File "<pyshell#41>", line 1, in <module>

x[1]=10

TypeError: 'tuple' object does not support item assignment

```
>>> x
```

```
(2, 5, 7, '4', 8) # the value is still the same
```

Loop through a tuple: We can loop the values of tuple using for loop

```
>>> x=4,5,6,7,2,'aa'
```

```
>>> for i in x:
```

```
    print(i)
```

4

5

6

7

2

aa

Count (): Returns the number of times a specified value occurs in a tuple

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
```

```
>>> x.count(2)
```

4

Index (): Searches the tuple for a specified value and returns the position of where it was found


```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> x.index(2)
1
```

(Or)

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=x.index(2)
>>> print(y)
1
```

Length (): To know the number of items or values present in a tuple, we use len().

```
>>> x=(1,2,3,4,5,6,2,10,2,11,12,2)
>>> y=len(x)
>>> print(y)
12
```

Tuple Assignment

Python has tuple assignment feature which enables you to assign more than one variable at a time. In here, we have assigned tuple 1 with the college information like college name, year, etc. and another tuple 2 with the values in it like number (1, 2, 3... 7).

For Example,

Here is the code,

- >>> tup1 = ('mrcet', 'eng college','2004','cse', 'it','csit');
- >>> tup2 = (1,2,3,4,5,6,7);
- >>> print(tup1[0])
- mrcet
- >>> print(tup2[1:4])
- (2, 3, 4)

Tuple 1 includes list of information of mrcet

Tuple 2 includes list of numbers in it

We call the value for [0] in tuple and for tuple 2 we call the value between 1 and 4

Run the above code- It gives name mrcet for first tuple while for second tuple it gives number (2, 3, 4)

Tuple as return values:

A Tuple is a comma separated sequence of items. It is created with or without (). Tuples are immutable.

A Python program to return multiple values from a method using tuple

This function returns a tuple

```
def fun():
```

```
    str = "mrcet college"
```

```
    x = 20
```

```
    return str, x; # Return tuple, we could also
```

```
        # write (str, x)
```

Driver code to test above method

```
str, x = fun() # Assign returned tuple
```

```
print(str)
```

```
print(x)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/tupretval.py
```

```
mrcet college
```

```
20
```

- Functions can return tuples as return values.

```
def circleInfo(r):
```

```
    """ Return (circumference, area) of a circle of radius r """
```

```
    c = 2 * 3.14159 * r
```

```
    a = 3.14159 * r * r
```

```
    return (c, a)
```

```
print(circleInfo(10))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/functupretval.py
```

```
(62.8318, 314.159)
```

```
def f(x):  
    y0 = x + 1  
    y1 = x * 3  
    y2 = y0 ** y3  
    return (y0, y1, y2)
```

Tuple comprehension:

Tuple Comprehensions are special: The result of a tuple comprehension is special. You might expect it to produce a tuple, but what it does is produce a special "generator" object that we can iterate over.

For example:

```
>>> x = (i for i in 'abc') #tuple comprehension  
>>> x  
<generator object <genexpr> at 0x033EEC30>  
  
>>> print(x)  
<generator object <genexpr> at 0x033EEC30>
```

You might expect this to print as ('a', 'b', 'c') but it prints as <generator object <genexpr> at 0x02AAD710> The result of a tuple comprehension is not a tuple: it is actually a generator. The only thing that you need to know now about a generator now is that you can iterate over it, but ONLY ONCE.

So, given the code

```
>>> x = (i for i in 'abc')  
>>> for i in x:  
    print(i)
```

a
b
c

Create a list of 2-tuples like (number, square):

```
>>> z=[(x, x**2) for x in range(6)]  
>>> z  
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

Set:

Similarly to list comprehensions, set comprehensions are also supported:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

```
>>> x={3*x for x in range(10) if x>5}
>>> x
{24, 18, 27, 21}
```

Dictionaries:

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

- Key-value pairs
- Unordered

We can construct or create dictionary like:

```
X={1:'A',2:'B',3:'c'}
X=dict([('a',3) ('b',4)])
X=dict('A'=1,'B'=2)
```

Example:

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1
{'brand': 'mrcet', 'model': 'college', 'year': 2004}
```

Operations and methods:

Methods that are available with dictionary are tabulated below. Some of them have already been used in the above examples.

Method	Description
clear()	Remove all items form the dictionary.

<code>copy()</code>	Return a shallow copy of the dictionary.
<code>fromkeys(seq[, v])</code>	Return a new dictionary with keys from seq and value equal to v (defaults to None).
<code>get(key[,d])</code>	Return the value of key. If key doesnot exit, return d (defaults to None).
<code>items()</code>	Return a new view of the dictionary's items (key, value).
<code>keys()</code>	Return a new view of the dictionary's keys.
<code>pop(key[,d])</code>	Remove the item with key and return its value or d if key is not found. If d is not provided and key is not found, raises <code>KeyError</code> .
<code>popitem()</code>	Remove and return an arbitrary item (key, value). Raises <code>KeyError</code> if the dictionary is empty.
<code>setdefault(key[,d])</code>	If key is in the dictionary, return its value. If not, insert key with a value of d and return d (defaults to None).
<code>update([other])</code>	Update the dictionary with the key/value pairs from other, overwriting existing keys.
<code>values()</code>	Return a new view of the dictionary's values

Below are some dictionary operations:

To access specific value of a dictionary, we must pass its key,

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> x=dict1["brand"]
>>> x
'mrcet'
```

To access keys and values and items of dictionary:

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1.keys()
dict_keys(['brand', 'model', 'year'])
>>> dict1.values()
dict_values(['mrcet', 'college', 2004])
>>> dict1.items()
dict_items([('brand', 'mrcet'), ('model', 'college'), ('year', 2004)])
```

```
>>> for items in dict1.values():
    print(items)
```

```
mrcet
college
2004
```

```
>>> for items in dict1.keys():
    print(items)
```

```
brand
model
year
```

```
>>> for i in dict1.items():
    print(i)
```

```
('brand', 'mrcet')
('model', 'college')
('year', 2004)
```

Some more operations like:

- Add/change

- Remove
- Length
- Delete

Add/change values: You can change the value of a specific item by referring to its key name

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> dict1["year"]=2005
>>> dict1
{'brand': 'mrcet', 'model': 'college', 'year': 2005}
```

Remove(): It removes or pop the specific item of dictionary.

```
>>> dict1 = {"brand":"mrcet","model":"college","year":2004}
>>> print(dict1.pop("model"))
college
>>> dict1
{'brand': 'mrcet', 'year': 2005}
```

Delete: Deletes a particular item.

```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> del x[5]
>>> x
```

Length: we use len() method to get the length of dictionary.

```
>>>{1: 1, 2: 4, 3: 9, 4: 16}
{1: 1, 2: 4, 3: 9, 4: 16}
>>> y=len(x)
>>> y
4
```

Iterating over (key, value) pairs:

```
>>> x = {1:1, 2:4, 3:9, 4:16, 5:25}
>>> for key in x:
    print(key, x[key])
```

```
1 1
2 4
3 9
```

```
4 16
5 25
>>> for k,v in x.items():
    print(k,v)
```

```
1 1
2 4
3 9
4 16
5 25
```

List of Dictionaries:

```
>>> customers = [{"uid":1,"name":"John"},
    {"uid":2,"name":"Smith"},
    {"uid":3,"name":"Andersson"},
    ]
>>> >>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'Andersson'}]
```

```
## Print the uid and name of each customer
>>> for x in customers:
    print(x["uid"], x["name"])
```

```
1 John
2 Smith
3 Andersson
```

```
## Modify an entry, This will change the name of customer 2 from Smith to Charlie
>>> customers[2]["name"]="charlie"
>>> print(customers)
[{'uid': 1, 'name': 'John'}, {'uid': 2, 'name': 'Smith'}, {'uid': 3, 'name': 'charlie'}]
```

```
## Add a new field to each entry
```

```
>>> for x in customers:
    x["password"]="123456" # any initial value
```

```
>>> print(customers)
```



```
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 2, 'name': 'Smith', 'password': '123456'}, {'uid': 3, 'name': 'charlie', 'password': '123456'}]
```

Delete a field

```
>>> del customers[1]
```

```
>>> print(customers)
```

```
[{'uid': 1, 'name': 'John', 'password': '123456'}, {'uid': 3, 'name': 'charlie', 'password': '123456'}]
```

```
>>> del customers[1]
```

```
>>> print(customers)
```

```
[{'uid': 1, 'name': 'John', 'password': '123456'}]
```

Delete all fields

```
>>> for x in customers:
```

```
    del x["uid"]
```

```
>>> x
```

```
{'name': 'John', 'password': '123456'}
```

Comprehension:

Dictionary comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
>>> z={x: x**2 for x in (2,4,6)}
```

```
>>> z
```

```
{2: 4, 4: 16, 6: 36}
```

```
>>> dict11 = {x: x*x for x in range(6)}
```

```
>>> dict11
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

UNIT – V**FILES, EXCEPTIONS, MODULES, PACKAGES**

Files and exception: text files, reading and writing files, command line arguments, errors and exceptions, handling exceptions, modules (datetime, time, OS , calendar, math module), Explore packages.

Files, Exceptions, Modules, Packages:**Files and exception:**

A **file** is some information or data which stays in the computer storage devices. Python gives you easy ways to manipulate these files. Generally files divide in two categories, text file and binary file. Text files are simple text where as the binary files contain binary data which is only readable by computer.

- **Text files:** In this type of file, Each line of text is terminated with a special character called EOL (End of Line), which is the new line character ('\n') in python by default.
- **Binary files:** In this type of file, there is no terminator for a line and the data is stored after converting it into machine understandable binary language.

An **exception** is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions. In general, when a Python script encounters a situation that it cannot cope with, it raises an exception. An exception is a Python object that represents an error.

Text files:

We can create the text files by using the syntax:

Variable name=open (“file.txt”, file mode)

For ex: f= open ("hello.txt","w+")

- We declared the variable f to open a file named hello.txt. **Open** takes 2 arguments, the file that we want to open and a string that represents the kinds of permission or operation we want to do on the file
- Here we used "w" letter in our argument, which indicates write and the plus sign that means it will create a file if it does not exist in library

- The available option beside "w" are "r" for read and "a" for append and plus sign means if it is not there then create it

File Modes in Python:

Mode	Description
'r'	This is the default mode. It Opens file for reading.
'w'	This Mode Opens file for writing. If file does not exist, it creates a new file. If file exists it truncates the file.
'x'	Creates a new file. If file already exists, the operation fails.
'a'	Open file in append mode. If file does not exist, it creates a new file.
't'	This is the default mode. It opens in text mode.
'b'	This opens in binary mode.
'+'	This will open a file for reading and writing (updating)

Reading and Writing files:

The following image shows how to create and open a text file in notepad from command prompt

```
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

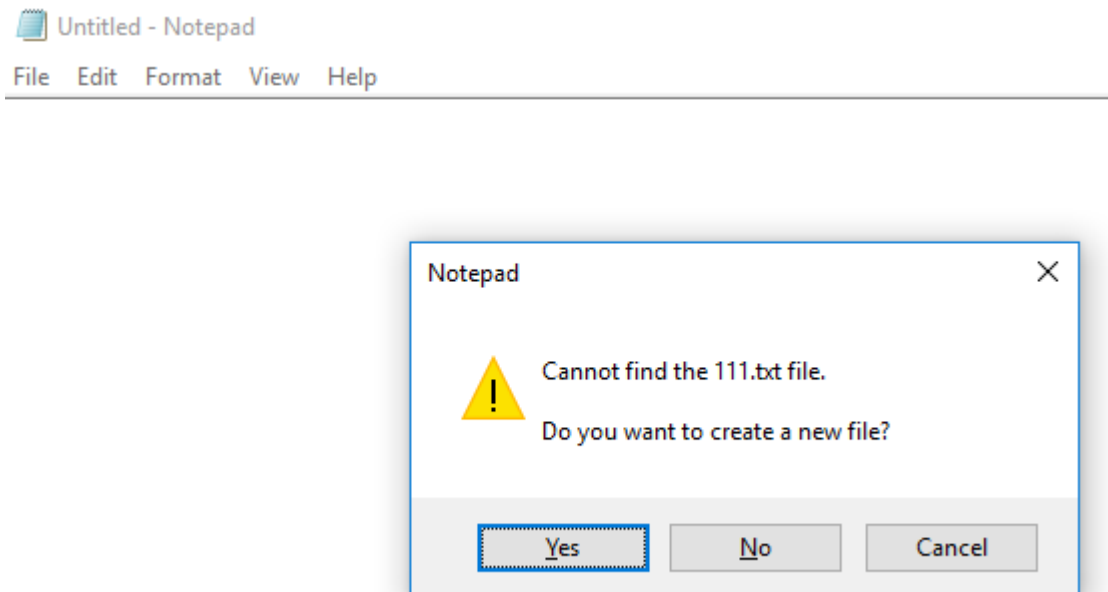
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>start notepad hello.txt

C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type hello.txt
Hello mrcet
good morning
how r u
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>
```

(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>notepad 111.txt
```

Hit on enter then it shows the following whether to open or not?



Click on “yes” to open else “no” to cancel

Write a python program to open and read a file

```
a=open(“one.txt”,”r”)
```

```
print(a.read())
```

```
a.close()
```

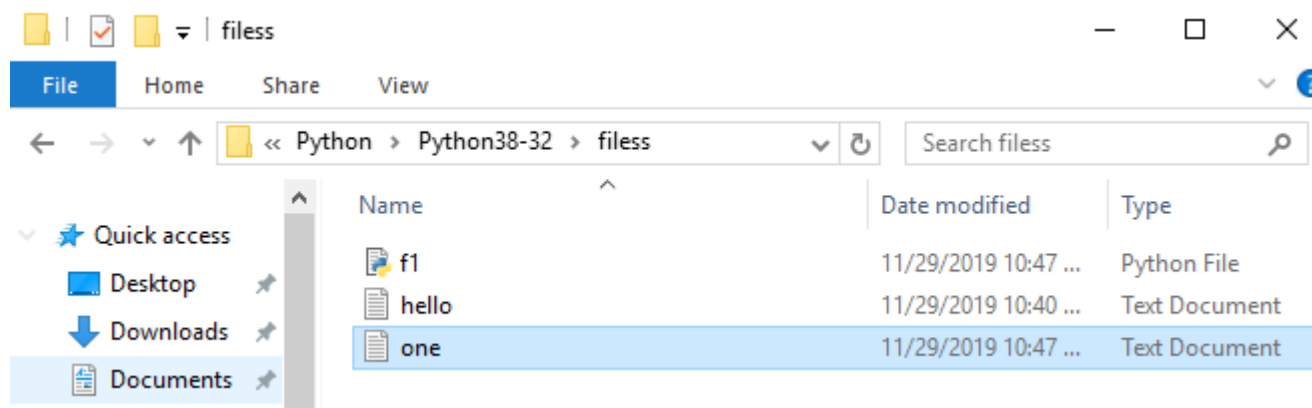
Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/files/f1.py  
welcome to python programming
```

(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>python f1.py  
welcome to python programming
```

Note: All the program files and text files need to be saved together in a particular file then only the program performs the operations in the given file mode



f.close() ---- This will close the instance of the file somefile.txt stored

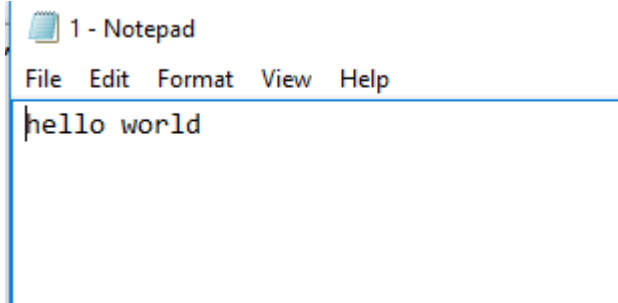
Write a python program to open and write “hello world” into a file?

```
f=open("1.txt","a")
```

```
f.write("hello world")
```

```
f.close()
```

Output:



(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type 1.txt  
hello world
```

Note: In the above program the 1.txt file is created automatically and adds hello world into txt file

If we keep on executing the same program for more than one time then it append the data that many times

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type 1.txt  
hello worldhello world
```

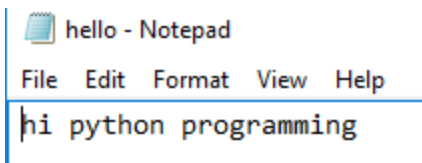
Write a python program to write the content “hi python programming” for the existing file.

```
f=open("1.txt",'w')
```

```
f.write("hi python programming")
```

```
f.close()
```

Output:



In the above program the hello.txt file consist of data like

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\filess>type hello.txt  
Hello mrcet  
good morning  
how r u
```

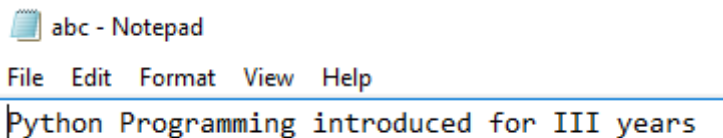
But when we try to write some data on to the same file it overwrites and saves with the current data (check output)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\files>type hello.txt
hi python programming
```

Write a python program to open and write the content to file and read it.

```
fo=open("abc.txt","w+")
fo.write("Python Programming")
print(fo.read())
fo.close()
```

Output:



abc - Notepad

File Edit Format View Help

Python Programming introduced for III years

(or)

```
C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\files>type abc.txt
Python Programming introduced for III years
```

Note: It creates the abc.txt file automatically and writes the data into it



Command line arguments:

The command line arguments must be given whenever we want to give the input before the start of the script, while on the other hand, `raw_input()` is used to get the input while the python program / script is running.

The command line arguments in python can be processed by using either 'sys' module, 'argparse' module and 'getopt' module.

'sys' module :

Python sys module stores the command line arguments into a list, we can access it using `sys.argv`. This is very useful and simple way to read command line arguments as String.

sys.argv is the list of commandline arguments passed to the Python program. `argv` represents all the items that come along via the command line input, it's basically an array holding the command line arguments of our program

```
>>> sys.modules.keys() -- this prints so many dict elements in the form of list.
```

```
# Python code to demonstrate the use of 'sys' module for command line arguments
```

```
import sys
```

```
# command line arguments are stored in the form
```

```
# of list in sys.argv
```

```
argumentList = sys.argv
```

```
print(argumentList)
```

```
# Print the name of file
```

```
print(sys.argv[0])
```

```
# Print the first argument after the name of file
```

```
#print(sys.argv[1])
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/cmdnlinarg.py
```

```
['C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/cmdnlinarg.py']
```

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/cmdnlinarg.py
```


Note: Since my list consist of only one element at '0' index so it prints only that list element, if we try to access at index position '1' then it shows the error like,

IndexError: list index out of range

```
import sys

print(type(sys.argv))
print('The command line arguments are:')
for i in sys.argv:
    print(i)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/symod.py ==
<class 'list'>
The command line arguments are:
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/symod.py
```

write a python program to get python version.

```
import sys
print("System version is:")
print(sys.version)
print("Version Information is:")
print(sys.version_info)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/s1.py =
System version is:
3.8.0 (tags/v3.8.0:fa919fd, Oct 14 2019, 19:21:23) [MSC v.1916 32 bit (Intel)]
Version Information is:
sys.version_info(major=3, minor=8, micro=0, releaselevel='final', serial=0)
```

‘argparse’ module :

Python getopt module is very similar in working as the C getopt() function for parsing command-line parameters. Python getopt module is useful in parsing command line arguments where we want user to enter some options too.

```
>>> parser = argparse.ArgumentParser(description='Process some integers.')
```

```
#-----
```

```
import argparse

parser = argparse.ArgumentParser()
print(parser.parse_args())
```

‘getopt’ module :

Python argparse module is the preferred way to parse command line arguments. It provides a lot of option such as positional arguments, default value for arguments, help message, specifying data type of argument etc

It parses the command line options and parameter list. The signature of this function is mentioned below:

```
getopt.getopt(args, shortopts, longopts=[ ])
```

- args are the arguments to be passed.
- shortopts is the options this script accepts.
- Optional parameter, longopts is the list of String parameters this function accepts which should be supported. Note that the -- should not be prepended with option names.

```
-h  ----- print help and usage message
-m  ----- accept custom option value
-d  ----- run the script in debug mode
```

```
import getopt
import sys
```

```
argv = sys.argv[0:]
try:
    opts, args = getopt.getopt(argv, 'hm:d', ['help', 'my_file='])
    #print(opts)
    print(args)
except getopt.GetoptError:
    # Print a message or do something useful
    print('Something went wrong!')
    sys.exit(2)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/gtopt.py ==

['C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/gtopt.py']

Errors and Exceptions:

Python Errors and Built-in Exceptions: Python (interpreter) raises exceptions when it encounters **errors**. When writing a program, we, more often than not, will encounter errors. Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error

ZeroDivisionError:

ZeroDivisionError in Python indicates that the second argument used in a division (or modulo) operation was zero.

OverflowError:

OverflowError in Python indicates that an arithmetic operation has exceeded the limits of the current Python runtime. This is typically due to excessively large float values, as integer values that are too big will opt to raise memory errors instead.

ImportError:

It is raised when you try to import a module which does not exist. This may happen if you made a typing mistake in the module name or the module doesn't exist in its standard path. In the example below, a module named "non_existing_module" is being imported but it doesn't exist, hence an import error exception is raised.

IndexError:

An IndexError exception is raised when you refer a sequence which is out of range. In the example below, the list abc contains only 3 entries, but the 4th index is being accessed, which will result an IndexError exception.

TypeError:

When two unrelated type of objects are combined, TypeErrorexception is raised. In example below, an int and a string is added, which will result in TypeError exception.

IndentationError:

Unexpected indent. As mentioned in the "expected an indented block" section, Python not only insists on indentation, it insists on consistent indentation. You are free to choose the number of spaces of indentation to use, but you then need to stick with it.

Syntax errors:

These are the most basic type of error. They arise when the Python parser is unable to understand a line of code. Syntax errors are almost always fatal, i.e. there is almost never a way to successfully execute a piece of code containing syntax errors.

Run-time error:

A run-time error happens when Python understands what you are saying, but runs into trouble when following your instructions.

Key Error :

Python raises a `KeyError` whenever a `dict()` object is requested (using the format `a = adict[key]`) and the key is not in the dictionary.

Value Error:

In Python, a value is the information that is stored within a certain object. To encounter a `ValueError` in Python means that is a problem with the content of the object you tried to assign the value to.

Python has many built-in exceptions which forces your program to output an error when something in it goes wrong. In Python, users can define such exceptions by creating a new class. This exception class has to be derived, either directly or indirectly, from `Exception` class.

Different types of exceptions:

- `ArrayIndexOutOfBoundsException`.
- `ClassNotFoundException`.
- `FileNotFoundException`.
- `IOException`.
- `InterruptedException`.
- `NoSuchFieldException`.
- `NoSuchMethodException`

Handling Exceptions:

The cause of an exception is often external to the program itself. For example, an incorrect input, a malfunctioning IO device etc. Because the program abruptly terminates on encountering an exception, it may cause damage to system resources, such as files. Hence, the exceptions should be properly handled so that an abrupt termination of the program is prevented.

Python uses try and except keywords to handle exceptions. Both keywords are followed by indented blocks.

Syntax:

try :

 #statements in try block

except :

 #executed when error in try block

Typically we see, most of the times

- **Syntactical errors** (wrong spelling, colon (:) missing),
At developer level and compile level it gives errors.
- **Logical errors** (2+2=4, instead if we get output as 3 i.e., wrong output),
As a developer we test the application, during that time logical error may obtained.
- **Run time error** (In this case, if the user doesn't know to give input, 5/6 is ok but if the user say 6 and 0 i.e., 6/0 (shows error a number cannot be divided by zero))
This is not easy compared to the above two errors because it is not done by the system, it is (mistake) done by the user.

The things we need to observe are:

1. You should be able to understand the mistakes; the error might be done by user, DB connection or server.
2. Whenever there is an error execution should not stop.
Ex: Banking Transaction
3. The aim is execution should not stop even though an error occurs.

For ex:

```
a=5
```

```
b=2
```

```
print(a/b)
```

```
print("Bye")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex1.py
```

```
2.5
```

```
Bye
```

- The above is normal execution with no error, but if we say when $b=0$, it is a critical and gives error, see below

```
a=5
```

```
b=0
```

```
print(a/b)
```

```
print("bye") #this has to be printed, but abnormal termination
```

Output:

Traceback (most recent call last):

```
File "C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex2.py", line 3, in <module>
```

```
    print(a/b)
```

ZeroDivisionError: division by zero

- To overcome this we handle exceptions using except keyword

```
a=5
```

```
b=0
```

```
try:  
    print(a/b)  
except Exception:  
    print("number can not be divided by zero")  
    print("bye")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex3.py

number can not be divided by zero

bye

- The except block executes only when try block has an error, check it below

a=5

b=2

```
try:
```

```
    print(a/b)
```

```
except Exception:
```

```
    print("number can not be divided by zero")
```

```
    print("bye")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex4.py

2.5

- For example if you want to print the message like what is an error in a program then we use “e” which is the representation or object of an exception.

a=5

b=0

```
try:
```

```
print(a/b)
```

except Exception as e:

```
    print("number can not be divided by zero",e)
```

```
print("bye")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex5.py

number can not be divided by zero **division by zero**

bye



(Type of error)

Let us see some more examples:

I don't want to print bye but I want to close the file whenever it is opened.

```
a=5
```

```
b=2
```

```
try:
```

```
    print("resource opened")
```

```
    print(a/b)
```

```
    print("resource closed")
```

except Exception as e:

```
    print("number can not be divided by zero",e)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex6.py

resource opened

2.5

resource closed

- **Note:** the file is opened and closed well, but see by changing the value of b to 0,

```
a=5
```

```
b=0
```

```
try:
```

```
    print("resource opened")
```

```
    print(a/b)
```

```
    print("resource closed")
```

```
except Exception as e:
```

```
    print("number can not be divided by zero",e)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex7.py
```

```
resource opened
```

```
number can not be divided by zero division by zero
```

- **Note:** resource not closed
- **To overcome this, keep print("resource closed") in except block, see it**

```
a=5
```

```
b=0
```

```
try:
```

```
    print("resource opened")
```

```
    print(a/b)
```

```
except Exception as e:
```

```
    print("number can not be divided by zero",e)
```

```
    print("resource closed")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex8.py

resource opened

number can not be divided by zero division by zero

resource closed

- **The result is fine that the file is opened and closed, but again change the value of b to back (i.e., value 2 or other than zero)**

```
a=5
```

```
b=2
```

```
try:
```

```
    print("resource opened")
```

```
    print(a/b)
```

```
except Exception as e:
```

```
    print("number can not be divided by zero",e)
```

```
    print("resource closed")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex9.py

resource opened

2.5

- **But again the same problem file/resource is not closed**
- **To overcome this python has a feature called **finally:****



This block gets executed though we get an error or not

Note: **Except block executes, only when **try** block has an error, but **finally** block executes, even though you get an exception.**

```
a=5
```

```
b=0
```

```
try:
```

```
print("resource open")
print(a/b)
k=int(input("enter a number"))
print(k)
except ZeroDivisionError as e:
    print("the value can not be divided by zero",e)
finally:
    print("resource closed")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex10.py
resource open
the value can not be divided by zero division by zero
resource closed
```

- **change the value of b to 2 for above program, you see the output like**

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex10.py
resource open
2.5
enter a number 6
6
resource closed
```

- **Instead give input as some character or string for above program, check the output**

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex10.py
resource open
2.5
enter a number p
resource closed
```

Traceback (most recent call last):

File "C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex10.py", line 7, in <module>

```
k=int(input("enter a number"))
```

ValueError: invalid literal for int() with base 10: ' p'

```
#-----  
a=5  
b=0  
  
try:  
    print("resource open")  
    print(a/b)  
    k=int(input("enter a number"))  
    print(k)  
except ZeroDivisionError as e:  
    print("the value can not be divided by zero",e)  
except ValueError as e:  
    print("invalid input")  
except Exception as e:  
    print("something went wrong...",e)  
  
finally:  
    print("resource closed")
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex11.py

resource open

the value can not be divided by zero division by zero

resource closed

- **Change the value of b to 2 and give the input as some character or string (other than int)**

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/ex12.py

resource open

2.5

enter a number p

invalid input

resource closed

Modules (Date, Time, os, calendar, math):

- Modules refer to a file containing Python statements and definitions.

- We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.
- We can define our most used functions in a module and import it, instead of copying their definitions into different programs.
- **Modular programming** refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or **modules**.

Advantages :

- **Simplicity:** Rather than focusing on the entire problem at hand, a module typically focuses on one relatively small portion of the problem. If you're working on a single module, you'll have a smaller problem domain to wrap your head around. This makes development easier and less error-prone.
- **Maintainability:** Modules are typically designed so that they enforce logical boundaries between different problem domains. If modules are written in a way that minimizes interdependency, there is decreased likelihood that modifications to a single module will have an impact on other parts of the program. This makes it more viable for a team of many programmers to work collaboratively on a large application.
- **Reusability:** Functionality defined in a single module can be easily reused (through an appropriately defined interface) by other parts of the application. This eliminates the need to recreate duplicate code.
- **Scoping:** Modules typically define a separate **namespace**, which helps avoid collisions between identifiers in different areas of a program.
- **Functions, modules and packages** are all constructs in Python that promote code modularization.

A file containing Python code, for e.g.: example.py, is called a module and its module name would be example.

```
>>> def add(a,b):  
    result=a+b  
    return result  
    """This program adds two numbers and return the result"""
```

```
example.py - C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/exempl...
File Edit Format Run Options Window Help
def add(a,b):
    """This program adds two numbers and return the result"""
    result=a+b
    return result
```

Here, we have defined a function add() inside a module named example. The function takes in two numbers and returns their sum.

How to import the module is:

- We can import the definitions inside a module to another module or the Interactive interpreter in Python.
- We use the import keyword to do this. To import our previously defined module example we type the following in the Python prompt.
- Using the module name we can access the function using dot (.) operation. For Eg:

```
>>> import example
```

```
>>> example.add(5,5)
```

```
10
```

- Python has a ton of standard modules available. Standard modules can be imported the same way as we import our user-defined modules.

Reloading a module:

```
def hi(a,b):
```

```
    print(a+b)
```

```
hi(4,4)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/add.py
```

```
8
```

```
>>> import add
```

```
8
```

```
>>> import add
```

```
>>> import add
```

```
>>>
```

Python provides a neat way of doing this. We can use the reload() function inside the imp module to reload a module. This is how its done.

- >>> import imp
- >>> import my_module
- This code got executed >>> import my_module >>> imp.reload(my_module) This code got executed <module 'my_module' from '.\\my_module.py'>how its done.

```
>>> import imp
```

```
>>> import add
```

```
>>> imp.reload(add)
```

```
8
```

```
<module 'add' from 'C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy\\add.py'>
```

The dir() built-in function

```
>>> import example
```

```
>>> dir(example)
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'add']
```

```
>>> dir()
```

```
['__annotations__', '__builtins__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '__warningregistry__', 'add', 'example', 'hi', 'imp']
```

It shows all built-in and user-defined modules.

For ex:

```
>>> example.__name__
```

```
'example'
```

Datetime module:

Write a python program to display date, time

```
>>> import datetime
```

```
>>> a=datetime.datetime(2019,5,27,6,35,40)
```

```
>>> a
```

```
datetime.datetime(2019, 5, 27, 6, 35, 40)
```

write a python program to display date

```
import datetime
```

```
a=datetime.date(2000,9,18)
```

```
print(a)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py =
```

```
2000-09-18
```

write a python program to display time

```
import datetime
```

```
a=datetime.time(5,3)
```

```
print(a)
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py =
```

```
05:03:00
```

#write a python program to print date, time for today and now.

```
import datetime
```



```
a=datetime.datetime.today()
b=datetime.datetime.now()
print(a)
print(b)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py =

2019-11-29 12:49:52.235581

2019-11-29 12:49:52.235581

#write a python program to add some days to your present date and print the date added.

```
import datetime
a=datetime.date.today()
b=datetime.timedelta(days=7)
print(a+b)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py =

2019-12-06

#write a python program to print the no. of days to write to reach your birthday

```
import datetime
a=datetime.date.today()
b=datetime.date(2020,5,27)
c=b-a
print(c)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py =

180 days, 0:00:00

#write an python program to print date, time using date and time functions

```
import datetime
```

```
t=datetime.datetime.today()
```

```
print(t.date())
```

```
print(t.time())
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/d1.py =

2019-11-29

12:53:39.226763

Time module:

#write a python program to display time.

```
import time
```

```
print(time.time())
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/t1.py =

1575012547.1584706

#write a python program to get structure of time stamp.

```
import time
```

```
print(time.localtime(time.time()))
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/t1.py =

```
time.struct_time(tm_year=2019, tm_mon=11, tm_mday=29, tm_hour=13, tm_min=1,
tm_sec=15, tm_wday=4, tm_yday=333, tm_isdst=0)
```

#write a python program to make a time stamp.

```
import time

a=(1999,5,27,7,20,15,1,27,0)

print(time.mktime(a))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/t1.py =
927769815.0
```

#write a python program using sleep().

```
import time

time.sleep(6)    #prints after 6 seconds

print("Python Lab")
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/t1.py =
Python Lab (#prints after 6 seconds)
```

os module:

```
>>> import os

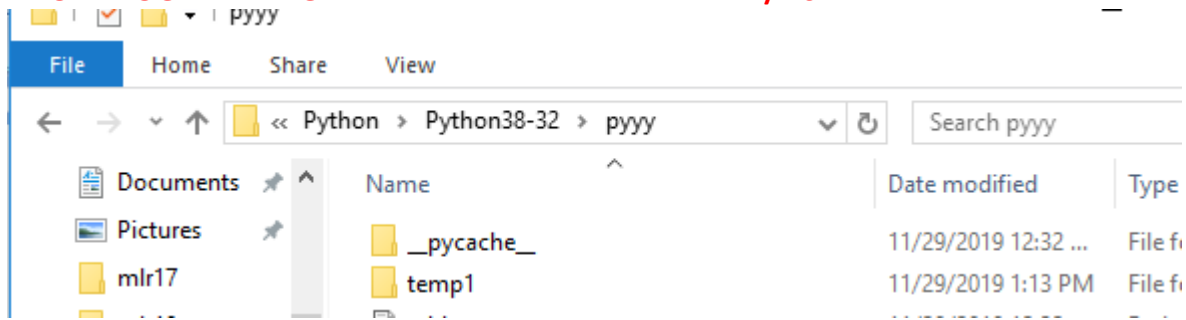
>>> os.name

'nt'

>>> os.getcwd()

'C:\\Users\\MRCET\\AppData\\Local\\Programs\\Python\\Python38-32\\pyyy'

>>> os.mkdir("temp1")
```



Note: temp1 dir is created

```
>>> os.getcwd()
```

```
'C:\\Users\\MRCET\\AppData\\Local\\Programs\\Python\\Python38-32\\pyyy'
```

```
>>> open("t1.py","a")
```

```
<_io.TextIOWrapper name='t1.py' mode='a' encoding='cp1252'>
```

```
>>> os.access("t1.py",os.F_OK)
```

```
True
```

```
>>> os.access("t1.py",os.W_OK)
```

```
True
```

```
>>> os.rename("t1.py","t3.py")
```

```
>>> os.access("t1.py",os.F_OK)
```

```
False
```

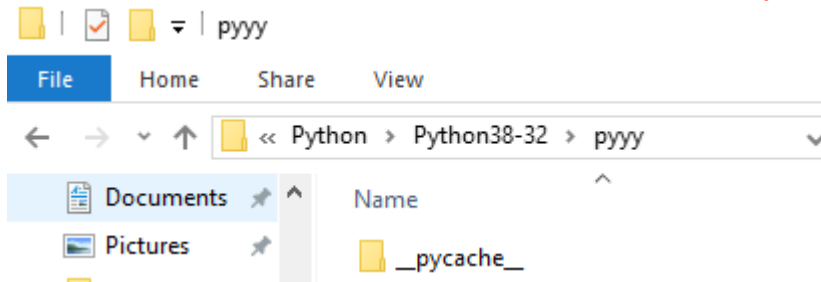
```
>>> os.access("t3.py",os.F_OK)
```

```
True
```

```
>>> os.rmdir('temp1')
```

(or)

```
os.rmdir('C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/temp1')
```



Note: Temp1dir is removed

```
>>> os.remove("t3.py")
```

Note: We can check with the following cmd whether removed or not

```
>>> os.access("t3.py",os.F_OK)
```

False

```
>>> os.listdir()
```

```
['add.py', 'ali.py', 'alia.py', 'arr.py', 'arr2.py', 'arr3.py', 'arr4.py', 'arr5.py', 'arr6.py', 'br.py', 'br2.py', 'bubb.py', 'bubb2.py', 'bubb3.py', 'bubb4.py', 'bubbdesc.py', 'clo.py', 'cmdnlinarg.py', 'comm.py', 'con1.py', 'cont.py', 'cont2.py', 'dl.py', 'dic.py', 'el.py', 'example.py', 'fl.y.py', 'flowof.py', 'fr.py', 'fr2.py', 'fr3.py', 'fu.py', 'fu1.py', 'if1.py', 'if2.py', 'ifelif.py', 'ifelse.py', 'iff.py', 'insertdesc.py', 'inserti.py', 'k1.py', 'l1.py', 'l2.py', 'link1.py', 'linklisttt.py', 'lis.py', 'listlooop.py', 'm1.py', 'merg.py', 'nesforr.py', 'nestedif.py', 'opprec.py', 'paraarg.py', 'qucksort.py', 'qukdsc.py', 'quu.py', 'r.py', 'rec.py', 'ret.py', 'rn.py', 's1.py', 'scoglo.py', 'selecasce.py', 'selectdecs.py', 'stk.py', 'strmodl.py', 'strr.py', 'strr1.py', 'strr2.py', 'strr3.py', 'strr4.py', 'strrmodl.py', 'wh.py', 'wh1.py', 'wh2.py', 'wh3.py', 'wh4.py', 'wh5.py', '__pycache__']
```

```
>>> os.listdir('C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32')
```

```
['argpar.py', 'br.py', 'bu.py', 'cmdnlinarg.py', 'DLLs', 'Doc', 'fl.py', 'fl.txt', 'filess', 'functupretval.py', 'funture.py', 'gtopt.py', 'include', 'Lib', 'libs', 'LICENSE.txt', 'lisparam.py', 'mysite', 'NEWS.txt', 'niru', 'python.exe', 'python3.dll', 'python38.dll', 'pythonw.exe', 'pyyy', 'Scripts', 'srp.py', 'sy.py', 'symod.py', 'tcl', 'the_weather', 'Tools', 'tupretval.py', 'vcruntime140.dll']
```

Calendar module:

#write a python program to display a particular month of a year using calendar module.

```
import calendar  
  
print(calendar.month(2020,1))
```

Output:

```
>>>  
= RESTART: C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/c11.py  
    January 2020  
Mo Tu We Th Fr Sa Su  
      1  2  3  4  5  
 6  7  8  9 10 11 12  
13 14 15 16 17 18 19  
20 21 22 23 24 25 26  
27 28 29 30 31  
,
```

Activate Windows

write a python program to check whether the given year is leap or not.

```
import calendar  
  
print(calendar.isleap(2021))
```

Output:

```
C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/c11.py  
  
False
```

#write a python program to print all the months of given year.

```
import calendar  
  
print(calendar.calendar(2020,1,1,1))
```

Output:

>>>

```
= RESTART: C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32,
2020
```

```

January          February          March
Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su
    1  2  3  4  5              1  2              1
  6  7  8  9 10 11 12    3  4  5  6  7  8  9    2  3  4  5  6  7  8
13 14 15 16 17 18 19   10 11 12 13 14 15 16    9 10 11 12 13 14 15
20 21 22 23 24 25 26   17 18 19 20 21 22 23   16 17 18 19 20 21 22
27 28 29 30 31         24 25 26 27 28 29       23 24 25 26 27 28 29
                                     30 31

April            May                June
Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su
    1  2  3  4  5              1  2  3    1  2  3  4  5  6  7
  6  7  8  9 10 11 12    4  5  6  7  8  9 10    8  9 10 11 12 13 14
13 14 15 16 17 18 19   11 12 13 14 15 16 17   15 16 17 18 19 20 21
20 21 22 23 24 25 26   18 19 20 21 22 23 24   22 23 24 25 26 27 28
27 28 29 30         25 26 27 28 29 30 31   29 30

July             August             September
Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su
    1  2  3  4  5              1  2              1  2  3  4  5  6
  6  7  8  9 10 11 12    3  4  5  6  7  8  9    7  8  9 10 11 12 13
13 14 15 16 17 18 19   10 11 12 13 14 15 16   14 15 16 17 18 19 20
20 21 22 23 24 25 26   17 18 19 20 21 22 23   21 22 23 24 25 26 27
27 28 29 30 31         24 25 26 27 28 29 30   28 29 30
                                     31

October          November          December
Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su
    1  2  3  4              1              1  2  3  4  5  6
  5  6  7  8  9 10 11    2  3  4  5  6  7  8    7  8  9 10 11 12 13
12 13 14 15 16 17 18    9 10 11 12 13 14 15   14 15 16 17 18 19 20
19 20 21 22 23 24 25   16 17 18 19 20 21 22   21 22 23 24 25 26 27
26 27 28 29 30 31     23 24 25 26 27 28 29   28 29 30 31
                                     30

```

Activate Windows

math module:

write a python program which accepts the radius of a circle from user and computes the area.

```
import math
```

```
r=int(input("Enter radius:"))
```

```
area=math.pi*r*r
```

```
print("Area of circle is:",area)
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/pyyy/m.py =

Enter radius:4

Area of circle is: 50.26548245743669

```
>>> import math
```

```
>>> print("The value of pi is", math.pi)
```

O/P: The value of pi is 3.141592653589793

Import with renaming:

- We can import a module by renaming it as follows.
- **For Eg:**

```
>>> import math as m
```

```
>>> print("The value of pi is", m.pi)
```

O/P: The value of pi is 3.141592653589793

- We have renamed the math module as m. This can save us typing time in some cases.
- Note that the name math is not recognized in our scope. Hence, math.pi is invalid, m.pi is the correct implementation.

Python from...import statement:

- We can import specific names from a module without importing the module as a whole. Here is an example.

```
>>> from math import pi
```

```
>>> print("The value of pi is", pi)
```

O/P: The value of pi is 3.141592653589793

- We imported only the attribute pi from the module.
- In such case we don't use the dot operator. We could have imported multiple attributes as follows.


```
>>> from math import pi, e
```

```
>>> pi
```

```
3.141592653589793
```

```
>>> e
```

```
2.718281828459045
```

Import all names:

- We can import all names(definitions) from a module using the following construct.

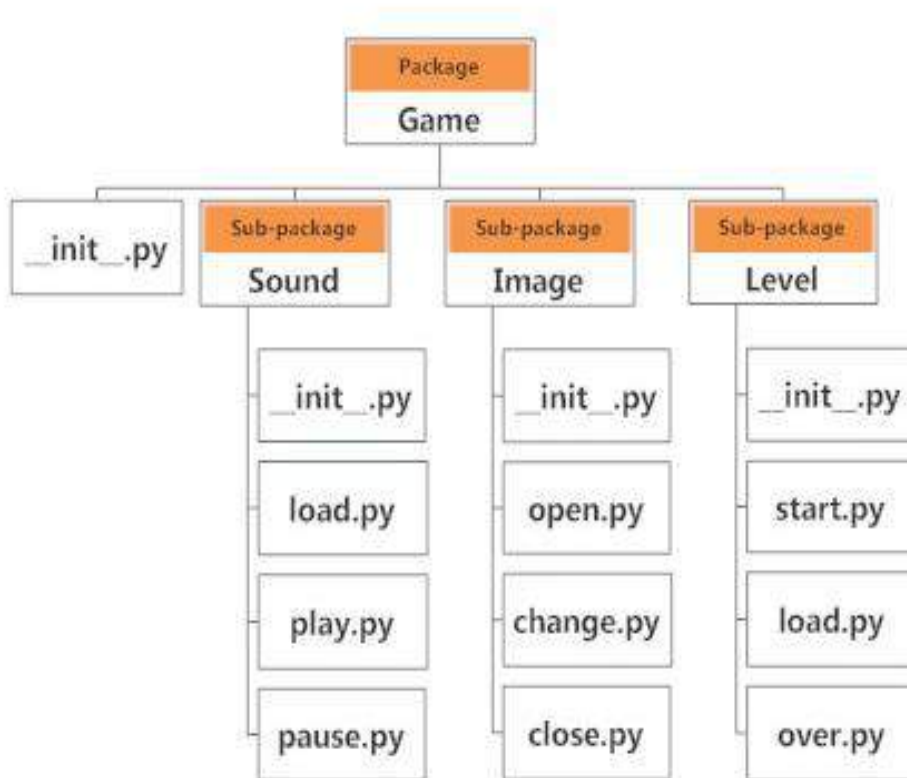
```
>>>from math import *
```

```
>>>print("The value of pi is", pi)
```

- We imported all the definitions from the math module. This makes all names except those beginning with an underscore, visible in our scope.

Explore packages:

- We don't usually store all of our files in our computer in the same location. We use a well-organized hierarchy of directories for easier access.
- Similar files are kept in the same directory, for example, we may keep all the songs in the "music" directory. Analogous to this, Python has packages for directories and modules for files.
- As our application program grows larger in size with a lot of modules, we place similar modules in one package and different modules in different packages. This makes a project (program) easy to manage and conceptually clear.
- Similar, as a directory can contain sub-directories and files, a Python package can have sub-packages and modules.
- A directory must contain a file named `__init__.py` in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.
- Here is an example. Suppose we are developing a game, one possible organization of packages and modules could be as shown in the figure below.



- If a file named `__init__.py` is present in a package directory, it is invoked when the package or a module in the package is imported. This can be used for execution of package initialization code, such as initialization of package-level data.
- For example `__init__.py`
- A **module** in the package can access the global by importing it in turn
- We can import modules from packages using the dot (.) operator.
- For example, if want to import the start module in the above example, it is done as follows.
- `import Game.Level.start`
- Now if this module contains a function named `select_difficulty()`, we must use the full name to reference it.
- `Game.Level.start.select_difficulty(2)`

- If this construct seems lengthy, we can import the module without the package prefix as follows.
- `from Game.Level import start`
- We can now call the function simply as follows.
- **`start.select_difficulty(2)`**
- Yet another way of importing just the required function (or class or variable) from a module within a package would be as follows.
- **`from Game.Level.start import select_difficulty`**
- Now we can directly call this function.
- **`select_difficulty(2)`**

Examples:

#Write a python program to create a package (II YEAR),sub-package(CSE),modules(student) and create read and write function to module

```
def read():
```

```
    print("Department")
```

```
def write():
```

```
    print("Student")
```

Output:

```
>>> from IIYEAR.CSE import student
```

```
>>> student.read()
```

```
Department
```

```
>>> student.write()
```

```
Student
```

```
>>> from IIYEAR.CSE.student import read
```

```
>>> read
```

```
<function read at 0x03BD1070>
```

```
>>> read()
```

Department

```
>>> from IIYEAR.CSE.student import write
```

```
>>> write()
```

Student

Write a program to create and import module?

```
def add(a=4,b=6):
```

```
    c=a+b
```

```
    return c
```

Output:

C:\Users\MRCET\AppData\Local\Programs\Python\Python38-32\IIYEAR\modu1.py

```
>>> from IIYEAR import modu1
```

```
>>> modu1.add()
```

10

Write a program to create and rename the existing module.

```
def a():
```

```
    print("hello world")
```

```
a()
```

Output:

C:/Users/MRCET/AppData/Local/Programs/Python/Python38-32/IIYEAR/exam.py

hello world

```
>>> import exam as ex
```

hello world



MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12 (B) of UGC ACT 1956

Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2008 Certified
Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, India



**Department of Computer Science and
Engineering**

WEB TECHNOLOGIES

LECTURE NOTES

B.TECH
(III YEAR – II SEM)

(2019-20)

Faculty in charge

HOD-CSE

Ch. Naveen Kumar Reddy, K. M. Rayudu & N. Vijay Kumar

Dr. D SUJATHA

UNIT - I

Web Basics and Overview: Introduction to Internet, World Wide Web, Web Browsers, URL, MIME, HTTP, Web Programmers Tool box.

HTML Common tags: List, Tables, images, forms, frames, Basics of CSS and types of CSS.

Client-Side Programming (Java Script): Introduction to Java Script, declaring variables, functions, Event handlers (onclick, onsubmit, etc.,) and Form Validation.

Introduction to Internet:- A global computer network providing a variety of information and communication facilities, consisting of interconnected networks using standardized communication protocols. "the guide is also available on the Internet"

It is a **network of networks** that consists of private, public, academic, business, and government networks of local to global scope, linked by a broad array of electronic, wireless, and optical networking technologies.

History of Internet

- The **First Practical schematics** for the internet arrived in the **early 1960s**, when **MIT** popularized the idea of an “**Galactic Network**” of computers. Shortly thereafter, computer scientists developed the concept of “**packet switching**,” a method for effectively transmitting electronic data that would later become one of the major building blocks of the internet.
- The **first workable prototype** of the Internet came in the **late 1960s** with the creation of **ARPANET**, or the **Advanced Research Projects Agency Network**. Originally funded by the U.S. **Department of Defense**, ARPANET used packet switching to allow **multiple computers to communicate on a single network**.
- On October 29, **1969**, ARPAnet delivered its first message: a “**node-to-node**” communication from one computer to another
- The technology continued to grow in the **1970s** after scientist **Vinton Cerf** developed Transmission Control Protocol and Internet Protocol, or **TCP/IP**, a communications model that set standards for how data could be transmitted between multiple networks.
- ARPANET adopted TCP/IP on **January 1, 1983**, and from there researchers began to assemble the “**network of networks**” that became the modern Internet
- The online world then took on a more recognizable form in **1990**, when computer scientist Tim Berners-Lee invented the **World Wide Web**

World Wide Web

- The World Wide Web, commonly known as the Web, is **a system of interlinked hypertext documents** accessed via the **Internet**
- The **World Wide Web** (abbreviated WWW or the Web) is an **information space /repository** where documents and other **web resources are identified by Uniform Resource Locators (URLs)**, interlinked by hypertext links, and can be accessed via the Internet.

The World Wide Web (abbreviated WWW or the Web) is an information space where documents and other web resources are identified by Uniform Resource Locators (URLs), interlinked by hypertext links, and can be accessed via the Internet. English scientist Tim Berners-Lee invented the World Wide Web in 1989. He wrote the first web browser computer program in 1990 while employed at CERN in Switzerland. The Web browser was released outside CERN in 1991, first to other research institutions starting in January 1991 and to the general public on the Internet in August 1991.

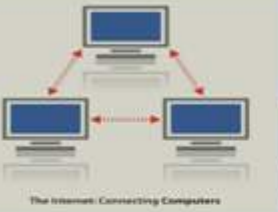

The World Wide Web has been central to the development of the Information Age and is the primary tool billions of people use to interact on the Internet. Web pages are primarily text documents formatted and annotated with Hypertext Markup Language (HTML). In addition to formatted text, web pages may contain images, video, audio, and software components that are rendered in the user's web browser as coherent pages of multimedia content.

Embedded hyperlinks permit users to navigate between web pages. Multiple web pages with a common theme, a common domain name, or both, make up a website. Website content can largely be provided by the publisher, or interactively where users contribute content or the content depends upon the users or their actions. Websites may be mostly informative, primarily for entertainment, or largely for commercial, governmental, or non-governmental organizational purposes



WWW is another example of client/server computing. Each time a link is followed, the client is requesting a document (or graphic or sound file) from a server (also called a Web server) that's part of the World Wide Web that "serves" up the document. The server uses a protocol called HTTP or Hyper Text Transport Protocol. The standard for creating hypertext documents for the WWW is Hyper Text Markup Language or HTML. HTML essentially codes plain text documents so they can be viewed on the Web.

Difference between internet and WWW

Internet	WWW
 <p>The Internet: Connecting Computers</p> <p>It is a network of network</p> <p>Network connecting computers</p>	 <p>The Web: Connecting People</p> <p>It is a repository of Common resources</p> <p>Connects users/resources over internet</p>
Nature of Internet is hardware .	Nature of www is software .
Internet consists of computers, routers, cables, bridges, servers, cellular towers, satellites etc	www consists of information like text, images, audio, video
Internet works on the basis of Internet Protocol (IP)	WWW works on the basis of Hyper Text Transfer Protocol (HTTP)
Computing devices are identified by IP Addresses	Information pieces are identified by Uniform Resource Locator (URL)

Browsers:

- A browser is a **program** that allows us to view and explore information on the web . A software application for **retrieving, presenting, and traversing** information resources on the World Wide Web.
- Web browser can **show text, audio, video, animation and more**. It is the responsibility of a web browser to **interpret text and commands contained in the web page**.

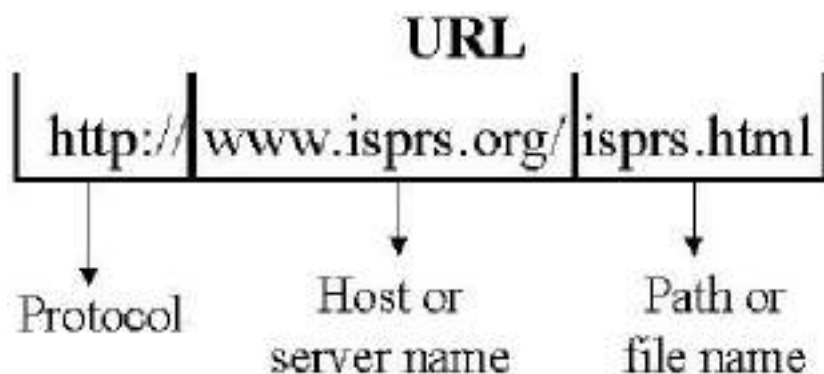
WWW Clients, or "Browser": The program you use to access the WWW is known as a browser because it "browses" the WWW and requests these hypertext documents. Browsers can be graphical, allows to see and hear the graphics and audio;

text-only browsers (i.e., those with no sound or graphics capability) are also available. All of these programs understand http and other Internet protocols such as FTP, gopher, mail, and news, making the WWW a kind of "one stop shopping" for Internet users.

YEAR	Browser
1990	Nexus -The first browser ever created by W3C director Tim Berners-lee
1992	"Lynx" was a texted-based browser that couldn't display any graphic content.
1993	Mosaic
1994	Netscape Navigator.
1995	Internet Explorer
1996	Opera
2003	Apple's Safari
2004	Firefox
2007	Mobile Safari
2008	Google Chrome
2011	Opera Mini
2015	Microsoft Edge

Uniform Resource Locators, or URLs:

- A **Uniform Resource Locator**, or **URL** is the **address of a document/ web page** found on the WWW.
- Each webpage has a designated unique address called the URL or the Uniform Resource Locator
- Browser **interprets the information in the URL** in order to connect to the proper Internet server and to retrieve the desired document.
- Each time a click on a hyperlink in a WWW document instructs browser to find the URL that's embedded within the hyperlink.



❖ **Protocol:** A protocol that is used to transfer pages on the web.

- Hypertext protocol: <http://www.aucegypt.edu>
- File Transfer Protocol: <ftp://ftp.dartmouth.edu>

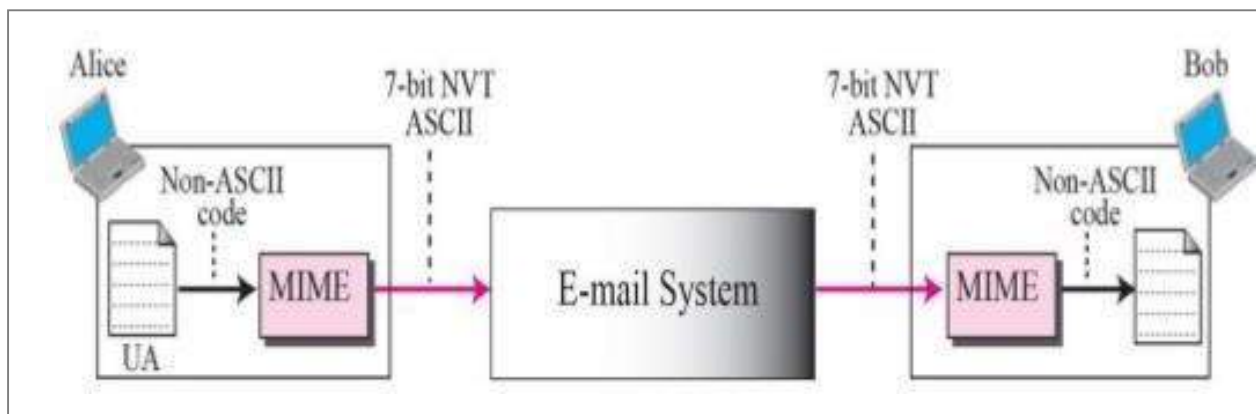
Domain It is a unique reference that identifies a website

It divides World Wide Web sites into **categories based on the nature** of their owner, and they form part of a site's address, or uniform resource locator (URL).

.com—commercial enterprises	.mil—military site
org—organization site (non-profits, etc.)	int—organizations established by international treaty
.net—network	.biz—commercial and personal
.edu—educational site (universities, schools, etc.)	.info—commercial and personal
.gov—government organizations	.name—personal sites

Additional three- letter, four- letter, and longer top- level domains are frequently added. Each country linked to the Web has a two- letter top- level domain, for example .fr is France, .ie is Ireland.

- **MIME (Multi-Purpose Internet Mail Extensions):-** MIME is an extension of the original **Internet e- mail** protocol that lets people use the protocol to exchange different kinds of data files on the Internet: audio, video, images, application programs, and other kind, as well as the ASCII text handled in the original protocol, Simple Mail Transport Protocol (SMTP)
- The features offered by MIME to email services are as follows:
 - Support for **multiple attachments** in a single message
 - Support for **non-ASCII characters**
 - Support for **layouts, fonts and colors** which are categorized as rich text.
 - Support for attachments which may contain executables, audio, images and video files, etc.
 - Support for unlimited message length.



Hypertext Transport Protocol:

- HTTP stands for **Hyper Text Transfer Protocol**.
- It is a protocol **used to access the data** on the **World Wide Web (www)**.
- This protocol defines **how messages** are **formatted and transmitted**
- For example, when we enter a URL in web browser, this actually **sends an HTTP command** to the **Web server** directing it to **fetch and transmit the requested Web page**
- HTTP is similar to the FTP as it also transfers the files from one host to another host. But, HTTP is simpler than FTP as HTTP uses only one connection, i.e., no control connection to transfer



files.

The Web Programmer's Toolbox:

- **HTML** - a *markup* language
 - To describe the general form and layout of documents
 - HTML is **not** a programming language - it cannot be used to describe **computations**.
 - An HTML document is a mix of **content** and **controls**
 - Controls are **tags** and their **attributes**
 - Tags often delimit content and specify something about how the content should be arranged in the document
For example, `<p>Write a paragraph here </p>` is an *element*.
 - Attributes provide additional information about the content of a tag
For example, ` `
- Plug ins
 - Integrated into tools like word processors, effectively converting them to WYSIWYG HTML editors
- Filters
 - Convert documents in other formats to HTML
 - - XML
 - A meta-markup language (a language for defining markup language)
 - Used to create a new markup language for a particular purpose or area
 - Because the tags are designed for a specific area, they can be meaningful
- JavaScript
 - A client-side HTML-embedded scripting language
 - Provides a way to access elements of HTML documents and dynamically change them
- Flash
 - A system for building and displaying text, graphics, sound, interactivity, and animation (movies)
 - Two parts:
 1. Authoring environment
 2. Player

Supports both motion and shape animation

PHP A server-side scripting language

Great for form processing and database access through the Web

Ajax

Asynchronous JavaScript + XML

- No new technologies or languages

Much faster for Web applications that have extensive user/server interactions

Uses asynchronous requests to the server

Requests and receives small parts of documents, resulting in much faster responses

Java Web Software

Servlets – server-side Java classes

JavaServer Pages (JSP) – a Java-based approach to server-side scripting

JavaServer Faces – adds an event-driven interface model on JSP

ASP.NET

Does what JSP and JSF do, but in the .NET environment

HTML Common tags:-

HTML is the building block for web pages. HTML is a format that tells a computer how to display a web page. The documents themselves are plain text files with special "tags" or codes that a web browser uses to interpret and display information on your computer screen.

- HTML stands for Hyper Text Markup Language
- An HTML file is a text file containing small markup tags
- The markup tags tell the Web browser how to display the page
- An HTML file must have an htm or html file extension.

HTML Tags:- HTML tags are used to mark-up HTML elements .HTML tags are surrounded by the two characters < and >. The surrounding characters are called angle brackets. HTML tags normally come in pairs like **and** The first tag in a pair is the start tag, the second tag is the end tag . The text between the start and end tags is the element content . HTML tags are not case sensitive, **means the same as**.

The most important tags in HTML are tags that define headings, paragraphs and line breaks.

Tag	Description
<!DOCTYPE...>	This tag defines the document type and HTML version.
<html>	This tag encloses the complete HTML document and mainly comprises of document header which is represented by <head>...</head> and document body which is represented by <body>...</body> tags.
<head>	This tag represents the document's header which can keep other HTML tags like <title>, <link> etc.
<title>	The <title> tag is used inside the <head> tag to mention the document title.
<body>	This tag represents the document's body which keeps other HTML tags like <h1>, <div>, <p> etc.
<p>	This tag represents a paragraph.
<h1> to <h6>	Defines header 1 to header 6
 	Inserts a single line break
<hr>	Defines a horizontal rule
<!-->	Defines a comment

Headings:-

Headings are defined with the <h1> to <h6> tags. <h1> defines the largest heading while <h6> defines the smallest.

<h1>This is a heading</h1>

<h2>This is a heading</h2>

<h3>This is a heading</h3>

<h4>This is a heading</h4>

<h5>This is a heading</h5>

<h6>This is a heading</h6>

Paragraphs:-

Paragraphs are defined with the <p> tag. Think of a paragraph as a block of text. You can use the align attribute with a paragraph tag as well.

```
<p align="left">This is a paragraph</p>
<p align="center">this is another paragraph</p>
```

Note: You must indicate paragraphs with <p> elements. A browser ignores any indentations or blank lines in the source text. Without <p> elements, the document becomes one large paragraph. HTML automatically adds an extra blank line before and after a paragraph.


Line Breaks:-

The
 tag is used when you want to start a new line, but don't want to start a new paragraph. The
 tag forces a line break wherever you place it. It is similar to single spacing in a document.

This Code	output
<code><p>This
 is a para
 graph with line breaks</p></code>	This is a para graph with line breaks

Horizontal Rule The element is used for horizontal rules that act as dividers between sections like this:

The horizontal rule does not have a closing tag. It takes attributes such as align and width

Code	Output
<code><hr width="50%" align="center"></code>	

Sample html program

```
<!DOCTYPE html>
<html>
  <head>
    <title>This is document title
    </title>
  </head>
  <body>
    <h1>This is a heading</h1>
    <p>Document content goes here ....</p>
  </body>
</html>
```



- Type the above program in notepad and save with some file name eg:sample.html
- Open the file with browser and the webpage looks like this

HTML Formatting Elements

Element name	Description
	Bold Text
	This is a logical tag, which tells the browser that the text is important.
<i>	used to make text italic.
	used to display content in italic.
<mark>	This tag is used to highlight text.
<u>	used to underline text written between it.
<strike>	This tag is used to draw a strikethrough on a section of text
<sup>	-Superscript text
<sub>	Subscript text
	This tag is used to display the deleted content.
<ins>	This tag displays the content which is added
<big>	This tag is used to increase the font size by one conventional unit.
<small>	This tag is used to decrease the font size by one unit from base font size.

Lists:-

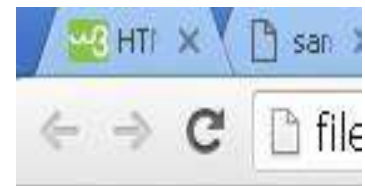
HTML offers web authors three ways for specifying lists of information. All lists must contain one or more list elements. Lists are of three types

- 1) Un ordered list
- 2) Ordered List
- 3) Definition list

1). HTML Unordered Lists: An unordered list is **a collection of related items that have no special order or sequence**. This list is created by using HTML `` tag. Each item in the list is marked with a bullet.

Example

```
<!DOCTYPE html>
<html>
  <head>
    <title>HTML Unordered List</title>
  </head>
  <body>
    <ul>
      <li>Beetroot</li>
      <li>Ginger</li> <li>Potato</li>
      <li>Radish</li>
    </ul>
  </body>
</html>
```



- Beetroot
- Ginger
- Potato
- Radish

The type Attribute

we can use type attribute for `` tag to specify the type of bullet we like. By default, it is a disc.

Following are the possible options –

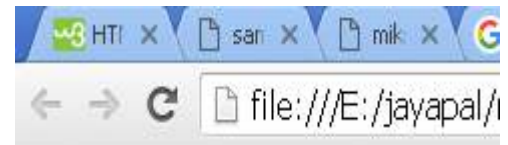
`<ul type = "square">` `<ul type = "disc">` `<ul type = "circle">`

Example

```
<!DOCTYPE html>
<html>
<head> <title>HTML Unordered List</title>
</head>
<body>
  <ul type = "square"> <li>Beetroot</li>
  <li>Ginger</li>
  <li>Potato</li>
  <li>Radish</li>
</ul>
</body>
</html>
```

- Beetroot
- Ginger
- Potato
- Radish


```
<!DOCTYPE html>
<html>
  <head>
    <title>HTML Ordered List</title>
  </head>
  <body>
    <ol>
      <li>Beetroot</li>
      <li>Ginger</li>
      <li>Potato</li>
      <li>Radish</li>
    </ol>
  </body>
</html>
```



1. Beetroot
2. Ginger
3. Potato
4. Radish

2). HTML Ordered Lists:- items are numbered list instead of bulleted, This list is created by using **** tag

The type Attribute:

we can use type attribute for **** tag to specify the type of numbering we like. By default, it is a number. Following are the possible options.

<ol type = "1"> - Default-Case Numerals.

<ol type = "I"> - Upper-Case Numerals.

<ol type = "i"> - Lower-Case Numerals.

<ol type = "A"> - Upper-Case Letters.

<ol type = "a"> - Lower-Case Letters.

The start Attribute

we can use **start** attribute for **** tag to specify the starting point of numbering we need. Following are the possible options –

<ol type = "1" start = "4"> - Numerals starts with 4.

<ol type = "I" start = "4"> - Numerals starts with IV.

<ol type = "i" start = "4"> - Numerals starts with iv.

<ol type = "a" start = "4"> - Letters starts with d.

<ol type = "A" start = "4"> - Letters starts with D.

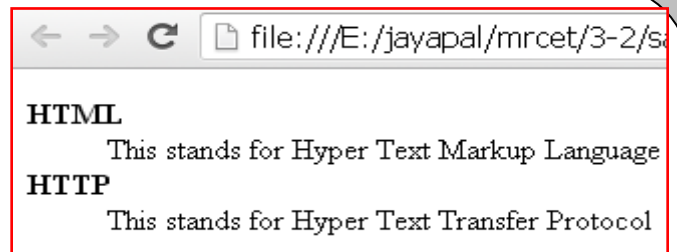
```
<html>
<head> <title>HTML Ordered List</title> </head>
<body>
<ol type = "i" start = "4">
<li>Beetroot</li>
<li>Ginger</li>
<li>Potato</li>
<li>Radish</li>
</ol>
</body>
</html>
```

```
iv. Beetroot
v. Ginger
vi. Potato
vii. Radish
```

3). HTML Definition Lists:- HTML and XHTML supports a list style which is called definition lists where entries are listed like in a dictionary or encyclopedia. The definition list is the ideal way to present a glossary, list of terms, or other name/value list. Definition List makes use of following three tags.

- 1). <dl> - Defines the start of the list
- 2). <dt> - A term
- 3). <dd> - Term definition
- 4). </dl> - Defines the end of the list

```
<!DOCTYPE html>
<html>
<head>
<title>HTML Definition List</title>
</head>
<body>
<dl>
<dt><b>HTML</b></dt> <dd>This stands for Hyper Text Markup Language</dd>
<dt><b>HTTP</b></dt> <dd>This stands for Hyper Text Transfer Protocol</dd>
</dl>
</body>
</html>
```



HTML tables:

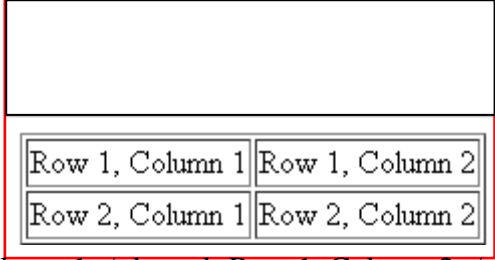
HTML table tag is used to display data in tabular form (row * column)

A table is created using the following tags.

We can create a table to display data in tabular form, using **<table>** element, with the help of **<tr>** , **<td>**, and **<th>** elements.

Example:

```
<html>
<head>
<title>HTML Tables</title>
</head>
<body>
  <table border="1">
    <tr>
      <td>Row 1, Column 1</td> <td>Row 1, Column 2</td>
    </tr>
    <tr>
      <td>Row 2, Column 1</td> <td>Row 2, Column 2</td>
    </tr>
  </table>
</body>
</html>
```



Example:

```
<html>
<head> <title>HTML Tables</title>
</head>
<body>
<table border="1">
  <tr>
    <td>Jill</td>
    <td>Smith</td>
    <td>50</td>
  </tr>
  <tr>
    <td>Eve</td>
    <td>Jackson</td>
    <td>94</td>
  </tr></table>
</body>
</html>
```

Jill	Smith	50
Eve	Jackson	94

- **Table Heading:** Table heading can be defined using `<th>` tag. This tag will be put to replace `<td>` tag, which is used to represent actual data cell. Normally you will put your top row as table heading as shown below, otherwise you can use `<th>` element in any row.

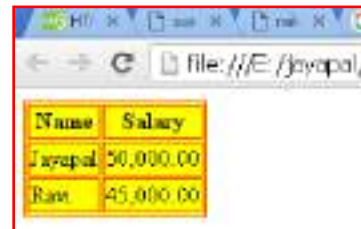
Tables Backgrounds: To set table background using one of the following two ways:

1) **bgcolor attribute** - You can set background color for whole table or just for one cell.

2) **background attribute** - You can set background image for whole table or just for one cell.

You can also set border color also using **bordercolor** attribute.

```
<html>
<head>
<title>HTML Tables</title> </head>
<body>
  <table border="1"bordercolor="red" bgcolor="yellow">
    <tr> <th>Name</th>
    <th>Salary</th> </tr>
    <td>Jayapal   </td> <td>50,000.00</td>
  </tr>
    <tr> <td>Ravi</td> <td>45,000.00</td>
  </tr>
  </table>
</body>
</html>
```



Name	Salary
Jayapal	50,000.00
Ravi	45,000.00

Images:

Images are very important to beautify as well as to depict many complex concepts in simple way on your web page.

Insert Image:

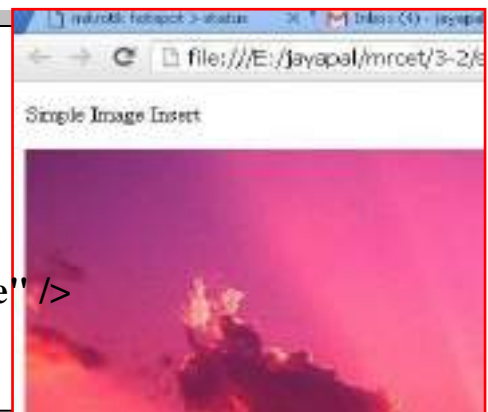
insert any image in the web page by using **** tag.

Attribute Values

Value	Description
left	Align the image to the left
right	Align the image to the right
middle	Align the image in the middle
top	Align the image at the top
bottom	Align the image at the bottom

**
<html>
 <head>
 <title>Using Image in Webpage</title>
 </head>
 <body> <p>Simple Image Insert</p>

 </body>
</html>
```



**FORMS:**

HTML Forms are required to collect some data from the site visitor. For example, during user registration you would like to collect information such as name, email address, credit card, etc. A form will take input from the site visitor and then will post it to a back-end application such as CGI, ASP Script or PHP script etc. The back-end application will perform required processing on the passed data based on defined business logic inside the application. There are various form elements available like text fields, text area fields, drop-down menus, radio buttons, checkboxes, etc.

```
<form action="Script URL" method="GET|POST"> form elements like input, text area etc. </form>
```

**Form Attributes**

Apart from common attributes, following is a list of the most frequently used form attributes:

Attribute	Description
action	Backend script ready to process your passed data.
method	Method to be used to upload data. The most frequently used are GET and POST methods.
target	Specify the target window or frame where the result of the script will be displayed. It takes values like _blank, _self, _parent etc.
enctype	<p>You can use the enctype attribute to specify how the browser encodes the data before it sends it to the server. Possible values are:</p> <p>application/x-www-form-urlencoded - This is the standard method most forms use in simple scenarios.</p> <p>mutlipart/form-data - This is used when you want to upload binary data in the form of files like image, word file etc.</p>

## HTML Form Controls :

There are different types of form controls that you can use to collect data using HTML form.

``

- Text Input Controls
- Checkboxes Controls
- Radio Box Controls
- Select Box Controls
- File Select boxes
- Hidden Controls
- Clickable Buttons
- Submit and Reset Button

### Text Input Controls:-

There are three types of text input used on forms:

- 1) **Single-line text input controls** - This control is used for items that require only one line of user input, such as search boxes or names. They are created using HTML `<input>` tag.

**`<input type="text">` defines a one-line input field for **text input**:**

#### Example:

```
<form>
First name:

<input type="text" name="firstname">

Last name:

<input type="text" name="lastname">
</form>
```



- 2) **Password input controls** - This is also a single-line text input but it masks the character as soon as a user enters it. They are also created using HTML `<input>` tag.

#### Input Type Password

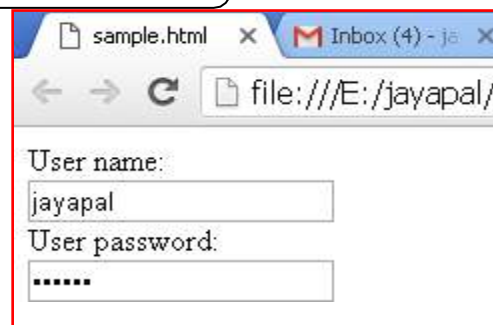
**`<input type="password">` defines a **password field**:**

```
<form>
User name:

<input type="text" name="username">

User password:

<input type="password" name="psw">
</form>
```

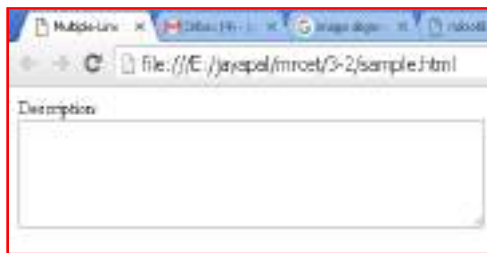




**3) Multi-line text input controls** - This is used when the user is required to give details that may be longer than a single sentence. Multi-line input controls are created using HTML `<textarea>` tag.

```
<!DOCTYPE html>
<html>
 <head>
 <title>Multiple-Line Input Control</title>
 </head>
 <body>
 <form> Description:

 <textarea rows="5" cols="50" name="description"> Enter description here... </textarea>
 </form>
 </body>
</html>
```



### Checkboxes Controls:-

Checkboxes are used when more than one option is required to be selected. They are also created using HTML `<input>` tag but type attribute is set to checkbox.

Here is an example HTML code for a form with two checkboxes:

```
<!DOCTYPE html>
<html> <head> <title>Checkbox Control</title> </head>
<body>
 <form>
 <input type="checkbox" name="C++" value="on"> C++

 <input type="checkbox" name="C#" value="on"> C#

 <input type="checkbox" name="JAVA" value="on"> JAVA
 </form>
</body> </html>
```



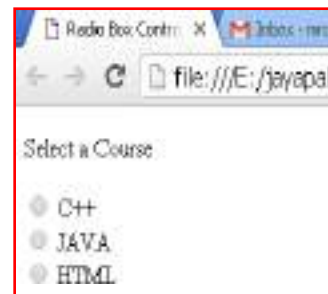
### Radio Button Control:-

Radio buttons are used when out of many options, just one option is required to be selected. They are also created using HTML `<input>` tag but type attribute is set to radio.

```
<!DOCTYPE html>
<html> <head> <title>Radio Box Control</title> </head>
 <body> <p>Select a Course</p>
 <form>
 <input type="radio" name="subject" value="C++"> C++

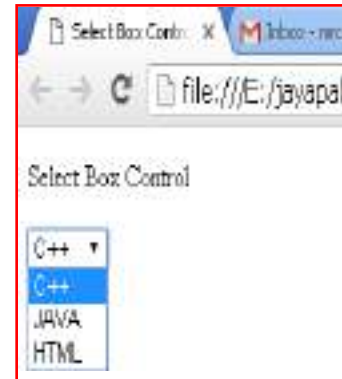
 <input type="radio" name="subject" value="JAVA"> JA VA

 <input type="radio" name="subject" value="HTML"> HTML
 </form>
 </body> </html>
```



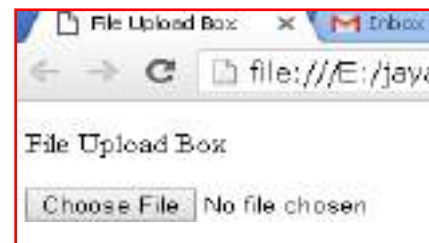
**Select Box Controls :-** A select box, also called drop down box which provides option to list down various options in the form of drop down list, from where a user can select one or more options.

```
<!DOCTYPE html>
<html>
<head>
 <title>Select Box Control</title>
</head>
<body>
 <form>
 <select name="dropdown">
 <option value="C++" selected>C++</option>
 <option value="JAVA">JA VA</option>
 <option value="HTML">HTML</option>
 </select>
 </form>
</body>
</html>
```



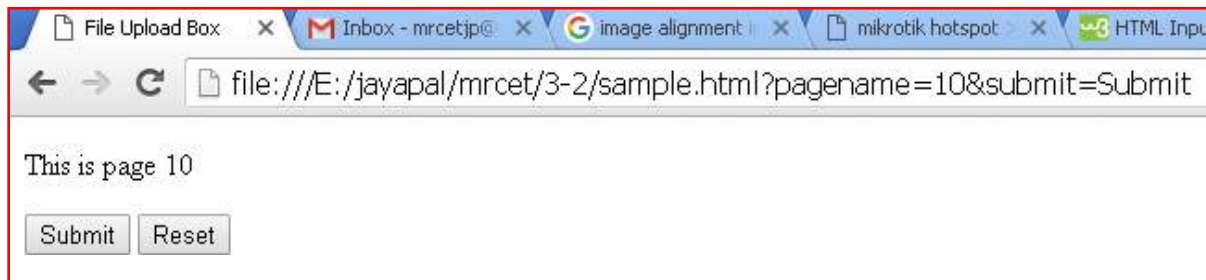
**File Select boxes:-** If you want to allow a user to upload a file to your web site, you will need to use a file upload box, also known as a file select box. This is also created using the `<input>` element but type attribute is set to **file**.

```
<!DOCTYPE html>
<html>
 <head>
 <title>File Upload Box</title>
 </head>
 <body>
 <p>File Upload Box</p>
 <form>
 <input type="file" name="fileupload" accept="image/*" />
 </form>
 </body>
</html>
```



**Hidden Controls:-** Hidden form controls are used to hide data inside the page which later on can be pushed to the server. This control hides inside the code and does not appear on the actual page. For example, following hidden form is being used to keep current page number. When a user will click next page then the value of hidden control will be sent to the web server and there it will decide which page will be displayed next based on the passed current page.

```
<html> <head> <title>File Upload Box</title> </head>
 <body>
 <form>
 <p>This is page 10</p>
 <input type="hidden" name="pagename" value="10" />
 <input type="submit" name="submit" value="Submit" />
 <input type="reset" name="reset" value="Reset" />
 </form>
 </body>
</html>
```



### Button Controls:-

There are various ways in HTML to create clickable buttons. You can also create a clickable button using `<input>` tag by setting its type attribute to **button**. The type attribute can take the following values:

Type	Description
submit	This creates a button that automatically submits a form.
reset	This creates a button that automatically resets form controls to their initial values.
button	This creates a button that is used to trigger a client-side script when the user clicks that button.
image	This creates a clickable button but we can use an image as background of the button.

```

<!DOCTYPE html>
<html>
<head>
 <title>File Upload Box</title>
</head>
<body>
 <form>
 <input type="submit" name="submit" value="Submit" />
 <input type="reset" name="reset" value="Reset" />
 <input type="button" name="ok" value="OK" />
 <input type="image" name="imagebutton" src="test1.png" />
 </form>
</body> </html>

```



**HTML frames:** These are used to divide your browser window into multiple sections where each section can load a separate HTML document. A collection of frames in the browser window is known as a frameset. The window is divided into frames in a similar way the tables are organized: into rows and columns.

To use frames on a page we use <frameset> tag instead of <body> tag. The <frameset> tag defines, how to divide the window into frames. The **rows** attribute of <frameset> tag defines horizontal frames and **cols** attribute defines vertical frames. Each frame is indicated by <frame> tag and it defines which HTML document shall open into the frame.

**Note:** HTML <frame> Tag. **Not Supported in HTML5.**

```
<frameset cols="25%,50%,25%">
 <frame src="frame_a.htm">
 <frame src="frame_b.htm">
 <frame src="frame_c.htm">
</frameset>
```

```
<!DOCTYPE html>
<html>
 <head>
 <title>Page Title</title>
 </head>
 <body>
 <iframe src="sample1.html" height="400" width="400" frameborder="1">
 <h1>This is a Heading</h1>
 <p>This is a paragraph.</p>
 </iframe>
 </body>
</html>
```



## HTML frames

- Frames divide **web-browser** window into **multiple sections** where **each section** can load a **separate HTML document**.
- A **collection of frames** in the browser window is known as a **frameset**.
- **Frames** can be created using **<frameset>** tag and **<frame>** tag

**Creating Frame:**

Vertical division

Horizontal division

frame1

frame2

40 %

60 %

**Syntax :**

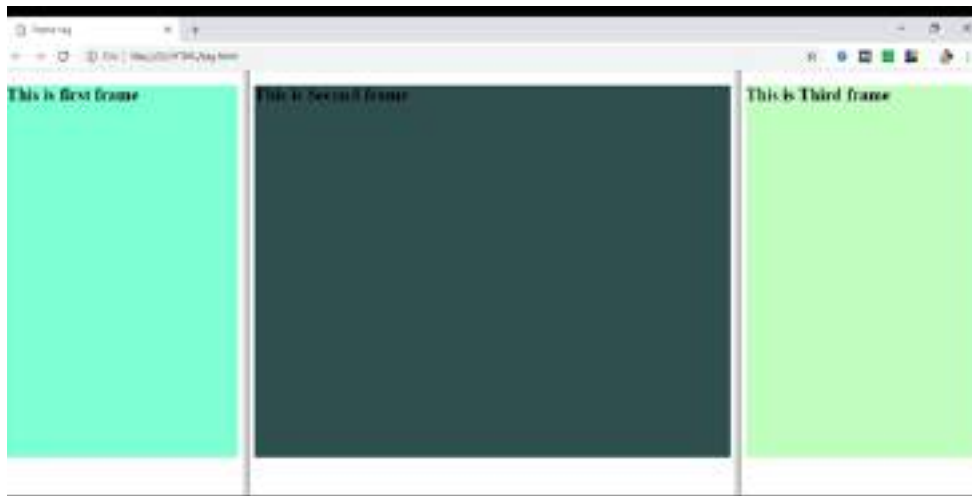
```
<frameset cols/rows="40%,60%">
 <frame name="left" src="frame_left.html">
 <frame name="right" src="frame_righ.html">
</frameset>
```

- The **<frameset>** tag defines, how to divide the window into frames.
- The **rows** attribute of **<frameset>** tag defines horizontal frames and **cols** attribute defines vertical frames.
- Each frame is indicated by **<frame>** tag and it defines which HTML document shall open into the frame.

**Create Vertical frames:** Following is the **example** to create three vertical frames:

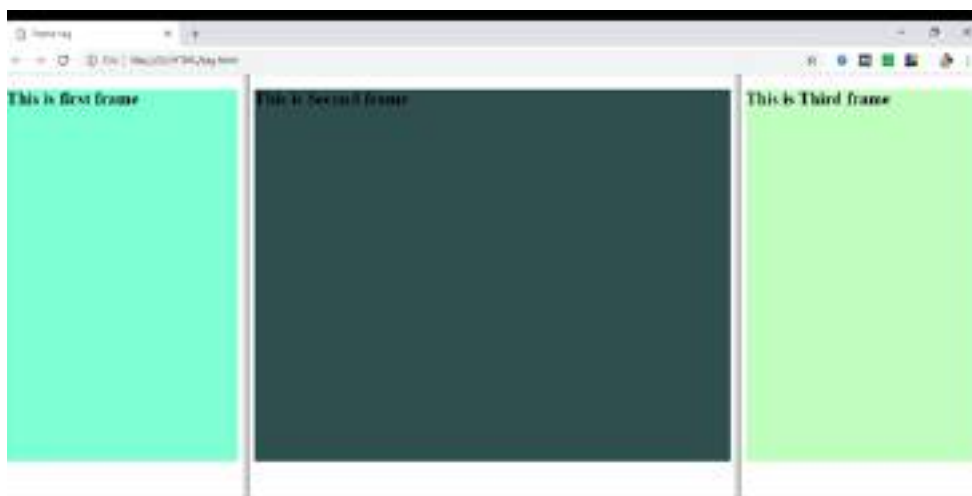
```
<html>
<head>
 <title>Frame tag</title>
</head>
<frameset cols="30%, 40%, 30%">
 <frame name="top" src="frame1.html" >
 <frame name="main" src="frame2.html">
 <frame name="bottom" src="frame3.html">

</frameset>
</html>
```



**Create Horizontal frames: Following is the example to create three horizontal frames:**

```
<html>
<head>
 <title>Frame tag</title>
</head>
<frameset cols="25%,50%,25%">
 <frame src="frame1.html" >
 <frame src="frame2.html">
 <frame src="frame3.html">
</frameset>
</html>
```



## HTML frames - &lt;frameset&gt; tag attributes

Attribute	Description
<b>cols</b>	<p>Specifies <b>how many columns</b> are contained in the frameset and the <b>size of each column</b>. You can specify the <b>width</b> of each column in one of the ways –</p> <ol style="list-style-type: none"> <li>1. Absolute values in <b>pixels</b>. For example, to create three vertical frames, use <i>cols = "100, 500, 100"</i>.</li> <li>2. A <b>percentage of the browser window</b>. For example, to create three vertical frames, use <i>cols = "10%, 80%, 10%"</i>.</li> <li>3. <b>Using a wildcard symbol</b>. For example, to create three vertical frames, use <i>cols = "10%, *, 10%"</i>. In this case wildcard * takes remainder of the window.</li> </ol>
<b>rows</b>	<p>This attribute works just like the cols attribute and takes the same values, but it is used to specify the <b>rows</b> in the frameset. For example, to create two horizontal frames, use <i>rows = "10%, 90%"</i> <b>We</b> can specify the height of each row in the same way as explained above for columns.</p>
<b>border</b>	<p>This attribute specifies the <b>width of the border of each frame in pixels</b>. For example, border = "5". A value of zero means no border.</p>

## HTML frames - &lt;frame&gt; tag attributes

Attribute	Description
<b>src</b>	<p>This attribute is used to give the file name that should be loaded in the frame.</p>
<b>name</b>	<p>Its value can be any URL. For example, src = "/html/top_frame.htm" This attribute allows you to give a name to a frame. It is used to indicate which frame a document should be loaded into.</p>
<b>marginwidth</b>	<p>This attribute allows you to specify the width of the space between the left and right of the frame's borders and the frame's content. The value is given in pixels. For example marginwidth = "10".</p>

margin height	This attribute allows you to specify the height of the space between the top and bottom of the frame's borders and its contents. The value is given in pixels. For example <code>marginheight = "10"</code> .
noresize	By default, you can resize any frame by clicking and dragging on the borders of a frame. The <code>noresize</code> attribute prevents a user from being able to resize the frame. For example <code>noresize = "noresize"</code> .
scrolling	This attribute controls the appearance of the scrollbars that appear on the frame. This takes values either "yes", "no" or "auto". For example <code>scrolling = "no"</code> means it should not have scroll bars.

### HTML frames - Nested frames

- It is possible to nest a frameset within another frameset

```
<html>
<frameset cols="40%,*,60%">
 <frame name="left" src="left.html">
 <frameset rows="50%,50%">
 <frame name="t_right" src="top-right.htm">
 <frame name="b_right" src="bottom-right.htm">
 </frameset>
</frameset>
</html>
```





## CSS (Cascading Style Sheets)

CSS describes **how HTML elements are to be displayed on screen, paper, or in other media.**

CSS **saves a lot of work.** It can control the layout of multiple web pages all at once.

CSS can be added to HTML elements in 3 ways:

- **Inline** - by using the style attribute in HTML elements
- **Internal** - by using a <style> element in the <head> section
- **External** - by using an external CSS file

### Inline CSS

An inline CSS is used to apply a unique style to a single HTML element.

An inline CSS uses the style attribute of an HTML element.

This example sets the text color of the <h1> element to blue:

```
<h1 style="color:blue;">This is a Blue Heading</h1>
```

```
<html> <head> <title>Page Title</title> </head>
 <body>
 <h1 style="color:blue;">This is a Blue Heading</h1>
 </body>
</html>
```



**Internal CSS:** An internal CSS is used to define a style for a single HTML page. An internal CSS is defined in the <head> section of an HTML page, within a <style> element:

```
<html>
 <head>
 <style>
 body { background-color: powderblue;}
 h1 { color: blue;}
 p { color: red;}
 </style>
 </head>
 <body>
 <h1>This is a heading</h1>
 <p>This is a paragraph.</p>
 </body>
</html>
```

### External CSS:-

An external style sheet is used to define the style for many HTML pages. **With an external style sheet, you can change the look of an entire web site, by changing one file!** To use an external style sheet, add a link to it in the <head> section of the HTML page:

```
<html>
 <head>
 <link rel="stylesheet" href="styles.css">
 </head>
 <body>
 <h1>This is a heading</h1>
 <p>This is a paragraph.</p>
 </body>
</html>
```

An external style sheet can be written in any text editor. The file must not contain any HTML code, and must be saved with a **.css extension**.

Here is how the "styles.css" looks:

```
body { background-color: powderblue; }
h1 { color: blue; }
p { color: red; }
```



**CSS Fonts:** The CSS **color** property defines the text color to be used.  
 The CSS **font-family** property defines the font to be used.  
 The CSS **font-size** property defines the text size to be used.

```
<html>
<head>
<style>
h1 {
 color: blue;
 font-family: verdana;
 font-size: 300%;
}
p {
 color: red;
 font-family: courier;
 font-size: 160%;
}
</style>
</head>
<body>
<h1>This is a heading</h1>
<p>This is a paragraph.</p>
</body>
</html>
```



**CSS Border:** The CSS **border** property defines a border around an HTML element.

**CSS Padding:** The CSS **padding** property defines a padding (space) between the text and the border.

**CSS Margin:** The CSS **margin** property defines a margin (space) outside the border.

```
<html> <head>
<style>
h1 {
 color: blue;
 font-family: verdana;
 font-size: 300%; }
p {
 color: red; font-size: 160%; border: 2px solid powderblue; padding: 30px; margin: 50px; }
</style>
</head>
<body>
<h1>This is a heading</h1>
<p>This is a paragraph.</p>
</body>
</html>
```



## JavaScript:

### What is JavaScript?

JavaScript is one popular scripting language over internet. Scripting means a small sneak (piece). It is always independent on other languages.

JavaScript is most commonly used as a client side scripting language. This means that JavaScript code is written into an HTML page. When a user requests an HTML page with JavaScript in it, the script is sent to the browser and it's up to the browser to do something with it.

### Difference between JavaScript and Java

JavaScript	Java
Cannot live outside a Web page	Can build stand-alone applications or live in a Web page as an <i>applet</i> .
Doesn't need a compiler	Requires a compiler
Knows all about your page	Applets are dimly aware of your Web page.
Untyped	Strongly typed
Somewhat object-oriented	Object-oriented

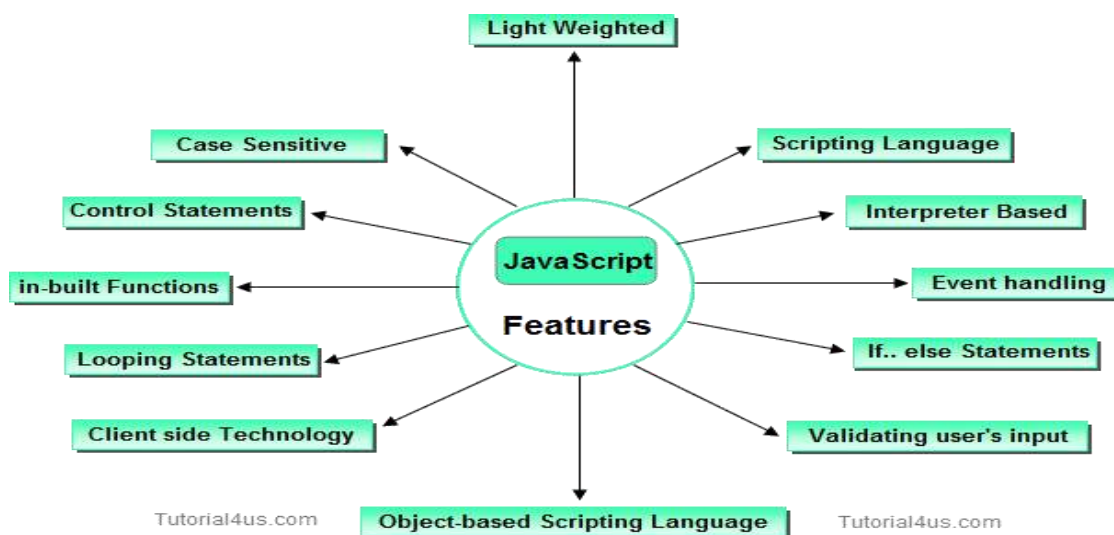
There are no relationship between in java & java script. Java Script is a scripting language that always dependent in HTML language. It used to css commands. It is mainly used to creating DHTML pages & validating the data. This is called client side validations.

### Why we Use JavaScript?

Using HTML we can only design a web page but you can not run any logic on web browser like addition of two numbers, check any condition, looping statements (for, while), decision making statement (if-else) at client side. All these are not possible using HTML So for perform all these task at client side you need to use JavaScript.

### Features of JavaScript

JavaScript is a client side technology, it is mainly used for gives client side validation, but it have lot of features which are given below;



→ **Java script is object based oriented language.**

Inheritance is does not support in JavaScript, so it is called object based oriented language.

→ JavaScript was developed by Netscape (company name) & initially called **live script**.

Later Microsoft developed & adds some features live script then it is called “**Jscript**”.

Jscript is nothing but **Java script**. We cannot create own classes in java script.

→ Java script is designed to **add interactivity to HTML pages**. It is usually embedded directly into html pages.

→ Java script is mainly useful to improve designs of WebPages, **validate form** data at client side, detects (find) visitor’s browsers, create and use to cookies, and much more.

→ Java script is also called **light weight programming language** , because Java script is return with very simple syntax. Java script is containing executable code.

→ Java script is also called **interpreted language** , because script code can be executed without preliminary compilation.

→ It Handling **dates, time, onSubmit, onLoad, onClick, onMouseOver & etc** .

→ JavaScript is **case sensitive**.

→ Most of the javascript control statements syntax is same as syntax of controlstatements in C language.

→ An important part of JavaScript is the ability to create new functions within scripts.

Declare a function in JavaScript using **function** keyword.

**Creating a java script:** - html script tag is used to script code inside the html page.

```
<script> </script>
```

The script is containing **2 attributes** . They are

**1) Language attribute:-**

It represents name of scripting language such as JavaScript, VbScript.

```
<script language=JavaScript>
```

**2) Type attribute:** - It indicates MIME (multi purpose internet mail extension) type of scripting code. It sets to an alpha-numeric MIME type of code.

```
<script type=JavaScript>
```

**Location of script or placing the script:** - Script code can be placed in both head & body section of html page.

**Script in head section**

```
<html>
<head>
<script type=-text / JavaScript>
 Script code here
</script>
</head>
<body>
</body>
</html>
```

**Script in body section**

```
<html>
<head>
</head>
<body>
 <script type= -text / JavaScript>
 Script code here
 </script>
</body>
</html>
```

**Scripting in both head & body section:** - we can create unlimited number of scripts inside the same page. So we can locate multiple scripts in both head & body section of page.

**Ex:** - <html>  
 <head>  
 <script type=-text / JavaScript>  
 Script code here  
 </script>  
 </head>  
 <body>  
 <script type=-text / JavaScript>  
 Script code here  
 </script>  
 </body>  
</html>

**Program:** -

```
<html>
<head>
<script language="JavaScript">
document.write("hai my name is Kalpana")
</script>
</head>
<body text="red">
<marquee>
<script language="JavaScript">
document.write("hai my name is Sunil Kumar Reddy")
</script> </marquee>
</body>
</html>
```

**O/P:** - hai my name is Kalpana

hai my name is Sunil Kumar Reddy

**document.write** is the proper name of object.

→ There are 2 ways of executing script code

- 1) direct execute
- 2) to execute script code dynamically

**Reacts to events:** - JavaScript can be set to execute when something happens. When the page is finished loading in browser window (or) when the user clicks on html element dynamically.

**Ex: -**

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional // EN">
<HTML>
<HEAD>
<script language="JavaScript">
function myf()
{
document.write("Hai Kalpana")
}
</script>
</HEAD>
<BODY>
to execute script code:
<input type="button" value="click me" onclick="myf()">
To execute script code:
<input type="button" value="touch me" onmouseover="myf()">
</BODY>
</HTML>
```

**O/P: - to execute script code:**

click me

**To execute script code:**

touch me

**Creating external script:** - some times you might want to run same script on several pages without having to write the script on each page. To simplify this, write external script & save .js extension. To use external script specify .js file in src attribute of script tag.

**Note:** - external script can not contain script tag.

**save: - external.js**

```
document.write("this is external script code 1 "+"
");
document.write("this is external script code 2 "+"
");
document.write("this is external script code 3 "+"
");
document.write("this is external script code 4 ");
```

```
<HTML> <BODY>
<script language="JavaScript">
document.write("this is document code 1 "+"
");
document.write("this is document code 2 "+"
");
</script>
<script src="external.js">
</script>
</BODY>
</HTML>
```

**O/P: -**

```
this is document code 1
this is document code 2
this is external script code 1
this is external script code 2
this is external script code 3
this is external script code 4
```

**JavaScript syntax rules:** - JavaScript is case sensitive language. In this upper case lower case letters are differentiated (not same).

**Ex:** - a=20;  
A=20;

Those the variable name „a“ is different from the variable named „A“.

**Ex:** - myf( ) // correct  
myF( ) // incorrect

→ ; is optional in general JavaScript.

**Ex:** - a=20 // valid  
b=30 // valid  
A=10; b=40; // valid

However it is required when you put multiple statements in the same line.

→ JavaScript ignore white space. In java script white space, tag space & empty lines are not preserved.  
→ To display special symbols we use \.

**Comment lines:** - comments lines are not executable.

// single line comment  
/\* this is multi line comment \*/

**Declaring variable:** - variable is a memory location where data can be stored. In java script variables with any type of data are declared by using the keyword `_var_`. All keywords are small letters only.

```
var a; a=20;
var str; str= "Sunil";
var c; c="a";
var d; d=30.7;
```

But the keyword is not mandatory when declare of the variable.

c; → not valid. In this solution var keyword must be declared.  
→ During the script, we can change value of variable as well as type of value of variable.

**Ex:** -  
a=20;  
a=30.7;

## **JavaScript functions:** -

In javascript functions are created with the keyword `_function_` as shown below

**Syntax:** - function funname()  
    {  
        -----  
    }

Generally we can place script containing function head section of web page. There are 2 ways to call the function.

- 1) direct call function
- 2) Events handlers to call the function dynamically.

1→ We can pass data to function as argument but that data will be available inside the function.



**Ex: -**

```

<HTML>
<HEAD>
<TITLE> Function direct call</TITLE>
<script language="JavaScript">
function add(x,y)
{
z=x+y
return z
}
</script>

```

```

</HEAD>
<BODY>
<script>
var r=add(30,60)
document.write("addition is :"+r);
</script>
</BODY>
</HTML>

```

**O/P: - addition is :90**

2→ to add dynamical effects, java script provide a list of events that call function dynamically. Hare each event is one attribute that always specified in html tags.

```

attrname="attrval"
eventName="funname()"

```

**Ex: -**

```

<HTML>
<HEAD>
<TITLE> Function dynamically</TITLE>
<script language="JavaScript">
function add()
{
x=20
y=30
z=x+y
document.write("addition is :"+z);
}
</script>

```

```

</HEAD>
<BODY> to call function:
<input type="button" value="click hare"
onclick="add()">
</script>
</BODY>
</HTML>

```

**O/P: - to call function:  
addition is :90**


Events are not case sensitive.

**Java script events: -****Attribute**

onclick  
 ondblclick  
 onmouseover  
 onmousedown  
 onmousemove  
 onmouseout  
 onmouseup  
 onkeydown  
 onkeypress  
 onkeyup  
 onfocus  
 onblur  
 onchange  
 onselect  
 onload  
 onunload

**The event occurs when...**

mouse click an object  
 mouse double clicks  
 a mouse cursor on touch here  
 a mouse button is pressed  
 the mouse is moved  
 the mouse is moved out an element  
 a mouse button is released  
 a keyboard key is pressed  
 a keyboard key is pressed or held down  
 a keyboard key is released  
 an elements get focus  
 an element loses focus  
 the content of a fieldchange  
 text is selected  
 a page or an image is finishedloading  
 the user exist the page

<b>onerror</b>	<b>an error occurs when loading a document or an image</b>
<b>onabort</b>	<b>loading an image is interrupted</b>
<b>onresize</b>	<b>a window or frame is resized</b>
<b>onreset</b>	<b>the reset button is pressed</b>
<b>onsubmit</b>	<b>the submit button is clicked</b>

**Ex: -**

```

<HTML>
<HEAD>
<TITLE> Mouse Events </TITLE>
<script language="JavaScript">
function add()
{
a=55
b=45
c=a+b
document.write("addition is :"+c)
}
</script>
</HEAD>
<BODY>
<b onclick="add()">
to call function click here :

<b onmouseover="add()">
to call function touch here :

<b ondblclick="add()">
to call function double click here :


```

```


<b onmousemove="add()">
to call function cursor move here :

<b onmouseup="add()">
to call function cursor up here :

<b onmouseout="add()">
to call function cursor out here :

</BODY>
</HTML>

```

**O/P: -**

```

to call function click here :
to call function touch here :
to call function double click here :
 addition is :100
to call function cursor move here :
to call function cursor up here :
to call function cursor out here :

```

**Program: -**

```

<HTML>
<HEAD>
<TITLE> display student name </TITLE>
<script language="JavaScript">
function disp()
{
// access from data
var name=window.document.student.sname.value
// (or) var name=window.document.getElementById("snameid").value
//checking name
if(name=""||!isNaN(name)||!isNaN(name.charAt(0)))
 window.alert("sname you entered is invalid")
else
 document.write("sname you have entered is : "+name);
}

```

```

</script>
</HEAD>
<BODY>
<form name="student">
Enter Student name:
<input type="text" name="sname" id="snameid" value="enter" onblur="disp()">
</form>
</BODY>
</HTML>

```

O/P: -

Enter Student name:

Enter Student name:   
sname you have entered is : true



**Popup boxes:** - popup (arises) box is a small window that always shown before opening the page. The purpose of popup box is to write message, accept some thing from user. Java script provides 3 types of popup boxes. They are **1) alert 2) Confirm. 3) Prompt.**

### **1) alert popup box :-**

Alert box is a very frequently useful to send or write cautionary messages to end use alert box is created by alert method of window object as shown below.

**Syntax: - window – alert (“message”);**

When alert popup, the user has to click ok before continue browsing.

**Ex: -**

```

<html>
<head>
<title> alert box </title>
<script language="JavaScript">
function add()
{
a=20
b=40
c=a+b

```

```

window.alert("This is for addition of 2
no's")
document.write("Result is: "+c)
}
</script>
</head>
<body onload="add()">
</body>
</html>

```

O/P: -



Result is: 60

**2) confirm popup box:-**

This is useful to verify or accept some thing from user. It is created by confirm method of window object as shown below.

**Syntax:-** `window.confirm ("message?");`

When the confirm box pop's up, user must click either ok or cancel buttons to proceed. If user clicks ok button it returns the boolean value true. If user clicks cancel button, it returns the boolean value false.

**Ex: -**

```
<HTML>
<HEAD>
<TITLE> Confirm </TITLE>
<script>
function sub()
{
a=50
b=45
c=a-b
x=window.confirm("Do you want to see
subtraction of numbers")
if(x==true)
{
```

```
document.write("result is :"+c)
}
else
{
document.write("you clicked cancel button")
}
}
</script>
</HEAD>
<BODY onload="sub()">
to see the o/p in pop up box:
</BODY>
</HTML>
```

**O/P: -**

to see the o/p in pop up box:

result is :5



**3) Prompt popup box:-** It is useful to accept data from keyboard at runtime. Prompt box is created by prompt method of window object.

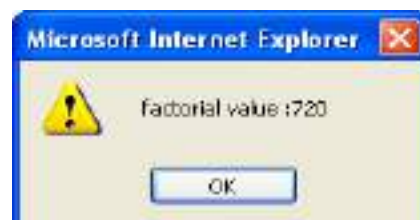
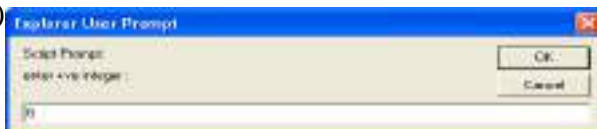
**Syntax:-** `window.prompt ("message", "default text");`

When prompt dialog box arises user will have to click either ok button or cancel button after entering input data to proceed. If user click ok button it will return input value. If user click cancel button the value -null will be returned.

**Ex: -**

```
<HTML>
<HEAD>
<TITLE> Prompt </TITLE>
<script>
function fact()
{
var b=window.prompt("enter +ve integer
:", "enter here")
var c=parseInt(b)
a=1
for(i=c; i>=1; i--)
```

```
{
a=a*i
}
window.alert("factorial value :"+a)
}
</script>
</HEAD>
<BODY onload="fact()">
</BODY>
</HTML>
```



**O/P: -**

**FORM VALIDATION:**

When we create forms, providing form validation is useful to ensure that your customers enter valid and complete data. For example, you may want to ensure that someone inserts a valid e-mail address into a text box, or perhaps you want to ensure that someone fills in certain fields.

We can provide custom validation for your forms in two ways: server-side validation and client-side validation.

**SERVER-SIDE VALIDATION**

In the server-side validation, information is being sent to the server and validated using one of server-side languages. If the validation fails, the response is then sent back to the client, page that contains the web form is refreshed and a feedback is shown. This method is secure because it will work even if JavaScript is turned off in the browser and it can't be easily bypassed by malicious users. On the other hand, users will have to fill in the information without getting a response until they submit the form. This results in a slow response from the server.

The exception is validation using Ajax. Ajax calls to the server can validate as you type and provide immediate feedback. Validation in this context refers to validating rules such as username availability.

**Server side validation** is performed by a web server, after input has been sent to the server.

**CLIENT-SIDE VALIDATION**

Server-side validation is enough to have a successful and secure form validation. For better user experience, however, you might consider using client-side validation. This type of validation is done on the client using script languages such as JavaScript. By using script languages user's input can be validated as they type. This means a more responsive, visually rich validation.

With client-side validation, form never gets submitted if validation fails. Validation is being handled in JavaScript methods that you create (or within frameworks/plugins) and users get immediate feedback if validation fails.

Main drawback of client-side validation is that it relies on JavaScript. If users turn JavaScript off, they can easily bypass the validation. This is why validation should always be implemented on both the client and server. By combining server-side and client-side methods we can get the best of the two: fast response, more secure validation and better user experience.

**Client side validation** is performed by a web browser, before input is sent to a web server.

```
<html>
<head>
<title>Form Validation</title>
<script type="text/javascript">
function validate()
{
var n = document.myForm.Name.value;
if(n == "")
{
alert("Please fill the username");
```

```
document.myForm.Name.focus();
exit;
}
if(n.length< 3 || n.length >= 8)
{
alert("Please enter valid name and minimum length 3 characters and maximum length 8 characters !");
document.myForm.Name.focus();
exit;
}
var email = document.myForm.EMail.value;
if(email == "")
{
alert("Please provide your Email!");
document.myForm.EMail.focus() ;
document.myForm.Email.onfocus();
exit;
}
if(email=="||email.indexOf('@')<1||email.lastIndexOf('.')<3)
{
 alert("Please Enter valid Email");
 document.myForm.Email.onfocus();
 exit;
}
var z = document.myForm.Zip.value;
if(z == "")
{
alert("Please provide a pin");
document.myForm.Zip.focus() ;
exit;
}
if(isNaN(z) || z.length != 6)
{
alert("Please provide a pi in the format #####");
document.myForm.Zip.focus() ;
exit;
}
document.writeln("successfully registered");
}
</script>
</head>
<body bgcolor="bisque">
<h1><p align="center"> Application Form Validation Using JavaScript</p>
</h1>
<form action="reg.html" name="myForm" ">
Name
<td><input type="text" name="Name"/>

EMail<input type="text" name="EMail"/>

Zip Code
<input type="text" name="Zip" size="50" />

<input type="button" value="Submit" onclick="validate()">
</form>
</body>
</html>
```

**TOPICS:**

**UNIT – II**

**Introduction to PHP**

- Declaring Variables
- Data Types
- Operators
- Control Structures
- Functions
- 

Reading data from WEB form controls  
like text boxes, radio buttons, lists etc..  
Handling File Uploads

- Handling Sessions and Cookies

**Introduction to XML**

- Basic XML document
- Presenting XML
- Document Type Definition(DTD)
- XML Schemas
- Document Object Model(DOM)
- Introduction to XHTML
- Using XML Processors: DOM and SAX

## PHP INTRODUCTION

PHP started out as a small open source project that evolved as more and more people found out how useful it was. **Rasmus Lerdorf** unleashed the first version of PHP way back in **1994**.

- PHP is a recursive acronym for "**PHP: Hypertext Preprocessor**".
- PHP is a **server side scripting language** that is embedded in HTML. PHP scripts are executed on the server
- It is used to manage dynamic content, databases, session tracking, even build entire e-commerce sites.
- PHP supports many databases (MySQL, Informix, Oracle, Sybase, Solid, PostgreSQL, Generic ODBC, Microsoft SQL Server , etc.)
- PHP is an open source software.
- PHP is pleasingly zippy in its execution, especially when compiled as an Apache module on the Unix side. The MySQL server, once started, executes even very complex queries with huge result sets in record-setting time.
- PHP supports a large number of major protocols such as POP3, IMAP, and LDAP.
- PHP is forgiving: PHP language tries to be as forgiving as possible.
- **PHP Syntax is C-Like.**

### SYNTAX OVERVIEW:

**Canonical PHP tags** *The most universally effective PHP tag style is:*  
<?php...?>

**Short-open (SGML-style) tags** *Short or short-open tags look like this:*  
<?...?>

**HTML script tags** *HTML script tags look like this:*  
<script language="PHP">...</script>

### PHP - VARIABLE TYPES

The main way to store information in the middle of a PHP program is by using a **variable**. Here are the most important things to know about variables in PHP.

- A variable is used to store information.
- All variables in PHP are denoted with a leading dollar sign (\$).
- The value of a variable is the value of its most recent assignment.
- Variables are assigned with the = **operator**, with the variable on the left-hand side and the expression to be evaluated on the right.
- Variables can, but do not need, to be declared before assignment.
- Variables used before they are assigned have default values.
- PHP does a good job of **automatically converting types from one to another** when necessary.
- PHP variables are Perl- like.

**Syntax:** \$var\_name = value;

**Eg:** creating a variable containing a string, and a variable containing a number:



```
$txt="HelloWorld!";
$x=16;
?>
```

### PHP is a Loosely Typed Language:

- ✓ In PHP, a variable does not need to be declared before adding a value to it.
- ✓ You do not have to tell PHP which data type the variable is
- ✓ PHP automatically converts the variable to the correct data type, depending on its value.

### Variable Naming

Rules for naming a variable is-

- Variable names must begin with a letter or underscore character.
- A variable name can consist of numbers, letters, underscores but you cannot use characters like +, -, %, (, ), . &, etc

**There is no size limit for variables.**

### PHP - Data Types:

PHP has a total of **eight data types** which we use to construct our variables:

- **Integers:** are whole numbers, without a decimal point, like 4195.
- **Doubles:** are floating-point numbers, like 3.14159 or 49.1. **Scalar types**
- **Booleans:** have only two possible values either true or false.
- **Strings:** are sequences of characters, like 'PHP supports string operations.'
- **Arrays:** are named and indexed collections of other values.
- **Objects:** are instances of programmer-defined classes. **Compound types**
- **NULL:** is a special type that only has one value: NULL.
- **Resources:** are special variables that hold references to resources external to PHP (such as database connections). **Special types**

The first four are simple types, and the next two (arrays and objects) are compound - the compound types can package up other arbitrary values of arbitrary type, whereas the simple types cannot.

### 1. PHP Integers

Integers are **primitive data types**. They are **whole numbers**, without a decimal point, like 4195. They are the simplest type. They correspond to simple whole numbers, both positive and negative {..., -2, -1, 0, 1, 2, ...}.

Integer can be in decimal (base 10), octal (base 8), and hexadecimal (base 16) format. Decimal format is the default, octal integers are specified with a leading 0, and hexadecimal have a leading 0x.

**Ex:** \$v = 12345;  
\$var1 = -12345 + 12345;

#### notation.php

```
<?php
$var1 = 31; $var2 = 031; $var3 = 0x31;
echo "$var1\n$var2\n$var3"; ?>
```

#### Output:

```
31
25
49
```

**Web Technologies** is the **decimal**. The script shows these three numbers in **decimal**, **integer** and **C**, if an integer value is bigger than the maximum value allowed, integer overflow happens. PHP works differently. In PHP, the integer becomes a float number. Floating point numbers have greater boundaries. In 32bit system, an integer value size is four bytes. The maximum integer value is 2147483647.

### **boundary.php**

```
<?php
$var = PHP_INT_MAX;
echo var_dump($var);
$var++;
echo var_dump($var);
?>
```

We assign a maximum integer value to the \$var variable. We increase the variable by one. And we compare the contents.

#### **Output:**

```
int(2147483647)
float(2147483648)
```

As we have mentioned previously, internally, the number becomes a floating point value.

**var\_dump():** The PHP var\_dump() function returns the data type and value.

## **2.PHP Doubles or Floating point numbers**

Floating point numbers represent real numbers in computing. Real numbers measure continuous quantities like weight, height or speed. Floating point numbers in PHP can be larger than integers and they can have a decimal point. The size of a float is platform dependent.

We can use various syntaxes to create floating point values.

```
<?php
$a = 1.245;
$b = 1.2e3;
$c = 2E-10;
$d = 1264275425335735;
var_dump($a);
var_dump($b);
var_dump($c);
var_dump($d);
?>
```

The **\$d** variable is assigned a large number, so it is automatically converted to float type.

#### **Output:**

```
float(1.245)
float(1200)
float(2.0E-10)
float(1264275425340000)
```

This is the output of beside script

### 3.PHP Boolean

A Boolean represents two possible states: TRUE or FALSE.

```
$x = true; $y = false;
```

Booleans are often used in conditional testing.

```
<?php
$male = False;
$r = rand(0, 1);
$male = $r ? True:
False; if ($male) {
 echo "We will use name John\n";
} else {
 echo "We will use name Victoria\n";
} ?>
```

The script uses a **random integer** generator to simulate our case. `$r = rand(0, 1);`

The **rand( )** function returns a random number from the given integer boundaries **0 or 1**.

**\$male = \$r? True: False;**

We use the ternary operator to set a \$male variable. The variable is based on the random \$r value. If \$r equals to **1**, the \$male variable is set to **True**. If \$r equals to **0**, the \$male variable is set to **False**.

### 4.PHP Strings

String is a data type representing textual data in computer programs. Probably the single most important data type in programming.

```
<?php
$a = "PHP ";
$b = 'PERL';
echo $a . $b; ?>
```

**Output: PHP PERL**

**We can use single quotes and double quotes to create string literals.**

The script outputs two strings to the console. The \n is a special sequence, a new line.

**The escape-sequence replacements are –**

- \n is replaced by the newline character
- \r is replaced by the carriage-return character
- \t is replaced by the tab character
- \\$ is replaced by the dollar sign itself (\$)
- \" is replaced by a single double-quote (")
- \\ is replaced by a single backslash (\)

### The Concatenation Operator

There is only one string operator in PHP.

The concatenation operator ( . ) is used to put two string values together. To concatenate two string variables together, use the concatenation operator:

```
<?php
$txt1="Hello Kalpana!";
$txt2="What a nice day!";
echo $txt1 . " " . $txt2;
?> O/P: Hello Kalpana! What a nice day!
```

**Search for a Specific Text within a String**

The **PHP strpos() function** searches for a specific text within a string. If a **match is found**, the function **returns the character position of the first match**. If **no match is found**, it will return **FALSE**. The example below searches for the text "world" in the string "Hello world!":

**Example**

```
<?php
echo strpos("Hello world!", "world");
?>
```

**output: 6**

**Tip:** The first character position in a string is 0 (not 1).

**Replace Text within a String**

The PHP **str\_replace()** function replaces some characters with some other characters in a string. The example below replaces the text "world" with "Dolly":

**Example**

```
<?php
echo str_replace("world", "Kalpana", "Hello world!");
?>
```

**Output: Hello Kalpana!**

**The strlen() function:**

The **strlen()** function is used to return the length of a string. Let's find the length of a string:

Eg: <?php

```
echo strlen("Hello world!"); ?>
```

**The output of the code above will be: 12**

**5.PHP Array**

Array is a complex data type which handles a collection of elements. Each of the elements can be accessed by an index. An array stores multiple values in one single variable. In the following example \$cars is an array. The PHP var\_dump() function returns the data type and value:

**Example**

```
<?php
$cars = array("Volvo", "BMW", "Toyota");
print_r($cars);
var_dump($cars);
?>
```

The **array keyword** is used to create a collection of elements. In our case we have names. The print\_r function prints human readable information about a variable to the console.

```
O/P: Array ([0] => Volvo [1] => BMW [2] => Toyota)
array(3) { [0]=> string(5) "Volvo" [1]=> string(3) "BMW" [2]=> string(6) "Toyota" }
```

**6.PHP Object**

An object is a data type which stores data and information on how to process that data. In PHP, an object must be explicitly declared. First we must declare a class of object. For this, we use the class keyword. A class is a structure that can contain properties and methods:

**Example**

```
<?php
class Car {
 function Car() {
 $this->model = "VW";
 }
}
$herbie = new Car(); // create an object
echo $herbie->model; // show object properties
?>
```

**Output: VW**

## 7.PHP NULL

NULL is a special data type that only has **one value: NULL**. To give a variable the NULL value, simply assign it like this –

**Ex: \$my\_var = NULL;**

The special constant NULL is capitalized by convention, but actually it is case insensitive; you could just as well have typed –

**\$my\_var = null;**

A variable that has been assigned NULL has the following properties –

- It evaluates to FALSE in a Boolean context.
- It returns FALSE when tested with **IsSet()** function.

**Tip:** If a variable is created without a value, it is automatically assigned a value of NULL. Variables can also be emptied by setting the value to NULL:

### Example1

```
<?php
$x = "Hello world!";
$x = null;
var_dump($x);
?>
```

## 8.PHP Resource

The special resource type is not an actual data type. It is the storing of a reference to functions and resources external to PHP. A common **example** of using the resource data type is a **database call**. Resources are handlers to opened files, database connections or image canvas areas. We will not talk about the resource type here, since it is an advanced topic.

### constant() function

As indicated by the name, this function will return the value of the constant. This is useful when you want to retrieve value of a constant, but you do not know its name, i.e. It is stored in a variable or returned by a function.constant() example

```
<?php
define("MINSIZE", 50);
echo MINSIZE;
echo constant("MINSIZE"); // same thing as the previous line
?>
```

**Output: 50 50**

Only scalar data (boolean, integer, float and string) can be contained in constants.

## PHP - Operators:

### What is Operator?

Simple answer can be given using expression  $4 + 5$  is equal to 9. Here **4 and 5** are called **operands** and **+** is called **operator**. PHP language supports following type of operators.

Arithmetic Operators	Assignment Operators
Increment/Decrement operators	Conditional (or ternary) Operators
Comparison Operators	String Operators
Logical (or Relational) Operators	Array Operators

### Arithmetic Operators:

There are following arithmetic operators supported by PHP language:

Assume variable **A** holds **10** and variable **B** holds **20** then:

-	Subtracts second operand from the first	$\$A - \$B$ will give -10
/	Divide numerator by denominator	$\$B / \$A$ will give 2
**	Exponentiation ( $\$x$ to the $\$y$ 'th power)	$\$A ** \$B$

### Increment/Decrement operators

--	Decrement operator, decreases integer value by one	$\$A--$ will give 9 / $--\$A$
----	----------------------------------------------------	-------------------------------

### Comparison Operators:

There are following comparison operators supported by PHP language Assume variable A holds 10 and variable B holds 20 then:

===	Identical(Returns true if $\$A$ is equal to $\$B$ , and they are of the same type)	$\$A === \$B$
<>	Returns true if $\$x$ is not equal to $\$y$	$\$A < \$B$
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	$(\$A > \$B)$ is not true.

>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then returns true.	(\$A >= \$B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(\$A <= \$B) is true.

**Logical Operators:**

There are following logical operators supported by PHP language  
Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
<b>and (or) &amp;&amp;</b>	Called Logical AND operator. If both the operands are true then then condition becomes true.	(\$A and \$B) is true. (\$A && \$B) is true.
<b>or (or)   </b>	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(\$A or \$B) is true. (\$A    \$B) is true.
<b>!</b>	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!( \$A && \$B) is false.

**Assignment Operators:**

There are following assignment operators supported by PHP language:

Operator	Description	Example
<b>=</b>	Simple assignment operator, Assigns values from right side operands to left side operand	\$C = \$A + \$B
<b>+=</b>	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	\$C += \$A is equivalent to \$C = \$C + \$A
<b>-=</b>	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	\$C -= \$A is equivalent to \$C = \$C - \$A
<b>*=</b>	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	\$C *= \$A is equivalent to \$C = \$C * \$A
<b>/=</b>	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	\$C /= \$A is equivalent to \$C = \$C / \$A
<b>%=</b>	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	\$C %= \$A is equivalent to \$C = \$C % \$A

**Conditional Operator**

There is one more operator called conditional operator. This first evaluates an expression for a true or false value and then execute one of the two given statements depending upon the result of the evaluation.

**The conditional operator has this syntax:**

Operator	Description	Example
<b>? :</b>	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

**PHP String Operators**

PHP has two operators that are specially designed for strings.

Operator	Description	Example
.	Concatenation	\$txt1 . \$txt2 (Concatenation of \$txt1 and \$txt2)
.=	Concatenation assignment	\$txt1 .= \$txt2 (Appends \$txt2 to \$txt1)

### PHP Array Operators

The PHP array operators are used to compare arrays.

Operator	Description	Example
+	Union	\$x + \$y (Union of \$x and \$y)
==	Equality	\$x == \$y (Returns true if \$x and \$y have the same key/value pairs)
===	Identity	\$x === \$y (Returns true if \$x and \$y have the same key/value pairs in the same order and of the same types)
!= or <>	Inequality	\$x != \$y or \$x <> \$y Returns true if \$x is not equal to \$y
!==	Non-identity	\$x !== \$y (Returns true if \$x is not identical to \$y)

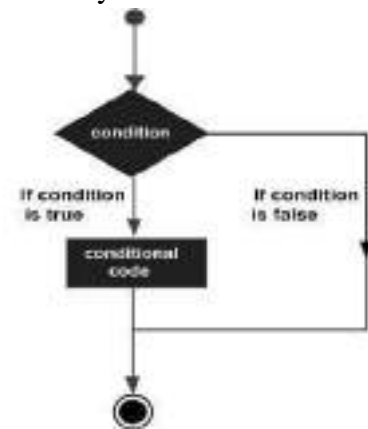
Category	Operator	Associativity
Unary	! ++ --	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Logical AND	&&	Left to right
Logical OR		Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=	Right to left



## PHP -CONTROL STRUCTURES

The if, elseif ...else and switch statements are used to take decision based on the different condition. You can use conditional statements in your code to make your decisions. PHP supports following three decision making statements –

- **if...else statement** – use this statement if you want to execute a set of code when a condition is true and another if the condition is not true
- **elseif statement** – is used with the if...else statement to execute a set of code if **one** of the several condition is true
- **switch statement** – is used if you want to select one of many blocks of code to be executed, use the Switch statement. The switch statement is used to avoid long blocks of if..elseif..else code.



### Syntax

if (*condition*)

*code to be executed if condition is true;*

else

*code to be executed if condition is false;*

### Example

The following example will output "Have a nice weekend!" if the current day is Friday, Otherwise, it will output "Have a nice day!":

**EX:** <html>

<body>

<?php

\$d=date("D");

if (\$d=="Fri")

echo "Have a nice weekend!";

else

echo "Have a nice day!";

?>

</body> </html>

**The If...Else Statement**

If you want to execute some code if a condition is true and another code if a condition is false, use the if...else statement.

**The elseif Statement**

If you want to execute some code if one of the several conditions is true use the elseif statement

<b>Syntax</b>	<b>EX:</b> <html>
if ( <i>condition</i> ) <i>code to be executed if condition is true;</i>	<body>
elseif ( <i>condition</i> ) <i>code to be executed if condition is true;</i>	<?php
else <i>code to be executed if condition is false;</i>	\$d=date("D");
	if (\$d=="Fri")
	echo "Have a nice weekend!";
	elseif (\$d=="Sun")
	echo "Have a nice Sunday!";
	else
	echo "Have a nice day!";
	?>
	</body>
	</html>
<b>Example</b> The following example will output "Have a nice weekend!" if the current day is Friday, and "Have a nice Sunday!" if the current day is Sunday. Otherwise, it will output "Have a nice day!"	

**The Switch Statement**

If you want to select one of many blocks of code to be executed, use the Switch statement. The switch statement is used to avoid long blocks of if..elseif..else code.

<b>Syntax</b>	<b>Example</b>
switch ( <i>expression</i> ) { case <i>label1</i> : <i>code to be executed if expression = label1;</i> break; case <i>label2</i> : <i>code to be executed if expression = label2;</i> break; default: <i>code to be executed if expression is different from both label1 and label2;</i> }	The <i>switch</i> statement works in an unusual way. First it evaluates given expression then seeks a label to match the resulting value. If a matching value is found then the code associated with the matching label will be executed or if none of the label matches then statement will execute any specified default code.

**PHP -Loop Types**

Loops in PHP are used to execute the same block of code a specified number of times. PHP supports following four loop types.

- **for** – loops through a block of code a specified number of times.
- **while** – loops through a block of code if and as long as a specified condition is true.
- **do. while** – loops through a block of code once, and then repeats the loop as long as a special condition is true.

- **foreach** – loops through a block of code for each element in an array.

We will discuss about **continue** and **break** keywords used to control the loops execution.

### The for loop statement

The for statement is used when you know how many times you want to execute a statement or a block of statements.

### Syntax

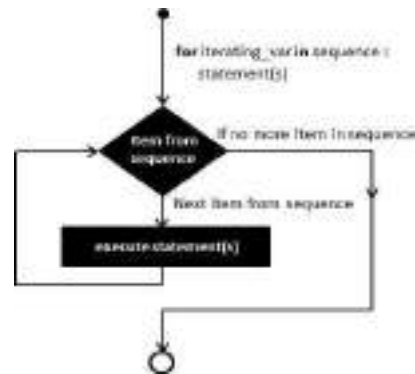
```
for (initialization; condition; increment)
{
 code to be executed;
}
```

The initializer is used to set the start value for the counter of the number of loop iterations. A variable may be declared here for this purpose and it is traditional to name it \$i.

### Example

The following example makes five iterations and changes the assigned value of two variables on each pass of the loop –

```
<html> <body>
 <?php
 $a = 0;
 $b = 0;
 for($i=0; $i<5; $i++)
 {
 $a += 10;
 $b += 5;
 }
 echo ("At the end of the loop a=$a and b=$b");
?> </body> </html>
```



This will produce the following result –

At the end of the loop a=50 and b=25

### The while loop statement

The while statement will execute a block of code if and as long as a test expression is true. If the test expression is true then the code block will be executed. After the code has executed the test expression will again be evaluated and the loop will continue until the test expression is found to be false.

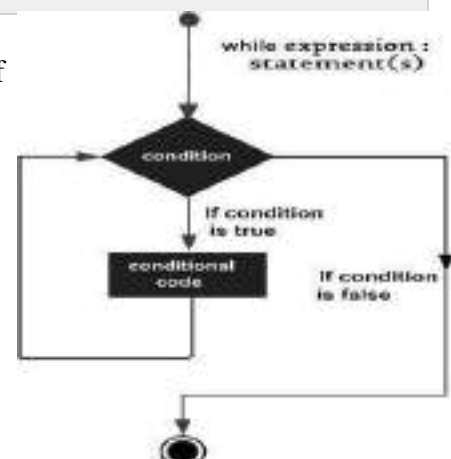
### Syntax

```
while (condition)
{
 code to be executed;
}
```

### Example

This example decrements a variable value on each iteration of the loop and the counter increments until it reaches 10 when the evaluation is false and the loop ends.

```
<html>
<body>
 <?php
 $i = 0;
```



```

$num = 50;
while($i < 10)
{
 $num--;
 $i++;
}
echo ("Loop stopped at i = $i and num = $num");
?> </body> </html>

```

**This will produce the following result –**

Loop stopped at i = 10 and num = 40

### The do...while loop statement

The do...while statement will execute a block of code at least once. It then will repeat the loop as long as a condition is true.

### Syntax

```

do
{
 code to be executed;
}while (condition);

```

### Example

The following example will increment the value of i at least once, and it will continue incrementing the variable i as long as it has a value of less than 10 –

```

<html> }while($i < 10);
<body> echo ("Loop stopped at i = $i");
<?php ?>
 $i = 0; $num = 0; </body>
 do{ </html>
 $i++;

```

**O/P:** Loop stopped at i = 10

### The foreach loop statement

The foreach statement is used to **loop through arrays**. For each pass the value of the current array element is assigned to \$value and the array pointer is moved by one and in the next pass next element will be processed.

### Syntax

```

foreach (array as value)
{
 code to be executed;
}

```

### Example

Try out beside example to list out the values of an array.

```

<html>
<body>
<?php
 $array = array(1, 2, 3, 4, 5);
 foreach($array as $value)
 {
 echo "Value is $value
";
 }
?>
</body> </html>

```

**This will produce the following result –**

Value is 1  
Value is 2  
Value is 3  
Value is 4  
Value is 5

### The break statement

The PHP **break** keyword is used to terminate the execution of a loop prematurely. The **break** statement is situated inside the statement block. It gives you full control and whenever you want to exit from the loop you can come out. After coming out of a loop immediate statement to the loop will be executed.

#### Example

In the following example condition test becomes true when the counter value reaches 3 and loop terminates.

```
<?php
$i = 0;
while($i < 10) {
 $i++;
 if($i == 3)break; }
echo ("Loop stopped at i = $i");
?> O/P: Loop stopped at i=3
```

### The continue statement

The PHP **continue** keyword is used to halt the current iteration of a loop but it does not terminate the loop. Just like the **break** statement the **continue** statement is situated inside the statement block containing the code that the loop executes, preceded by a conditional test. For the pass encountering **continue** statement, rest of the loop code is skipped and next pass starts.

#### Example

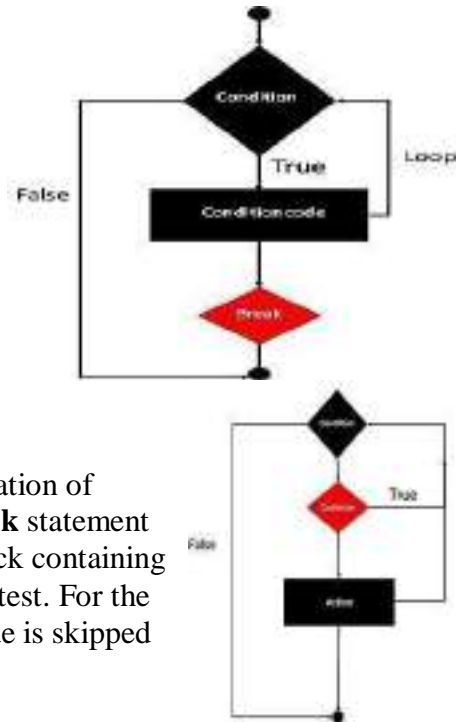
In the following example loop prints the value of array but for which condition becomes true it just skip the code and next value is printed.

```
<html>
<body>
<?php
$nos = array(1, 2, 3, 4, 5);
foreach($nos as $value)
{
 if($value == 3) continue;
 echo "Value is $value
";
}

?>
</body>
</html>
```

This will produce the

```
Value is 1
Value is 2
Value is 4
Value is 5
```



## PHP – Functions

PHP functions are similar to other programming languages. A function is a piece of code which takes one more input in the form of parameter and does some processing and returns a value. You already have seen many functions like **fopen()** and **fread()** etc. They are built-in functions but PHP gives you option to create your own functions as well.

There are two parts which should be clear to you –

- **Creating a PHP Function**
- **Calling a PHP Function**

In fact you hardly need to create your own PHP function because there are already more than 1000 of built-in library functions created for different area and you just need to call them according to your requirement.

### Creating PHP Function

It's very easy to create your own PHP function. Suppose you want to create a PHP function which will simply write a simple message on your browser when you will call it. Following example creates a function called `writeMessage()` and then calls it just after creating it.

```
<html> <head>
 <title>Writing PHP Function</title>
</head>
<body>
 <?php
 /* Defining a PHP Function */
 function writeMessage()
 {
```

```
 echo "Have a nice time Kalpana!";
 } /* Calling a PHP Function */
 writeMessage();
 ?>
</body>
</html>
```

**Output:** Have a nice time Kalpana!

### PHP Functions with Parameters

PHP gives you option to pass your parameters inside a function. You can pass as many as parameters you're like. These parameters work like variables inside your function. Following example takes two integer parameters and add them together and then print them.

```
<html>
<head> <title>Writing PHP Function with Parameters</title> </head>
<body>
 <?php
 function addFunction($num1, $num2)
 {
 $sum = $num1 + $num2;
 echo "Sum of the two numbers is : $sum";
 }
 addFunction(10, 20);
 ?> </body> </html>
```

**Output:** Sum of the two numbers is : 30

### Passing Arguments by Reference

It is possible to pass arguments to functions by reference. This means that a reference to the variable is manipulated by the function rather than a copy of the variable's value. Any

```
<html>
 <head>
 <title>Passing Argument by Reference</title>
 </head>
 <?php
 function addFive($num)
 {
```

changes made to an argument in these cases will change the value of the original variable. You can pass an argument by reference by adding an ampersand to the variable name in either the function call or the function definition.

```
$num += 5;
}
function addSix(&$num)
{
 $num += 6;
}
$orignum = 10;
addFive($orignum);
echo "Original Value is $orignum
";
addSix($orignum);
echo "Original Value is $orignum
";
?>
</body>
</html>
```

**Output:** Original Value is 10  
Original Value is 16

### PHP Functions returning value

A function can return a value using the **return** statement in conjunction with a value or object. return stops the execution of the function and sends the value back to the calling code. You can return more than one value from a function using **return array(1,2,3,4)**.

```
<html> <head> <title>Writing PHP Function which returns value</title> </head>
<body>
<?php
 function addFunction($num1, $num2)
 {
 $sum = $num1 + $num2;
 return $sum;
 }
 $return_value = addFunction(10, 20);
 echo "Returned value from the function : $return_value";
?> </body> </html>
```

**Output:** Returned value from the function : 30

### Setting Default Values for Function Parameters

You can set a parameter to have a default value if the function's caller doesn't pass it. Following function prints NULL in case use does not pass any value to this function.

```
<html> <head> <title>Writing PHP Function which returns value</title> </head>
<body>
<?php
 function printMe($param = NULL)
 {
 print $param;
 }
 printMe("This is test");
 printMe();
?>
</body> </html>
```

**Output:** This is test

### Dynamic Function Calls

It is possible to assign function names as strings to variables and then treat these variables exactly as you would the function name itself.

```
<html>
<head>
<title>Dynamic Function Calls</title>
</head>
<body>
<?php
 function sayHello()
 {
 echo "Hello
";
 }
 $function_holder = "sayHello";
 $function_holder();
?> </body> </html>
```

**Output:** Hello

```
<html>
<head>
<title>Dynamic Function Calls</title>
</head>
<body>
<?php
 function add($x,$y)
 {
 echo "addition=" . ($x+$y);
 }
 $function_holder = "add";
 $function_holder(20,30);
?> </body> </html>
```

**Output:** addition=50

### PHP Default Argument Value

The following example shows how to use a default parameter. If we call the function setHeight() without arguments it takes the default value as argument:

#### Example

```
<?php
function setHeight($minheight = 50) {
 echo "The height is : $minheight \t";
}
setHeight(350);
setHeight(); // will use the default value of 50
setHeight(135);
setHeight(80);
?>
```

**O/P: 350      50      135      80**



## PHP -Web Concepts and Reading data from WEB

### Identifying Browser & Platform

PHP creates some useful **environment variables** that can be seen in the **phpinfo.php** page that was used to setup the PHP environment. One of the environment variables set by PHP is **HTTP\_USER\_AGENT** which identifies the user's browser and operating system.

### Using HTML Form with name validation in PHP: test.php

```
<?php
if($_POST["name"] || $_POST["age"])
{
 if (preg_match("/^[A-Za-z'-]$/",$_POST['name']))
 {
 die ("invalid name and name should be alpha"); }
 echo "Welcome ". $_POST['name']. "
";

 echo "You are ". $_POST['age']. " years old.";
 exit();
} ?>
<html>
<body>
<form action="<?php $_PHP_SELF ?>" method="POST">
 Name: <input type="text" name="name" />
 Age: <input type="text" name="age" />
 <input type="submit" />
</form> </body> </html>
```

- The PHP default variable **\$\_PHP\_SELF** is used for the PHP script name and when you click "submit" button then same PHP script will be called and will produce following result
- The method = "POST" is used to post user data to the server script.

### PHP Forms and User Input:

The PHP **\$\_GET** and **\$\_POST** variables are used to retrieve information from forms, like user input.

### PHP - GET & POST Methods

There are two ways the browser client can send information to the web server.

- The GET Method
- The POST Method

Before the browser sends the information, it encodes it using a scheme called **URL encoding or URL Parameters**. In this scheme, name/value pairs are joined with equal signs and different pairs are separated by the ampersand.

### The GET Method

The GET method sends the encoded user information appended to the page request. The page and the encoded information are separated by the **?** character.

- The GET method produces a long string that appears in your server logs, in the **browser's Location: box**.

- The GET method is restricted to send upto **1024 characters only**.
- Never use GET method if you have **password or other sensitive information** to be sent to the server.
- **GET can't be used to send binary data**, like images or word documents, to the server.
- The PHP provides **\$\_GET** associative array to access all the sent information using GET method.

```
<?php
if($_GET["name"] || $_GET["age"])
{
 echo "Welcome ". $_GET['name']. "
";
 echo "You are ". $_GET['age']. " years old.";
 exit();
}
?> <html> <body>
```

```
<form action="<?php $_PHP_SELF ?>" method="GET">
 Name: <input type="text" name="name" />
 Age: <input type="text" name="age" />
 <input type="submit" />
</form> </body> </html>
```

### The POST Method

The POST method transfers information **via HTTP headers or HTTP Parameters**. The information is encoded as described in case of GET method and put into a header called QUERY\_STRING.

- The POST method does not have any restriction on data size to be sent.
- The POST method can be used to send ASCII as well as binary data.
- The data sent by POST method goes through HTTP header so security depends on HTTP protocol. By using Secure HTTP you can make sure that your information is secure.
- The PHP provides **\$\_POST** associative array to access all the sent information using POST method.

### PHP Form Handling

**Example :** The example below contains an HTML form with two input fields and a submit button:

```
<html>
<body>
<form action="welcome.php" method="post">
Name: <input type="text" name="fname" />
Age: <input type="text" name="age" />
<input type="submit" />
</form>
</body>
</html>
```

When a user fills out the form above and click on the submit button, the form data is sent to a PHP file, called "**welcome.php**": its looks like this:

```
<html> <body>
Welcome <?php echo $_POST["fname"]; ?>!
You are <?php echo $_POST["age"]; ?> years old.
</body>
</html> Output: Welcome Kalpana! You are 29 years old.
```

`$_GET` is an array of variables passed to the current script via the **URL parameters**.

`$_POST` is an array of variables passed to the current script via the **HTTP POST method**.

## PHP -File Uploading

A PHP script can be used with a HTML form to allow users to upload files to the server. Initially files are uploaded into a temporary directory and then relocated to a target destination by a PHPscript.

Information in the **phpinfo.php** page describes the temporary directory that is used for file uploads as **upload\_tmp\_dir** and the maximum permitted size of files that can be uploaded is stated as **upload\_max\_filesize**. These parameters are set into PHP configuration file **php.ini**

### The process of uploading a file follows these steps –

The user opens the page containing a HTML form featuring a text files, a browse button and a submit button.

The user clicks the browse button and selects a file to upload from the local PC.

The full path to the selected file appears in the text filed then the user clicks the submit button.

The selected file is sent to the temporary directory on the server.

The PHP script that was specified as the form handler in the form's action attribute checks that the file has arrived and then copies the file into an intended directory.

The PHP script confirms the success to the user.

As usual when writing files it is necessary for both temporary and final locations to have permissions set that enable file writing. If either is set to be read-only then process will fail. An uploaded file could be a text file or image file or any document.

### Creating an upload form

The following HTML code below creates an up loader form. This form is having method attribute set to **post** and **enctype attribute** is set to **multipart/form-data**

There is one global PHP variable called **\$\_FILES**. This variable is an associate double dimension array and keeps all the information related to uploaded file. So if the value assigned to the input's name attribute in uploading form was **file**, then PHP would create following five variables –

**\$\_FILES['file']['tmp\_name']** the uploaded file in the temporary dir on the web server.

**\$\_FILES['file']['name']** – the actual name of the uploaded file.

**\$\_FILES['file']['size']** – the size in bytes of the uploaded file.

**\$\_FILES['file']['type']** – the MIME type of the uploaded file.

**\$\_FILES['file']['error']** – the error code associated with this file upload.

**Example:** Below example should allow upload images and gives back result as uploaded file information.

```
<?php
if(isset($_FILES['image']))
{
 $file_name =
$_FILES['image']['name'];
 $file_size=$_FILES['image']['size'];
 $file_type=$_FILES['image']['type'];
}
?>
<html> <body>
 <form action="" method="POST"
enctype="multipart/form-data">
 <input type="file" name="image" />
 <input type="submit"/>
```

```

 Sent file: <?php echo
$_FILES['image']['name']; ?>
 File size: <?php echo
$_FILES['image']['size']; ?>
 File type: <?php echo
$_FILES['image']['type'] ?>

</form>
</body>
</html>
```

No file chosen

- Sent file: images.jpg
- File size: 8926
- File type: image/jpeg



## PHP -Cookies

Cookies are text files stored on the client computer and they are kept of use tracking purpose. PHP transparently supports HTTP cookies.

There are three steps involved in identifying returning users –

- Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
- Browser stores this information on local machine for future use.
- When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user.

### Setting Cookies with PHP

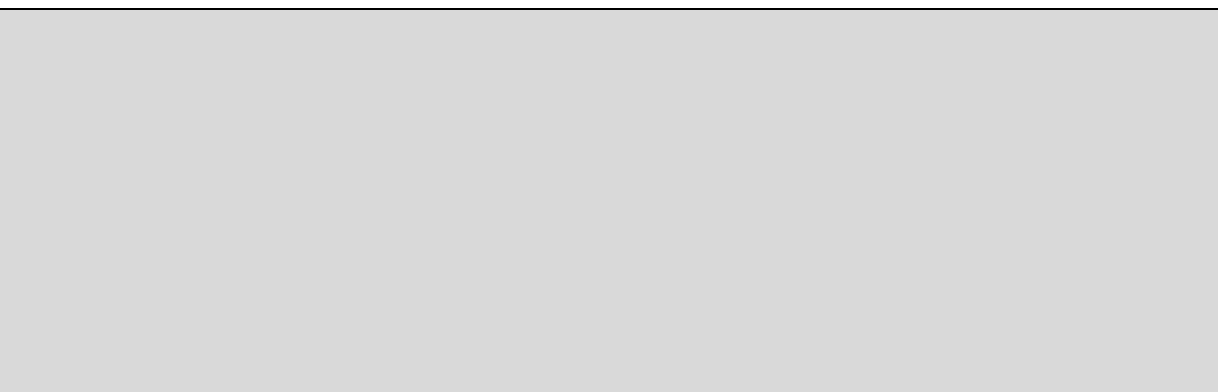
PHP provided **setcookie()** function to set a cookie. This function requires upto six arguments and should be called before <html> tag. For each cookie this function has to be called separately.

```
setcookie(name, value, expire, path, domain, security);
```

Here is the detail of all the arguments –


- **Name** – This sets the name of the cookie and is stored in an environment variable called HTTP\_COOKIE\_VARS. This variable is used while accessing cookies.
- **Value** – This sets the value of the named variable and is the content that you actually want to store.
- **Expiry** – This specify a future time in seconds since 00:00:00 GMT on 1st Jan 1970. After this time cookie will become inaccessible. If this parameter is not set then cookie will automatically expire when the Web Browser is closed.
- **Path** – This specifies the directories for which the cookie is valid. A single forward slash character permits the cookie to be valid for all directories.
- **Domain** – This can be used to specify the domain name in very large domains and must contain at least two periods to be valid. All cookies are only valid for the host and domain which created them.
- **Security** – This can be set to 1 to specify that the cookie should only be sent by secure transmission using HTTPS otherwise set to 0 which mean cookie can be sent by regular HTTP.

Following example will create two cookies **name** and **age** these cookies will be expired after one hour.

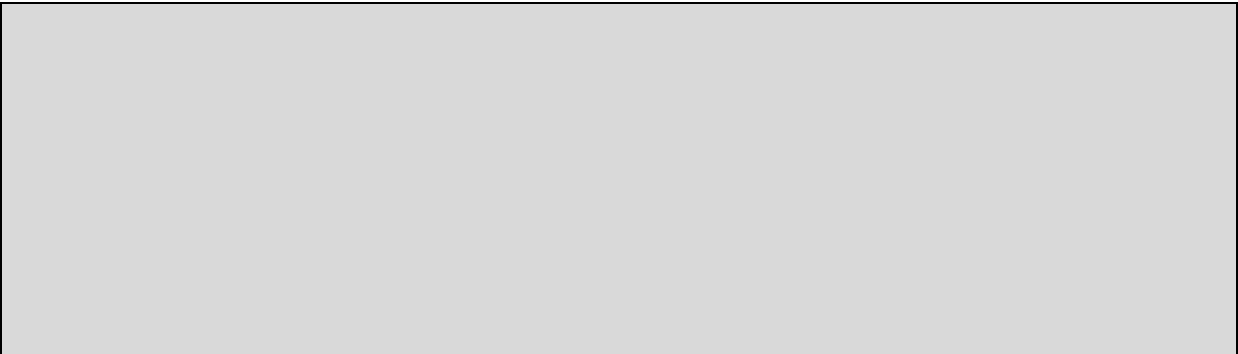


### Accessing Cookies with PHP

PHP provides many ways to access cookies. Simplest way is to use either \$\_COOKIE or HTTP\_COOKIE\_VARS variables. Following example will access all the cookies set in above example.



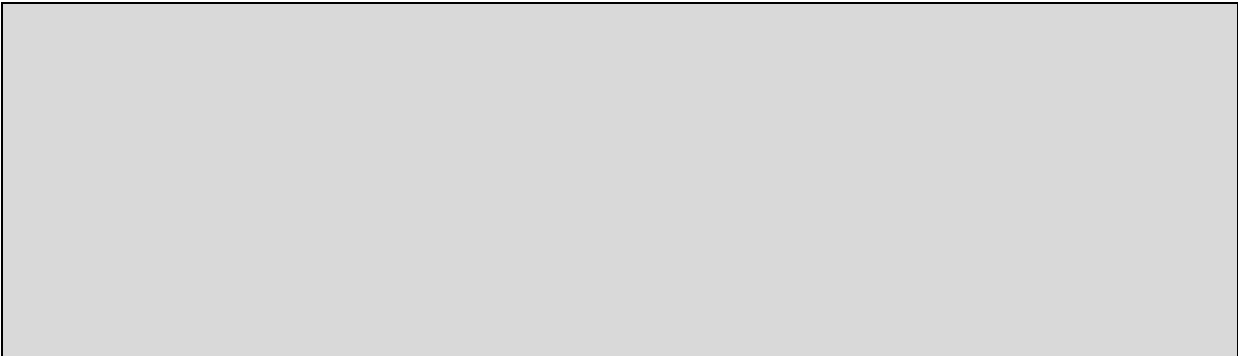
You can use **isset()** function to check if a cookie is set or not.



### Deleting Cookie with PHP

Officially, to delete a cookie you should call `setcookie()` with the name argument only but this does not always work well, however, and should not be relied on.

It is safest to set the cookie with a date that has already expired –



### PHP - Session

- When you work with an application, you open it, do some changes, and then you close it. This is much like a Session.
- Session ID is stored as a cookie on the client box or passed along through URL's.
- The values are actually stored at the server and are accessed via the session id from your cookie. On the client side the session ID expires when connection is broken.
- Session variables solve this problem by storing user information to be used across multiple pages (e.g. username, favorite color, etc). By default, session variables last until the user closes the browser.
- Session variable values are stored in the 'superglobal' associative array `$_SESSION`

### Start a PHP Session

A session is started with the `session_start()` function.

Session variables are set with the PHP global variable: `$_SESSION`.

<b>demo_session1.php</b> <pre>&lt;?php // Start the session session_start(); ?&gt; &lt;html&gt; &lt;body&gt; &lt;?php // Set session variables \$_SESSION["favcolor"] = "green"; \$_SESSION["favanimal"] = "cat"; echo "Session variables are set."; ?&gt; &lt;/body&gt; &lt;/html&gt;</pre>	<b>Modify a PHP Session Variable</b> To change a session variable, just overwrite it:  <pre>&lt;?php session_start(); ?&gt; &lt;html&gt; &lt;?php \$_SESSION["favcolor"] = "yellow"; print_r(\$_SESSION); ?&gt; &lt;/html&gt;</pre>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

### Get PHP Session Variable Values

Next, we create another page called "**demo\_session2.php**". From this page, we will access the session information we set on the first page ("**demo\_session1.php**").

Notice that session variables are not passed individually to each new page, instead they are retrieved from the session we open at the beginning of each page (`session_start()`).

```
<?php
session_start();
?>
<!DOCTYPE html>
<html>
<body>
<?php
// Echo session variables that were set on previous page
echo "Favorite color is " . $_SESSION["favcolor"] . "
";
echo "Favorite animal is " . $_SESSION["favanimal"] . ".";
?>
</body> </html>
```

### Destroy a PHP Session

To remove all global session variables and destroy the session, use `session_unset()` and `session_destroy()`:

#### Example

```
<?php
session_start();
?>
<html>
<body>
<?php
session_unset(); // remove all session variables
session_destroy(); // destroy the session
?> </body> </html>
```

### Difference between a session and a cookie

- The main **difference** between a **session** and a **cookie** is that **session** data is stored on the server, whereas **cookies** store data in the visitor's browser.
- **Sessions** are more secure than **cookies** as it is stored in server. **Cookie** can be turn off from browser.

Sessions	Cookies
1 Sessions are <b>server-side</b> files that contain user information	Cookies are <b>client-side</b> files that contain user information
2 Session Max life time is 1440 Seconds(24 Minutes) as defined in php.ini file in <b>php.ini</b> on line 1604 you can find ; http://php.net/session.gc- maxlifetime session.gc_maxlifetime = 1440 You can edit this value if you need custom session life.	We have to set cookie max life time manually with php code with setcookie function. setcookie("email", 'test@example.com', time()+3600); /* expire in 1 hour */ Expire time : I hour after current time
3 In php <b>\$_SESSION</b> super global variable is used to manage session.	In php <b>\$_COOKIE</b> super global variable is used to manage cookie.
4 Before using <b>\$_SESSION</b> , you have to write session_start(); In that way session will start	You don't need to start Cookie as It is stored in your local machine.
5 You can store as much data as you like within in sessions.(default is 128MB.) memory_limit= 128M php.ini line 479 ;http://php.net/memory- limit	Official MAX Cookie size is 4KB
6 Session is dependent on COOKIE. Because when you start session with session_start() then SESSIONID named key will be set in COOKIE with Unique Identifier Value for your system.	
7 <b>session_destroy()</b> ; is used to " <b>Destroys all data registered to a session</b> ", and if you want to unset some key's of SESSION then use unset() function. unset(\$_SESSION["key1"], \$_SESSION["key2"])	There is no function named unsetcookie() time()-3600); //expire before 1hour In that way you unset cookie(Set cookie in previous time)
8 Session ends when user closes his browser.	Cookie ends depends on the life time you set for it.
9 A <b>session</b> is a group of information on the server that is associated with the cookie information.	<b>Cookies</b> are used to identify sessions.

**Write a Program to create simple Login and Logout example using sessions.**

#### login.php

```
<html>
<head>
<title>Login Form</title>
</head>
<body>
<h2>Login Form</h2>
<form method="post" action="checklogin.php">
User Id: <input type="text" name="uid">

```



```
Password: <input type="password" name="pw">

<input type="submit" value="Login">
</form>
</body>
</html>
```

**checklogin.php**

```
<?php
$uid = $_POST['uid'];
$pw = $_POST['pw'];
if($uid == 'arun' and $pw == 'arun123')
{
 session_start();
 $_SESSION['sid']=session_id();
 header("location:securepage.php");
}
?>
```

**securepage.php**

```
<?php
 session_start();
 if($_SESSION['sid']==session_id())
 {
echo "Welcome to you
";
echo "Logout";
 }
 else
 {
 header("location:login.php");
 }
?>
```

**logout.php**

```
<?php
 echo "Logged out successfully";
 session_start();
 session_destroy();
 setcookie(PHPSESSID,session_id(),time()-1);
?>
```

**XML -**

stands for **Extensible Mark-up Language**, developed by W3C in 1996. It is a text-based mark-up language derived from Standard Generalized Mark-up Language (SGML). XML 1.0 was officially adopted as a W3C recommendation in 1998. XML was designed to carry data, not to display data. XML is designed to be self-descriptive. XML is a subset of SGML that can define your own tags. A Meta Language and tags describe the content. XML Supports CSS, XSL, DOM. XML does not qualify to be a programming language as it does not performs any computation or algorithms. It is usually stored in a simple text file and is processed by special software that is capable of interpreting XML.

**The Difference between XML and HTML**

1. HTML is about displaying information, where asXML is about carrying information. In other words, XML was created to structure, store, and transport information. HTML was designed to display the data.
2. Using XML, we can create own tags where as in HTML it is not possible instead it offers several built in tags.
3. XML is platform independent neutral and language independent.
4. XML tags and attribute names are case-sensitive where as in HTML it is not.
5. XML attribute values must be single or double quoted where as in HTML it is not compulsory.
6. XML elements must be properly nested.
7. All XML elements must have a closing tag.

**Well Formed XML Documents**

**A "Well Formed" XML document must have the following correct XML syntax:**

- XML documents must have a root element
- XML elements must have a closing tag(start tag must have matching end tag).
- XML tags are case sensitive
- XML elements must be properly nested Ex:<one><two>Hello</two></one>
- XML attribute values must be quoted

XML with correct syntax is "Well Formed" XML. XML validated against a DTD is "Valid" XML.

**What is Markup?**

XML is a markup language that defines set of rules for encoding documents in a format that is both human-readable and machine-readable.

**Example for XML Document**

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?> <!--xml declaration-->
<note>
<to>MRCET</to>
<from>MRGI</from>
<heading>KALPANA</heading>
<body>Hello, world! </body>
</note>
```

- Xml document begins with XML declaration statement: <? xml version="1.0" encoding="ISO-8859-1"?> .

- The next line describes the **root element** of the document: `<note>`.
- This element is "the parent" of all other elements.
- The next 4 lines describe 4**child elements** of the root: to, from, heading, and body. And finally the last line defines the end of the root element : `</note>`.
- The XML declaration has no closing tag i.e. `<?xml>`
- The **default standalone value** is set to **no**. Setting it to **yes** tells the processor there are no external declarations (DTD) required for parsing the document. The file name extension used for xml program is.xml.

### Valid XML document

If an XML document is well- formed and has an associated Document Type Declaration (DTD), then it is said to be a valid XML document. We will study more about DTD in the chapter XML - DTDs.

### XML DTD

Document Type Definition purpose is to define the structure of an XML document. It defines the structure with a list of defined elements in the xml document. Using DTD we can specify the various elements types, attributes and their relationship with one another. Basically DTD is used to specify the set of rules for structuring data in any XML file.

### Why use a DTD?

XML provides an application independent way of sharing data. With a DTD, independent groups of people can agree to use a common DTD for interchanging data. Your application can use a standard DTD to verify that data that you receive from the outside world is valid. You can also use a DTD to verify your own data.

### DTD - XML building blocks

Various building blocks of XML are-

**1. Elements:** The basic entity is **element**. The elements are used for defining the tags. The elements typically consist of opening and closing tag. Mostly only one element is used to define a single tag.

**Syntax1:** `<!ELEMENT element- name (element-content)>`

**Syntax 2:** `<!ELEMENT element- name (#CDATA)>`

#CDATA means the element contains character data that is not supposed to be parsed by a parser. or

**Syntax 3:** `<!ELEMENT element- name (#PCDATA)>`

#PCDATA means that the element contains data that IS going to be parsed by a parser. or

**Syntax 4:** `<!ELEMENT element- name (ANY)>`

The keyword ANY declares an element with any content.

### Example:

`<!ELEMENT note (#PCDATA)>`

### Elements with children (sequences)

Elements with one or more children are defined with the name of the children elements inside the parentheses:

```
<!ELEMENT parent-name (child-element- name)> EX:<!ELEMENT student (id)>
 <!ELEMENT id (#PCDATA)> or
<!ELEMENT element-name (child-element-name, child-element-name,)>
```

**Example:** `<!ELEMENT note (to,from,heading,body)>`

When children are declared in a sequence separated by commas, the children must appear in the same sequence in the document. In a full declaration, the children must also be declared, and the children can also have children. The full declaration of the note document will be:

```
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#CDATA)>
<!ELEMENT from (#CDATA)>
<!ELEMENT heading (#CDATA)>
<!ELEMENT body (#CDATA)>
```

## 2. Tags

Tags are used to markup elements. A starting tag like `<element_name>` mark up the beginning of an element, and an ending tag like `</element_name>` mark up the end of an element.

### Examples:

A body element: `<body>body text in between</body>`.

A message element: `<message>some message in between</message>`

**3. Attribute:** The attributes are generally used to specify the values of the element. These are specified within the double quotes. Ex: `<flag type="true">`

## 4. Entities

Entities as variables used to define common text. Entity references are references to entities. Most of you will know the HTML entity reference: "&nbsp;" that is used to insert an extra space in an HTML document. Entities are expanded when a document is parsed by an XML parser.

### The following entities are predefined in XML:

&lt; (<), &gt; (>), &amp; (&), &quot; (") and &apos; (').

**5. CDATA:** It stands for character data. CDATA is text that will **NOT be parsed by a parser**. Tags inside the text will NOT be treated as markup and entities will not be expanded.

**6. PCDATA:** It stands for Parsed Character Data(i.e., text). Any parsed character data should not contain the markup characters. The markup characters are `<` or `>` or `&`. If we want to use these characters then make use of `&lt;`, `&gt;` or `&amp;`. Think of character data as the text found between the start tag and the end tag of an XML element. PCDATA is text that will be **parsed by a parser**. Tags inside the text will be treated as markup and entities will be expanded.

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

Where PCDATA refers parsed character data. In the above xml document the elements to, from, heading, body carries some text, so that, these elements are declared to carry text in DTD file.

This definition file is stored with **.dtd** extension.

DTD identifier is an identifier for the document type definition, which may be the path to a file on the system or URL to a file on the internet. If the DTD is pointing to external path, it is called External Subset.

The square brackets [ ] enclose an optional list of entity declarations called Internal Subset.

**Types of DTD:**

1. Internal DTD
2. External DTD

**1. Internal DTD**

A DTD is referred to as an internal DTD if elements are declared within the XML files. To refer it as internal DTD, standalone attribute in XML declaration must be set to yes. This means, the declaration works independent of external source.

**Syntax:**

The syntax of internal DTD is as shown:

```
<!DOCTYPE root-element [element-declarations]>
```

Where root-element is the name of root element and element-declarations is where you declare the elements.

**Example:**

Following is a simple example of internal DTD:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE address [
 <!ELEMENT address (name,company,phone)>
 <!ELEMENT name (#PCDATA)>
 <!ELEMENT company (#PCDATA)>
 <!ELEMENT phone (#PCDATA)>
]>
<address>
 <name>Kalpana</name>
 <company>MRCET</company>
 <phone>(040) 123-4567</phone>
</address>
```

**Let us go through the above code:**

Start Declaration- Begin the XML declaration with following statement `<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>`

DTD- Immediately after the XML header, the document type declaration follows, commonly referred to as the DOCTYPE:

```
<!DOCTYPE address [
```

The DOCTYPE declaration has an exclamation mark (!) at the start of the element name. The DOCTYPE informs the parser that a DTD is associated with this XML document.

**DTD Body-** The DOCTYPE declaration is followed by body of the DTD, where you declare elements, attributes, entities, and notations:

```
<!ELEMENT address (name,company,phone)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT company (#PCDATA)>
<!ELEMENT phone_no (#PCDATA)>
```

Several elements are declared here that make up the vocabulary of the `<name>` document.

`<!ELEMENT name (#PCDATA)>` defines the element name to be of type `"#PCDATA"`. Here `#PCDATA` means parse-able text data. End Declaration - Finally, the declaration section of the DTD is closed using a closing bracket and a closing angle bracket (`]>`). This effectively ends the definition, and thereafter, the XML document follows immediately.

**Rules**

- ✓ The document type declaration must appear at the start of the document (preceded only by the XML header) — it is not permitted anywhere else within the document.
- ✓ Similar to the DOCTYPE declaration, the element declarations must start with an exclamation mark.
- ✓ The Name in the document type declaration must match the element type of the root element.

**External DTD**

In external DTD elements are declared outside the XML file. They are accessed by specifying the system attributes which may be either the legal .dtd file or a valid URL. To refer it as external DTD, standalone attribute in the XML declaration must be set as no. This means, declaration includes information from the external source.

**Syntax** Following is the syntax for external DTD:

```
<!DOCTYPE root-element SYSTEM "file-name">
```

where file- name is the file with **.dtd** extension.

**Example** The following example shows external DTD usage:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE address SYSTEM "address.dtd">
<address>
 <name>Kalpana</name>
 <company>MRCET</company>
 <phone>(040) 123-4567</phone>
</address>
```

The content of the DTD file **address.dtd** are as shown:

```
<!ELEMENT address (name,company,phone)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT company (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
```

**Types**

You can refer to an external DTD by using either system identifiers or public identifiers.

**SYSTEM IDENTIFIERS**

A system identifier enables you to specify the location of an external file containing DTD declarations. Syntax is as follows:

```
<!DOCTYPE name SYSTEM "address.dtd" [...]>
```

As you can see, it contains keyword SYSTEM and a URI reference pointing to the location of the document.

**PUBLIC IDENTIFIERS**

Public identifiers provide a mechanism to locate DTD resources and are written as below:

```
<!DOCTYPE name PUBLIC "-//Beginning XML//DTD Address Example//EN">
```

As you can see, it begins with keyword PUBLIC, followed by a specialized identifier. Public identifiers are used to identify an entry in a catalog. Public identifiers can follow any format; however, a commonly used format is called Formal Public Identifiers, or FPIs.

## XML Schemas

- XML Schema is commonly known as XML Schema Definition (XSD). It is used to describe and validate the structure and the content of XML data. XML schema defines the elements, attributes and data types. Schema element supports Namespaces. It is similar to a database schema that describes the data in a database. XSD extension is **“.xsd”**.
- This can be used as an alternative to XML DTD. The XML schema became the W3C recommendation in 2001.
- XML schema defines elements, attributes, element having child elements, order of child elements. It also defines fixed and default values of elements and attributes.
- XML schema also allows the developer to use **data types**.

**Syntax :** You need to declare a schema in your XML document as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
```

### Example : contact.xsd

The following example shows how to use schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="contact">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="name" type="xs:string" />
 <xs:element name="company" type="xs:string" />
 <xs:element name="phone" type="xs:int" />
 </xs:sequence>
 </xs:complexType>
</xs:element>
</xs:schema>
```

The basic idea behind XML Schemas is that they describe the legitimate format that an XML document can take.

### XML Document: myschema.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<contact xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="contact.xsd">
<name>KALPANA</name>
<company>04024056789</company>
<phone>9876543210</phone>
</contact>
```

### Limitations of DTD:

- There is no built-in data type in DTDs.
- No new data type can be created in DTDs.
- The use of cardinality (no. of occurrences) in DTDs is limited.
- Namespaces are not supported.
- DTDs provide very limited support for modularity and reuse.
- We cannot put any restrictions on text content.
- Defaults for elements cannot be specified.
- DTDs are written in a non-XML format and are difficult to validate.

**Strengths of Schema:**

- XML schemas provide much greater specificity than DTDs.
- They supports large number of built- in-data types.
- They are namespace-aware.
- They are extensible to future additions.
- They support the uniqueness.
- It is easier to define data facets (restrictions on data).

**SCHEMA STRUCTURE****The Schema Element**

```
<xs: schema xmlns: xs="http://www.w3.org/2001/XMLSchema">
```

**Element definitions**

As we saw in the chapter XML - Elements, elements are the building blocks of XML document. An element can be defined within an XSD as follows:

```
<xs:element name="x" type="y"/>
```

**Data types:**

These can be used to specify the type of data stored in an Element.

- String (xs:string)
- Date (xs:date or xs:time)
- Numeric (xs:integer or xs:decimal)
- Boolean (xs:boolean)

**EX: Sample.xsd**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/XMLSchema">
 <xs:element name="sname" type="xs:string"/>
 /* <xs:element name="dob" type="xs:date"/>
 <xs:element name="dobtime" type="xs:time"/>
 <xs:element name="marks" type="xs:integer"/>
 <xs:element name="avg" type="xs:decimal"/>
 <xs:element name="flag" type="xs:boolean"/> */
</xs:schema>
```

**Sample.xml:**

```
<?xml version="1.0" encoding="UTF-8"?>
<sname xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="sample.xsd">
 Kalpana /*yyyy-mm-dd 23:14:34 600 92.5 true/false */
</sname>
```

**Definition Types**

You can define XML schema elements in following ways:

**Simple Type** - Simple type element is used only in the context of the text. Some of predefined simple types are: xs:integer, xs:boolean, xs:string, xs:date. For example:

```
<xs:element name="phone_number" type="xs:int" />
<phone>9876543210</phone>
```



**Default and Fixed Values for Simple Elements**

In the following example the default value is "red":

```
<xs:element name="color" type="xs:string" default="red"/>
```

In the following example the fixed value is "red":

```
<xs:element name="color" type="xs:string" fixed="red"/>
```

**Complex Type** - A complex type is a container for other element definitions. This allows you to specify which child elements an element can contain and to provide some structure within your XML documents. For example:

```
<xs:element name="Address">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="name" type="xs:string" />
 <xs:element name="company" type="xs:string" />
 <xs:element name="phone" type="xs:int" />
 </xs:sequence>
 </xs:complexType>
</xs:element>
```

In the above example, Address element consists of child elements. This is a container for other `<xs:element>` definitions, that allows to build a simple hierarchy of elements in the XML document.

**Global Types** - With global type, you can define a single type in your document, which can be used by **all other references**. For example, suppose you want to generalize the person and company for different addresses of the company. In such case, you can define a general type as below:

```
<xs:element name="AddressType">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="name" type="xs:string" />
 <xs:element name="company" type="xs:string" />
 </xs:sequence>
 </xs:complexType>
</xs:element>
```

Now let us use this type in our example as below:

```
<xs:element name="Address1">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="address" type="AddressType" />
 <xs:element name="phone1" type="xs:int" />
 </xs:sequence>
 </xs:complexType>
</xs:element>
<xs:element name="Address2">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="address" type="AddressType" />
 <xs:element name="phone2" type="xs:int" />
 </xs:sequence>
 </xs:complexType>
</xs:element>
```

Instead of having to define the name and the company twice (once for Address1 and once for Address2), we now have a single definition. This makes maintenance simpler, i.e., if you decide to add "Postcode" elements to the address, you need to add them at just one place.

### Attributes

Simple elements cannot have attributes. If an element has attributes, it is considered to be of a complex type. But the attribute itself is always declared as a simple type. Attributes in XSD provide extra information within an element. Attributes have name and type property as shown below:

```
<xs:attribute name="x" type="y"/>
```

**Ex:** <lastname lang="EN">Smith</lastname>

```
<xs:attribute name="lang" type="xs:string"/>
```

### Default and Fixed Values for Attributes

```
<xs:attribute name="lang" type="xs:string" default="EN"/>
```

```
<xs:attribute name="lang" type="xs:string" fixed="EN"/>
```

### Optional and Required Attributes

Attributes are optional by default. To specify that the attribute is required, use the "use" attribute:

```
<xs:attribute name="lang" type="xs:string" use="required"/>
```

### Restrictions on Content

When an XML element or attribute has a data type defined, it puts restrictions on the element's or attribute's content. If an XML element is of type "xs:date" and contains a string like "Hello World", the element will not validate.

### Restrictions on Values:

The value of **age** cannot be lower than 0 or greater than 120:

```
<xs:element name="age">
```

```
<xs:simpleType>
```

```
<xs:restriction base="xs:integer">
```

```
<xs:minInclusive value="0"/>
```

```
<xs:maxInclusive value="120"/>
```

```
</xs:restriction>
```

```
</xs:simpleType> </xs:element>
```

### Restrictions on a Set of Values

The example below defines an element called "car" with a restriction. The only acceptable values are: Audi, Golf, BMW:

```
<xs:element name="car">
```

```
<xs:simpleType>
```

```
<xs:restriction base="xs:string">
```

```
<xs:enumeration value="Audi"/>
```

```
<xs:enumeration value="Golf"/>
```

```
<xs:enumeration value="BMW"/>
```

```
</xs:restriction>
```

```
</xs:simpleType>
```

```
</xs:element>
```

**Restrictions on Length**

To limit the length of a value in an element, we would use the length, maxLength, and minLength constraints. **The value must be exactly eight characters:**

```
<xs:element name="password">
 <xs:simpleType>
 <xs:restriction base="xs:string">
 <xs:length value="8"/> [<xs:minLength value="5"/> <xs:maxLength value="8"/>]
 </xs:restriction> </xs:simpleType> </xs:element>
```

**XSD Indicators**

We can control HOW elements are to be used in documents with indicators.

**Indicators:** There are seven indicators

**Order indicators:**

- All
- Choice
- Sequence

**Occurrence indicators:**

- maxOccurs
- minOccurs

**Group indicators:**

- Group name
- attributeGroup name

**→Order Indicators**

Order indicators are used to define the order of the elements.

**All Indicator**

The <all> indicator specifies that the child elements can appear in any order, and that each child element must occur only once:

```
<xs:element name="person">
 <xs:complexType>
 <xs:all>
 <xs:element name="firstname" type="xs:string"/>
 <xs:element name="lastname" type="xs:string"/>
 </xs:all>
 </xs:complexType>
</xs:element>
```

**Note:** When using the <all> indicator you can set the <minOccurs> indicator to 0 or 1 and the <maxOccurs> indicator can only be set to 1 (the <minOccurs> and <maxOccurs> are described later).

**Choice Indicator**

The <choice> indicator specifies that either one child element or another can occur:

```
<xs:element name="person">
 <xs:complexType>
 <xs:choice>
 <xs:element name="employee" type="employee"/>
 <xs:element name="member" type="member"/>
 </xs:choice> </xs:complexType> </xs:element>
```

## Sequence Indicator

The <sequence> indicator specifies that the child elements must appear in a specific order:

```
<xs:element name="person">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="firstname" type="xs:string"/>
 <xs:element name="lastname" type="xs:string"/>
 </xs:sequence>
 </xs:complexType>
</xs:element>
```

## → Occurrence Indicators

Occurrence indicators are used to define how often an element can occur.

**Note:** For all "Order" and "Group" indicators (any, all, choice, sequence, group name, and group reference) **the default value for maxOccurs and minOccurs is 1.**

### maxOccurs Indicator

The <maxOccurs> indicator specifies the maximum number of times an element can occur:

```
<xs:element name="person">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="full_name" type="xs:string"/>
 <xs:element name="child_name" type="xs:string" maxOccurs="10"/>
 </xs:sequence>
 </xs:complexType>
</xs:element>
```

### minOccurs Indicator

The <minOccurs> indicator specifies the minimum number of times an element can occur:

```
<xs:element name="person">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="full_name" type="xs:string"/>
 <xs:element name="child_name" type="xs:string" maxOccurs="10" minOccurs="0"/>
 </xs:sequence>
 </xs:complexType>
</xs:element>
```

**Tip:** To allow an element to appear an unlimited number of times, use the

**maxOccurs="unbounded"** statement:

**EX:** An XML file called "Myfamily.xml":

```
<?xml version="1.0" encoding="UTF-8"?>
<persons xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="family.xsd">
 <person>
 <full_name>KALPANA</full_name>
 <child_name>mrcet</child_name>
 </person>
 <person>
 <full_name>Tove Refsnes</full_name>
 <child_name>Hege</child_name>
 <child_name>Stale</child_name>
```

```
<child_name>Jim</child_name>
<child_name>Borge</child_name>
</person>
<person>
 <full_name>Stale Refsnes</full_name>
</person>
</persons>
```

The XML file above contains a root element named "persons". Inside this root element we have defined three "person" elements. Each "person" element must contain a "full\_name" element and it can contain up to five "child\_name" elements.

Here is the schema file "**family.xsd**":

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema
 elementFormDefault="qualified">
 <xs:element name="persons">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="person" maxOccurs="unbounded">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="full_name" type="xs:string"/>
 <xs:element name="child_name" type="xs:string" minOccurs="0" maxOccurs="5"/>
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 </xs:sequence>
 </xs:complexType>
 </xs:element>
</xs:schema>
```

→ **Group Indicators:** Group indicators are used to define related sets of elements.

### Element Groups

Element groups are defined with the group declaration, like this:

```
<xs:group name="groupname">
...
</xs:group>
```

You must define an all, choice, or sequence element inside the group declaration. The following example defines a group named "persongroup", that defines a group of elements that must occur in an exact sequence:

```
<xs:group name="persongroup">
 <xs:sequence>
 <xs:element name="firstname" type="xs:string"/>
 <xs:element name="lastname" type="xs:string"/>
 <xs:element name="birthday" type="xs:date"/>
 </xs:sequence>
</xs:group>
```

After you have defined a group, **you can reference it in another definition**, like this:

```
<xs:element name="person" type="personinfo"/>
<xs:complexType name="personinfo">
 <xs:sequence>
 <xs:group ref="persongroup"/>
 <xs:element name="country" type="xs:string"/>
 </xs:sequence>
</xs:complexType>
```

### Attribute Groups

Attribute groups are defined with the attributeGroup declaration, like this:

```
<xs:attributeGroup name="groupname">
...
</xs:attributeGroup>
```

The following example defines an attribute group named "personattrgroup":

```
<xs:attributeGroup name="personattrgroup">
 <xs:attribute name="firstname" type="xs:string"/>
 <xs:attribute name="lastname" type="xs:string"/>
 <xs:attribute name="birthday" type="xs:date"/>
</xs:attributeGroup>
```

After you have defined an attribute group, you can reference it in another definition, like this:

```
<xs:element name="person">
 <xs:complexType>
 <xs:attributeGroup ref="personattrgroup"/> </xs:complexType> </xs:element>
```

### Example Program: "shiporder.xml"

```
<?xml version="1.0" encoding="UTF-8"?>
<shiporder orderid="889923"
 xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
 xsi:noNamespaceSchemaLocation="shiporder.xsd">
 <orderperson>John Smith</orderperson>
 <shipto>
 <name>Ola Nordmann</name>
 <address>Langgt 23</address>
 <city>4000 Stavanger</city>
 <country>Norway</country>
 </shipto>
 <item>
 <title>Empire Burlesque</title>
 <note>Special Edition</note>
 <quantity>1</quantity>
 <price>10.90</price>
 </item>
 <item>
 <title>Hide your heart</title> <quantity>1</quantity>
 <price>9.90</price> </item>
</shiporder>
```

**Create an XML Schema "shiporder.xsd":**

```

<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="shiporder">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="orderperson" type="xs:string"/>
 <xs:element name="shiporder">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="name" type="xs:string"/>
 <xs:element name="address" type="xs:string"/>
 <xs:element name="city" type="xs:string"/>
 <xs:element name="country" type="xs:string"/>
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 <xs:element name="item" maxOccurs="unbounded">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="title" type="xs:string"/>
 <xs:element name="note" type="xs:string" minOccurs="0"/>
 <xs:element name="quantity" type="xs:positiveInteger"/>
 <xs:element name="price" type="xs:decimal"/>
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 </xs:sequence>
 <xs:attribute name="orderid" type="xs:string" use="required"/>
 </xs:complexType>
</xs:element>
</xs:schema>

```

**XML DTD vs XML Schema**

The schema has more advantages over DTD. A DTD can have two types of data in it, namely the CDATA and the PCDATA. The CDATA is not parsed by the parser whereas the PCDATA is parsed. In a schema you can have primitive data types and custom data types like you have used in programming.

**Schema vs. DTD**

- XML Schemas are extensible to future additions
- XML Schemas are richer and more powerful than DTDs
- XML Schemas are written in XML
- XML Schemas support data types
- XML Schemas support namespaces

**XML Parsers**

An XML parser converts an XML document into an XML DOM object - which can then be manipulated with a JavaScript.

**Two types of XML parsers:****➤ Validating Parser**

- It requires document type declaration
- It generates error if document does not
  - Conform with DTD and
  - Meet XML validity constraints

**➤ Non-validating Parser**

- It checks well- formedness for xml document
- It can ignore external DTD

**What is XML Parser?**

XML Parser provides way how to access or modify data present in an XML document. Java provides multiple options to parse XML document. Following are various types of parsers which are commonly used to parse XML documents.

**Types of parsers:**

- **Dom Parser** - Parses the document by loading the complete contents of the document and creating its complete hierarchical tree in memory.
- **SAX Parser** - Parses the document on event based triggers. Does not load the complete document into the memory.
- **JDOM Parser** - Parses the document in similar fashion to DOM parser but in more easier way.
- **StAX Parser** - Parses the document in similar fashion to SAX parser but in more efficient way.
- **XPath Parser** - Parses the XML based on expression and is used extensively in conjunction with XSLT.
- **DOM4J Parser** - A java library to parse XML, XPath and XSLT using Java Collections Framework , provides support for DOM, SAX and JAXP.

**DOM-Document Object Model**

The Document Object Model protocol converts an XML document into a collection of objects in your program. XML documents have a hierarchy of informational units called nodes; this hierarchy allows a developer to navigate through the tree looking for specific information. Because it is based on a hierarchy of information, the DOM is said to be tree based. DOM is a way of describing those nodes and the relationships between them.

You can then manipulate the object model in any way that makes sense. This mechanism is also known as the "random access" protocol, because you can visit any part of the data at any time. You can then modify the data, remove it, or insert new data.

The XML DOM, on the other hand, also provides an API that allows a developer to add, edit, move, or remove nodes in the tree at any point in order to create an application. A DOM parser creates a tree structure in memory from the input document and then waits for requests from client. A DOM parser always serves the client application with the **entire document no matter how much is actually needed** by the client. With DOM parser, method calls in client application have to be explicit and forms a kind of chained method calls.

Document Object Model is for defining the standard for accessing and manipulating XML documents. **XML DOM** is used for



- Loading the xml document
- Accessing the xml document
- Deleting the elements of xml document
- Changing the elements of xml document

According to the DOM, everything in an XML document is a node. It considers

- The entire document is a document node
- Every XML element is an element node
- The text in the XML elements are text nodes
- Every attribute is an attribute node
- Comments are comment nodes

**The W3C DOM specification is divided into three major parts:**

**DOM Core-** This portion defines the basic set of interfaces and objects for any structured documents.

**XML DOM-** This part specifies the standard set of objects and interfaces for XML documents only.

**HTML DOM-** This part specifies the objects and interfaces for HTML documents only.

#### DOM Levels

- Level 1 Core: W3C Recommendation, October 1998
  - ✓ It has feature for primitive navigation and manipulation of XML trees
  - ✓ other Level 1 features are: All HTML features
- Level 2 Core: W3C Recommendation, November 2000
  - ✓ It adds Namespace support and minor new features
  - ✓ other Level 2 features are: Events, Views, Style, Traversal and Range
- Level 3 Core: W3C Working Draft, April 2002
  - ✓ It supports: Schemas, XPath, XSL, XSLT

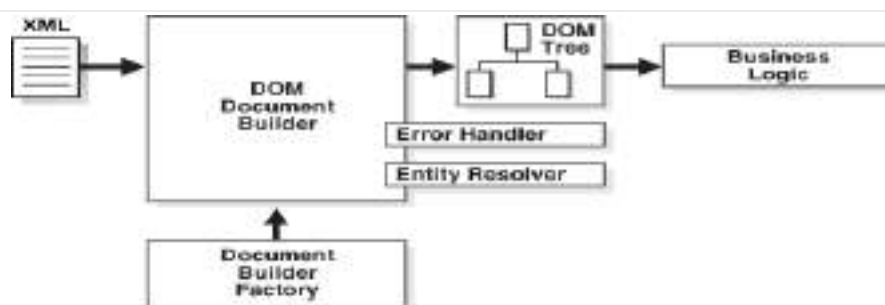
We can access and parse the XML document in two ways:

- Parsing using DOM (tree based)
- Parsing using SAX (Event based)

Parsing the XML doc. using DOM methods and properties are called as **tree based approach** whereas using SAX (Simple Api for Xml) methods and properties are called as **event based approach**.

#### Steps to Using DOM Parser

Let's note down some broad steps involved in using a DOM parser for parsing any XML file in java.



**DOM based XML Parsing:(tree based)**

JAXP is a tool, stands for Java Api for Xml Processing, used for accessing and manipulating xml document in a tree based manner.

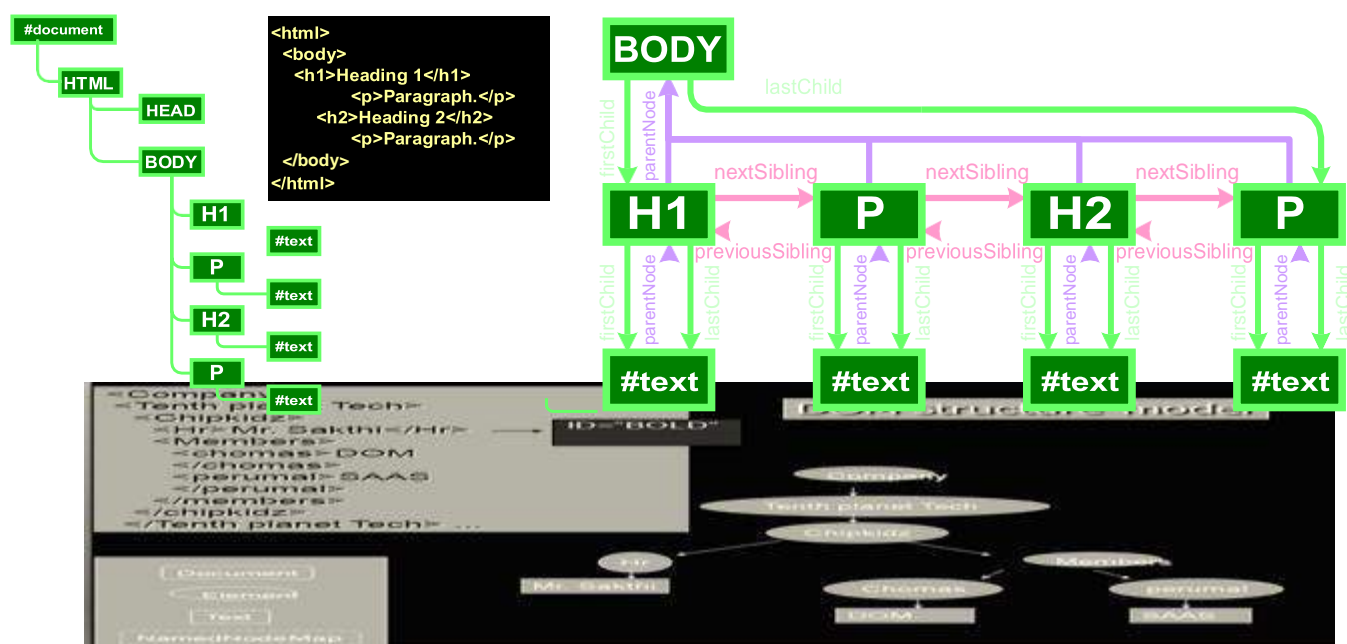
The following DOM javaClasses are necessary to process the XML document:

- DocumentBuilderFactory class creates the instance of DocumentBuilder.
- DocumentBuilder produces a Document (a DOM) that conforms to the DOM specification.

The following methods and properties are necessary to process the XML document:

Property	Meaning
nodeName	Finding the name of the node
nodeValue	Obtaining value of the node
parentNode	To get parnet node
childNodes	Obtain child nodes
Attributes	For getting the attributes values

Method	Meaning
getElementByTagName(name)	To access the element by specifying its name
appendChild(node)	To insert a child node
removeChild(node)	To remove existing child node

**DOM Document Object**

- ✓ There are 12 types of nodes in a DOM Document object

1. Document node	7. EntityReference node
2. Element node	8. Entity node
3. Text node	9. Comment node
4. Attribute node	10. DocumentType node
5. Processing instruction node	11. DocumentFragment node
6. CDATA Section node	12. Notation node

**Examples for Document method**

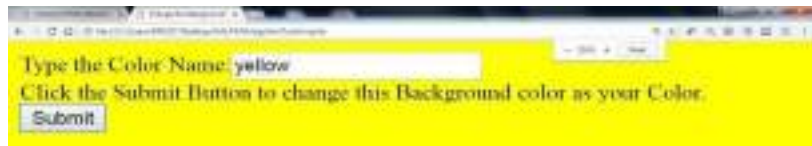
```

<html>
 <head>
 <title>Change the Background</title>
 </head>
 <body>
 <script language = "JavaScript">
 function background()
 {
 var color = document.bg.color.value;
 document.body.style.backgroundColor=color;
 }
 </script>
 <form name="bg">
 Type the Color Name:<input type="text" name="color" size="20">

 Click the Submit Button to change this Background color as your Color.

 <input type="button" value="Submit" onClick='background()>
 </form>
 </body>
</html>

```

**DOM NODE Methods**

Method Name	Description
<b>appendChild</b>	Appends a child node.
<b>cloneNode</b>	Duplicates the node.
<b>getAttributes</b>	Returns the node's attributes.
<b>getChildNodes</b>	Returns the node's child nodes.
<b>getNodeName</b>	Returns the node's name.
<b>getNodeType</b>	Returns the node's type (e.g., element, attribute, text, etc.).
<b>getNodeValue</b>	Returns the node's value.
<b>getParentNode</b>	Returns the node's parent.
<b>hasChildNodes</b>	Returns <b>true</b> if the node has child nodes.
<b>removeChild</b>	Removes a child node from the node.
<b>replaceChild</b>	Replaces a child node with another node.
<b>setNodeValue</b>	Sets the node's value.
<b>insertBefore</b>	Appends a child node in front of a child node.

**DOM Advantages & Disadvantages****ADVANTAGES**

- Robust API for the DOM tree
- Relatively simple to modify the data structure and extract data
- It is good when random access to widely separated parts of a document is required
- It supports both read and write operations
- 

**Disadvantages**

- Stores the entire document in memory, It is memory inefficient
- As Dom was written for any language, method naming conventions don't follow standard java programming conventions

**DOM or SAX****DOM**

- Suitable for small documents
- Easily modify document
- Memory intensive; Load the complete XML document

**SAX**

- Suitable for large documents; saves significant amounts of memory
- Only traverse document once, start to end
- Event driven
- Limited standard functions.
- 

**Loading an XML file:one.html**

```
<html> <body>
<script type=||text/javascript||>
try
{
xmlDocument=new ActiveXObject(-Microsoft.XMLDOM||);
}
catch(e)
{
try {
xmlDocument=document.implementation.createDocument("", "",null);
}
catch(e){alert(e.message)}
}
try
{
xmlDocument.async=false;
xmlDocument.load(-faculty.xml||);
document.write(-XML document student is loaded||);
}
catch(e){alert(e.message)}
</script>
</body> </html>
```

**faculty.xml:**

```
<?xml version=||1.0||?>
< faculty >
 <eno>30</eno>
 <personal_inf>
 <name>Kalpana</name>
 <address>Hyd</address>
 <phone>9959967192</phone>
 </personal_inf>
 <dept>CSE</dept>
 <col>MRCET</col>
 <group>MRGI</group>
</faculty>
```

**OUTPUT:** XML document student is loaded

**ActiveXObject:** It creates empty xml document object.

**Use separate function for Loading an XML document: two.html**

```

<html> <head>
<script type="text/javascript">
Function My_function(doc_file)
{
try
{
xmlDocument=new ActiveXObject("Microsoft.XMLDOM");
}
catch(e)
{
try
{
xmlDocument=document.implementation.createDocument("", "", null);
}
catch(e){ alert(e.message)}
}
try
{
xmlDocument.async=false;
xmlDocument.load("faculty.xml");
return(xmlDocument);
}
catch(e){ alert(e.message)}
return(null);
}
</script>
</head>
<body>
<script type="text/javascript">
xmlDoc=My_function("faculty.xml");
document.write("XML document student is loaded");
</script>
</body> </html>

```

**OUTPUT:** XML document student is loaded

**Use of properties and methods: three.html**

```

<html> <head>
<script type="text/javascript" src="my_function_file.js"></script>
</head> <body>
<script type="text/javascript">
xmlDocument=My_function("faculty.xml");
document.write("XML document faculty is loaded and content of this file is:");
document.write("
");
document.write("ENO:");
xmlDocument.getElementsByTagName("eno")[0].childNodes[0].nodeValue);
document.write("
");
document.write("Name:");
xmlDocument.getElementsByTagName("name")[0].childNodes[0].nodeValue);

```

```

document.write(
||);
document.write(-ADDRESS:||+
xmlDocument.getElementsByTagName(-address||)[0].childNodes[0].nodeValue);
document.write(
||);
document.write(-PHONE:||+
xmlDocument.getElementsByTagName(-phone||)[0].childNodes[0].nodeValue);
document.write(
||);
document.write(-DEPARTMENT:||+
xmlDocument.getElementsByTagName(-dept||)[0].childNodes[0].nodeValue);
document.write(
||);
document.write(-COLLEGE:||+
xmlDocument.getElementsByTagName(-coll||)[0].childNodes[0].nodeValue);
document.write(
||);
document.write(-GROUP:||+
xmlDocument.getElementsByTagName(-group||)[0].childNodes[0].nodeValue);
</script>
</body>
</html>

```

**OUTPUT:**

XML document faculty is loaded and content of this file is

```

ENO: 30
NAME: Kalpana
ADDRESS: Hyd
PHONE: 9959967192
DEPARTMENT: CSE
COLLEGE: MRCET
GROUP: MRGI

```

**We can access any XML element using the index value: four.html**

```

<html> <head>
<script type=||text/javascript|| src=||my_function_file.js||></script>
</head> <body>
<script type=||text/javascript||>
xmlDoc=My_function("faculty1.xml");
value=xmlDoc.getElementsByTagName(-name||);
document.write(-value[0].childNodes[0].nodeValue||);
</script></body></html>

```

**OUTPUT:** Kalpana

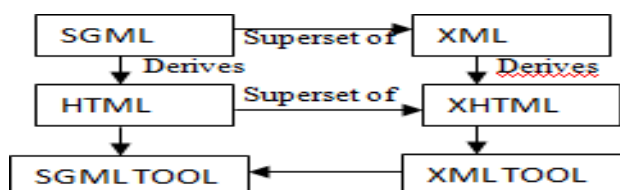
**XHTML: eXtensible Hypertext Markup Language**

**Hypertext** is simply a piece of text that works as a link. **Markup language** is a language of writing layout information within documents. The XHTML recommended by W3C. Basically an XHTML document is a plain text file and it is very much similar to HTML. It contains rich text, means text with tags. The extension to this program should be either **html** or **htm**. These programs can be opened in some web browsers and the corresponding web page can be viewed.

## HTML Vs XHTML

HTML	XHTML
1. The HTML tags are case insensitive. EX: <BoDy>-----</body>	1. The XHTML tags are casesensitive. EX: <body> -----</body>
2. We can omit the closing tags sometimes.	2. For every tag there must be a closing tag. EX: <h1>-----</h1> or <h1----- />
3. The attribute values not always necessary to quote.	3. The attribute values are must be quoted.
4. In HTML there are some implicit attribute values.	4. In XHTML the attribute values must be specified explicitly.
5. In HTML even if we do not follow the nesting rules strictly it does not cause much difference.	5. In XHTML the nesting rules must be strictly followed. These nesting rules are- - A form element cannot contain another form element. -an anchor element does not contain another form element -List element cannot be nested in the list element -If there are two nested elements then the inner element must be enclosed first before closing the outer element -Text element cannot be directly nested in form element

The relationship between SGML, XML, HTML and XHTML is as given below



**Standard structure:** DOCTYPE, html, head and body

The doctype is specified by the DTD. The XHTML syntax rules are specified by the file xhtml11.dtd file. There are 3 types of DTDs.

1. **XHTML 1.0 Strict:** clean markup code
2. **XHTML 1.0 Transitional:** Use some html features in the existing XHTML document.
3. **XHTML 1.0 Frameset:** Use of Frames in an XHTML document.

**EX:**

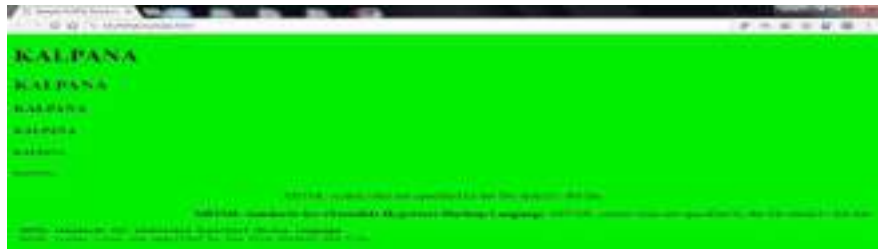
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml11.dtd">
```

```
<html xmlns="http://www.w3c.org/1999/xhtml">
<head>
<title>Sample XHTML Document</title>
</head>
<body bgcolor=||#FF0000||>
<basefont face=||arial|| size=||14|| color=||white||>
```

```

<h1>MRCET</h1>
<h2>MRCET</h2>
<h3>MRCET</h3>
<h4> KALPANA </h4>
<h5> KALPANA </h5>
<h6>KALPANA</h6>

```



```

<p><center> XHTML syntax rules are specified by the file xhtml11.dtd file. </center></p>
<div align="right"> XHTML standards for eXtensible Hypertext Markup Language.
 XHTML syntax rules are specified by the file xhtml11.dtd file.</div>
<pre> XHTML standards for <i>eXtensible Hypertext Markup Language.</i>
 XHTML syntax rules are specified by the file xhtml11.dtd file.</pre>
</basefont>
</body>
</html>

```

## DOM in JAVA

### DOM interfaces

The DOM defines several Java interfaces. Here are the most common interfaces:

- **Node** - The base datatype of the DOM.
- **Element** - The vast majority of the objects you'll deal with are Elements.
- **Attr** Represents an attribute of an element.
- **Text** The actual content of an Element or Attr.
- **Document** Represents the entire XML document. A Document object is often referred to as a DOM tree.

### Common DOM methods

When you are working with the DOM, there are several methods you'll use often:

- **Document.getDocumentElement()** - Returns the root element of the document.
- **Node.getFirstChild()** - Returns the first child of a given Node.
- **Node.getLastChild()** - Returns the last child of a given Node.
- **Node.getNextSibling()** - These methods return the next sibling of a given Node.
- **Node.getPreviousSibling()** - These methods return the previous sibling of a given Node.
- **Node.getAttribute(attrName)** - For a given Node, returns the attribute with the requested name.

### Steps to Using DOM

Following are the steps used while parsing a document using DOM Parser.

- Import XML-related packages.
- Create a DocumentBuilder
- Create a Document from a file or stream
- Extract the root element
- Examine attributes
- Examine sub-elements



**DOM**

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.*;
public class parsing_DOMDemo
{
 public static void main(String[] args)
 {
 try
 {
 System.out.println("enter the name of XML document ");
 BufferedReader input=new BufferedReader(new InputStreamReader(System.in));
 String file_name=input.readLine();
 File fp=new File(file_name);
 if(fp.exists())
 {
 try
 {
 DocumentBuilderFactory Factory_obj= DocumentBuilderFactory.newInstance();
 DocumentBuilder builder=Factory_obj.newDocumentBuilder();
 InputSource ip_src=new InputSource(file_name);
 Document doc=builder.parse(ip_src);
 System.out.println("file_name+is well- formed.");
 }
 catch (Exception e)
 {
 System.out.println("file_name+is not well- formed.");
 System.exit(1);
 }
 }
 else
 {
 System.out.println("file not found:"+file_name);
 }
 }
 catch(IOException ex)
 {
 ex.printStackTrace();
 }
 }
}
```

**SAX:**

**SAX (the Simple API for XML)** is an event-based parser for xml documents. Unlike a DOM parser, a SAX parser creates no parse tree. SAX is a streaming interface for XML, which means that applications using SAX receive event notifications about the XML document being processed an element, and attribute, at a time in sequential order starting at the top of the document, and ending with the closing of the ROOT element.

- Reads an XML document from top to bottom, recognizing the tokens that make up a well- formed XML document

- Tokens are processed in the same order that they appear in the document
- Reports the application program the nature of tokens that the parser has encountered as they occur
- The application program provides an "event" handler that must be registered with the parser
- As the tokens are identified, callback methods in the handler are invoked with the relevant information

**When to use?**

You should use a SAX parser when:

- You can process the XML document in a linear fashion from the top down
- The document is not deeply nested
- You are processing a very large XML document whose DOM tree would consume too much memory. Typical DOM implementations use ten bytes of memory to represent one byte of XML
- The problem to be solved involves only part of the XML document
- Data is available as soon as it is seen by the parser, so SAX works well for an XML document that arrives over a stream

**Disadvantages of SAX**

- We have no random access to an XML document since it is processed in a forward-only manner
- If you need to keep track of data the parser has seen or change the order of items, you must write the code and store the data on your own
- The data is broken into pieces and clients never have all the information as a whole unless they create their own data structure

**The kinds of events are:**

- The start of the document is encountered
- The end of the document is encountered
- The start tag of an element is encountered
- The end tag of an element is encountered
- Character data is encountered
- A processing instruction is encountered

Scanning the XML file from start to end, each event invokes a corresponding callback method that the programmer writes.

**SAX packages**

**javax.xml.parsers:** Describing the main classes needed for parsing □

**org.xml.sax:** Describing few interfaces for parsing

**SAX classes**

- **SAXParser** Defines the API that wraps an XMLReader implementation class
- **SAXParserFactory** Defines a factory API that enables applications to configure and obtain a SAX based parser to parse XML documents
- **ContentHandler** Receive notification of the logical content of a document.

- **DTDHandler** Receive notification of basic DTD-related events.
- **EntityResolver** Basic interface for resolving entities.
- **ErrorHandler** Basic interface for SAX error handlers.
- **DefaultHandler** Default base class for SAX event handlers.

### SAX parser methods

**StartDocument() and endDocument()** – methods called at the start and end of an XML document.

**StartElement() and endElement()** – methods called at the start and end of a document element.

**Characters()** – method called with the text contents in between the start and end tags of an XML document element.

### ContentHandler Interface

This interface specifies the callback methods that the SAX parser uses to notify an application program of the components of the XML document that it has seen.

- **void startDocument()** - Called at the beginning of a document.
- **void endDocument()** - Called at the end of a document.
- **void startElement(String uri, String localName, String qName, Attributes atts)** - Called at the beginning of an element.
- **void endElement(String uri, String localName, String qName)** - Called at the end of an element.
- **void characters(char[] ch, int start, int length)** - Called when character data is encountered.
- **void ignorableWhitespace(char[] ch, int start, int length)** - Called when a DTD is present and ignorable whitespace is encountered.
- **void processingInstruction(String target, String data)** - Called when a processing instruction is recognized.
- **void setDocumentLocator(Locator locator)** - Provides a Locator that can be used to identify positions in the document.
- **void skippedEntity(String name)** - Called when an unresolved entity is encountered.
- **void startPrefixMapping(String prefix, String uri)** - Called when a new namespace mapping is defined.
- **void endPrefixMapping(String prefix)** - Called when a namespace definition ends its scope.

### Attributes Interface

This interface specifies methods for processing the attributes connected to an element.

- **int getLength()** - Returns number of attributes, etc.

### SAX simple API for XML

```
import java.io.*;
import org.xml.sax;
import org.xml.sax.helpers;
public class parsing_SAXDemo
{
 public static void main(String[] args) throws IOException
 {
```

```

try {
 System.out.println("enter the name of XML document ");
 BufferedReader input=new BufferedReader(new InputStreamReader(System.in));
 String file_name=input.readLine();
 File fp=new File(file_name);
 if(fp.exists())
 {
 try
 {
 XMLReader reader=XMLReaderFactory.createXMLReader();
 reader.parse(file_name);
 System.out.println("file_name+ is well- formed. ");
 }
 catch (Exception e)
 {
 System.out.println(file_name+"is not well- formed.");
 System.exit(1);
 }
 }
 else
 {
 System.out.println("file not found:"+file_name);
 }
}
catch(IOException ex){ex.printStackTrace();}
} }

```

### Differences between DOM and SAX

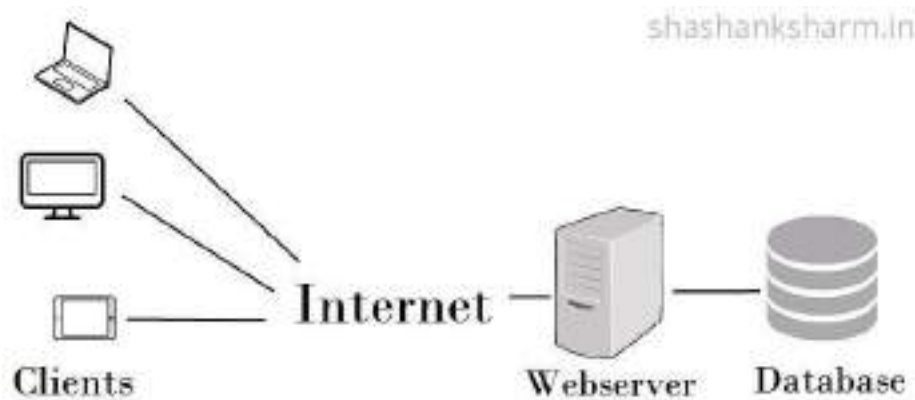
DOM	SAX
Stores the entire XML document into memory before processing	Parses node by node
Occupies more memory	Doesn't store the XML in memory
We can insert or delete nodes	We can't insert or delete a node
DOM is a tree model parser	SAX is an event based parser
Document Object Model (DOM) API	SAX is a Simple API for XML
Preserves comments	Doesn't preserve comments
DOM is slower than SAX, heavy weight.	SAX generally runs a little faster than DOM, light weight.
Traverse in any direction.	Top to bottom traversing is done in this approach
Random Access	Serial Access
<b>Packages required to import</b> import java x.xml.parsers.*; import java x.xml.parsers.Docu mentBuilder; import java x.xml.parsers.Docu mentBuilderFactory;	<b>Packages required to import</b> import java.xml.parsers.*; import org.xml.sax.*; import org.xml.sax.helpers;

## WT III

### Web Servers and Servlets

#### Web Servers:

Web server is a program that uses HTTP (Hypertext Transfer Protocol) to serve the files that form Web pages to users, in response to their requests, which are forwarded by their computers' HTTP clients



#### Examples of servers:

1. Apache Server
2. IIS Server
3. WebLogic
4. JBoss. etc

#### Install TOMCAT web server

While installation, we assign port number 8080 to APACHE. Make sure that these ports are available i.e., no other process is using this port.

#### DESCRIPTION:

##### *Set the JAVA\_HOME Variable*

You must set the JAVA\_HOME environment variable to tell Tomcat where to find Java. Failing to properly set this variable prevents Tomcat from handling JSP pages. This variable should list the base JDK installation directory, not the bin subdirectory. On Windows XP, you could also go to the Start menu, select Control Panel, choose System, click on the Advanced tab, press the Environment Variables button at the bottom, and enter the JAVA\_HOME variable and value directly as:

**Name:** JAVA\_HOME

**Value:** C:\jdk

##### *Set the CLASSPATH*

Since servlets and JSP are not part of the Java 2 platform, standard edition, you have to identify the servlet classes to the compiler. The server already knows about the servlet classes, but the compiler (i.e., javac) you use for development probably doesn't. So, if you

don't set your CLASSPATH, attempts to compile servlets, tag libraries, or other classes that use the servlet and JSP APIs will fail with error messages about unknown classes.

**Name:** `JAVA_HOME`

**Value:** `install_dir/common/lib/servlet-api.jar`

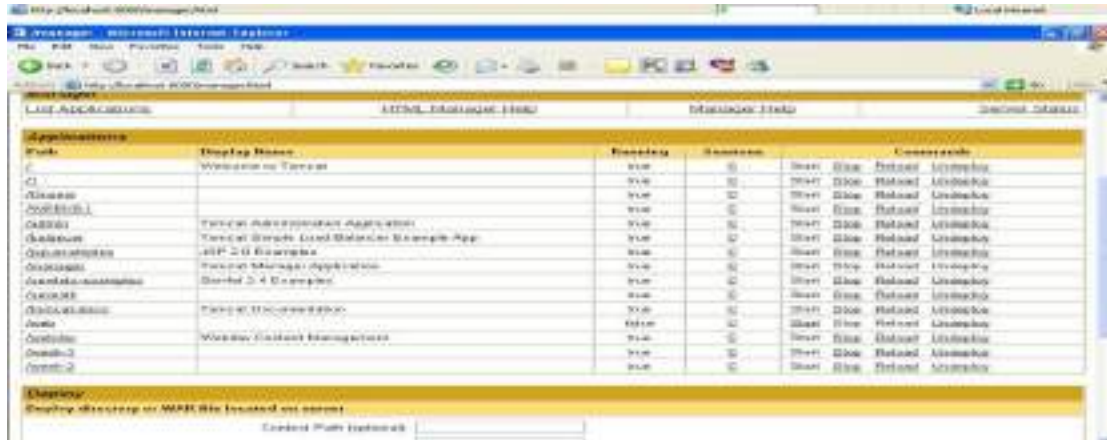
### **Turn on Servlet Reloading**

The next step is to tell Tomcat to check the modification dates of the class files of requested servlets and reload ones that have changed since they were loaded into the server's memory. This slightly degrades performance in deployment situations, so is turned off by default. However, if you fail to turn it on for your development server, you'll have to restart the server every time you recompile a servlet that has already been loaded into the server's memory.



**RESULT:** Thus TOMCAT web server was installed successfully.

Access the developed static web pages for books web site, using these servers by putting the web pages developed in week-1 and week-2 in the document root.



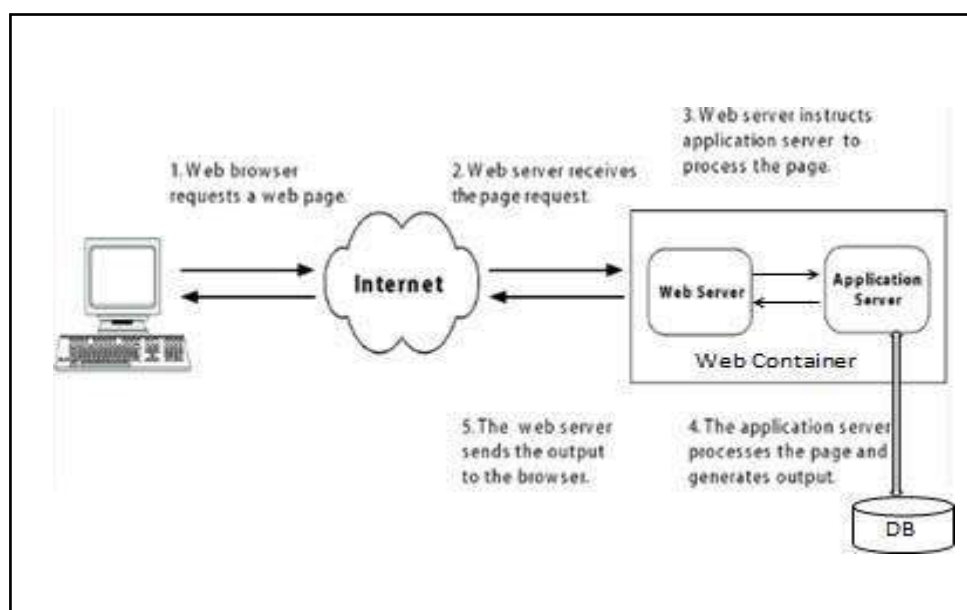
**RESULT:** These pages are accessed using the TOMCAT web server successfully.

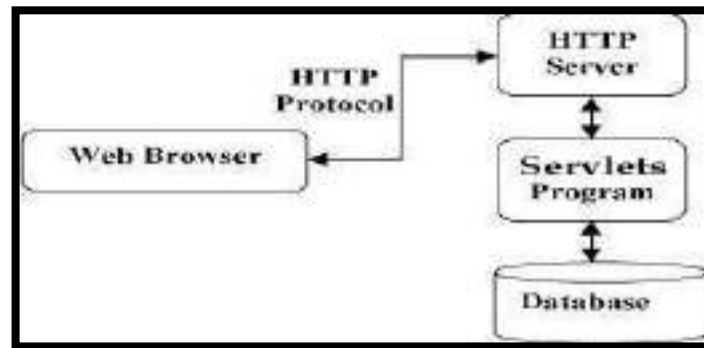
## INTRODUCTION TO SERVLETS

### Servlets:

- **Servlets are server side programs** that run on a Web or Application server and **act as a middle layer** between a requests coming from a client and the server.
- Using Servlets, you can **collect input from users** through web page forms, present records from a database or another source, and create web pages dynamically.
- Servlets don't fork new process for each request, instead a new thread is created.
- Servlets are loaded and ready for each request.
- The same servlet can handle many requests simultaneously.

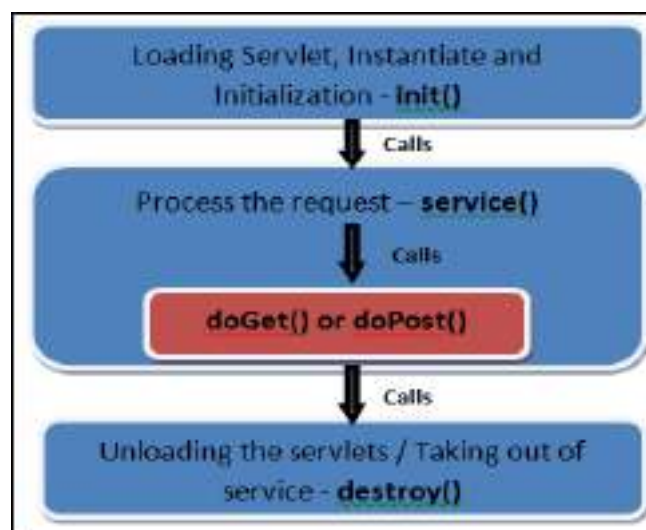
**Web Container:** It is web server that supports servlet execution. Individual Servlets are registered with a container. Tomcat is a popular servlet and JSP container.



**Servlet Architecture:****Servlets Tasks:**

Servlets perform the following major tasks:

- Read the explicit data sent by the clients (browsers). This includes an HTML form on a Web page or it could also come from an applet or a custom HTTP client program.
- Read the implicit HTTP request data sent by the clients (browsers). This includes cookies, media types and compression schemes the browser understands, and so forth.
- Process the data and generate the results. This process may require talking to a database, executing an RMI or CORBA call, invoking a Web service, or computing the response directly.
- Send the explicit data (i.e., the document) to the clients (browsers). This document can be sent in a variety of formats, including text (HTML or XML), binary (GIF images), Excel, etc.
- Send the implicit HTTP response to the clients (browsers). This includes telling the browsers or other clients what type of document is being returned (e.g., HTML), setting cookies and caching parameters, and other such tasks.

**Life Cycle of Servlet****Life Cycle**



**Steps:**

The sequence in which the Web container calls the life cycle methods of a servlet is:

1. The Web container loads the servlet class and creates one or more instances of the servlet class.
2. The Web container invokes `init()` method of the servlet instance during initialization of the servlet. The `init()` method is invoked only once in the servlet life cycle.
3. The Web container invokes the `service()` method to allow a servlet to process a client request.
4. The `service()` method processes the request and returns the response back to the Web container.
5. The servlet then waits to receive and process subsequent requests as explained in steps 3 and 4.
6. The Web container calls the `destroy()` method before removing the servlet instance from the service. The `destroy()` method is also invoked only once in a servlet life cycle.

**The `init()` method :**

The `init` method is designed to be called only once. It is called when the **servlet is first created**, and not called again for each user request.

The servlet is normally created when a user first invokes a URL corresponding to the servlet.

When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to `doGet` or `doPost` as appropriate. The `init()` method simply creates or loads some data that will be used throughout the life of the servlet.

```
public void init() throws ServletException {
 // Initialization code...
}
```

**The `service()` method :**

- The `service()` method is the main method to **perform the actual task**.
- The servlet container (i.e. web server) calls the `service()` method to handle requests coming from the client( browsers) and to write the formatted response back to the client.

Each time the server receives a request for a servlet, the server spawns a new thread and calls `service`.

The `service()` method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls `doGet`, `doPost`, `doPut`, `doDelete`, etc. methods as appropriate.

```
public void service(ServletRequest request,
 ServletResponse response)
 throws ServletException, IOException {
}
```

**The `doGet()` Method**

The `doGet()` method processes client request, which is sent by the client, using the HTTP GET method.

To handle client requests that are received using GET method, we need to override the `doGet()` method in the servlet class.

In the `doGet()` method, we can retrieve the client information of the `HttpServletRequest` object. We can use the `HttpServletResponse` object to send the response back to the client.

```
public void doGet(HttpServletRequest request,
```

```
HttpServletResponse response)
throws ServletException,IOException{
// Servlet code
}
```

**The doPost() Method:**

- The doPost() method handles requests in a servlet, which is sent by the client, using the HTTP POST method.
- For example, if a client is entering registration data in an HTML form, the data can be sent using the POST method.
- Unlike the GET method, the POST request sends the data as part of the HTTP request body. As a result, the data sent does not appear as a part of URL.
- To handle requests in a servlet that is sent using the POST method, we need to override the doPost() method. In the doPost() method, we can process the request and send the response back to the client.

```
publicvoiddoPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException,IOException{
// Servlet code
}
```

**The destroy() method :**

- The destroy() method is called only once at the end of the life cycle of a servlet.
- This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.
- After the destroy() method is called, the servlet object is marked for garbage collection.

```
publicvoid destroy()
{
// Finalization code...
}
```

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {
 private String message;

 public void init() throws ServletException {
 // Do required initialization
 message = "Hello KALPANA";
 }

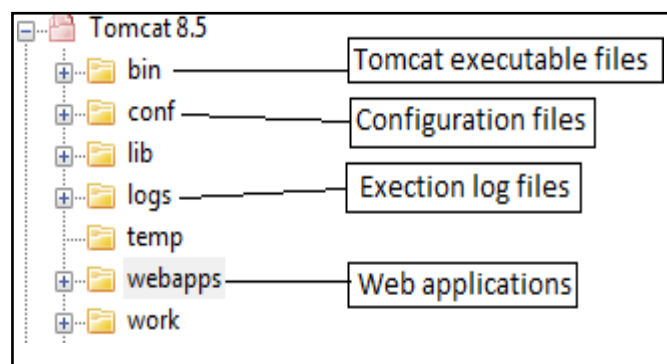
 public void doGet(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {
 // Set response content type
 response.setContentType("text/html");
 // Actual logic goes here.
 PrintWriter out = response.getWriter();
 out.println("<h1>" + message + "</h1>");
 }

 public void destroy() {
 // do nothing.
 }
}
```



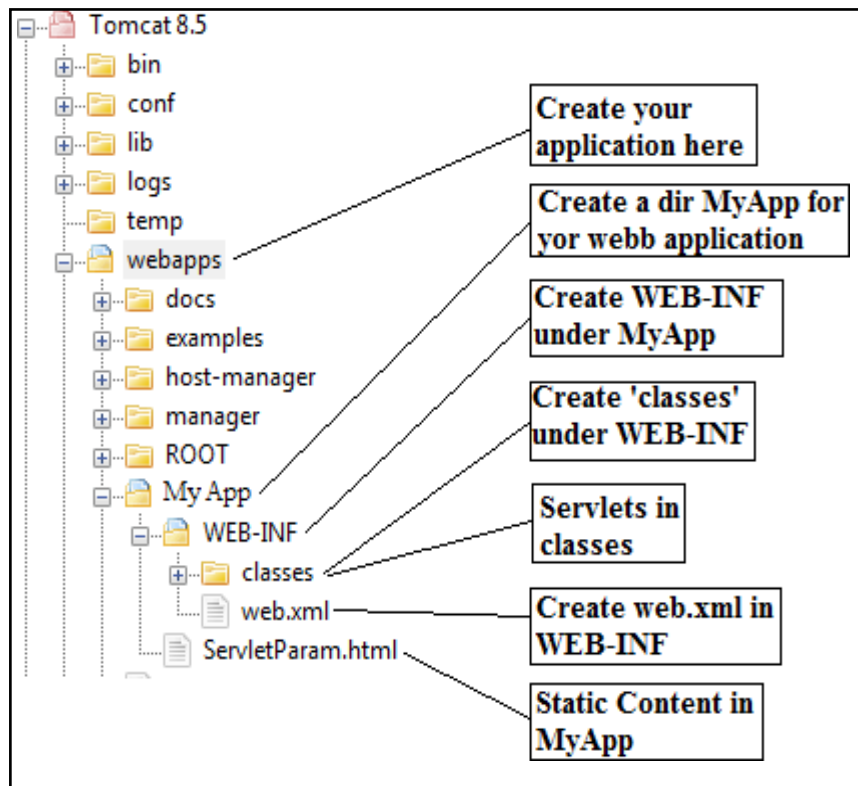
### Deploying a Servlet- Steps:

1. Download and install the **Java Software Development kit(SDK)**.
2. Download a server(Tomcat).
3. Configure the server
  - After installation Tomcat folder will contain -Start Tomcat|| and -Stop Tomcat|| shortcuts.
  - The JAVA\_HOME environment variable should be set so that Tomcat can find JDK  
JAVA\_HOME = c:\jdk1.5
4. Setup deployment environment



**Tomcat Directory Structure**

- To set up a new application, add a directory under the **webapps** directory and create a subdirectory called **WEB-INF**.
- WEB-INF needs to contain **web.xml** (servlet configuration file)
- After WEB-INF dir is created, create a subdirectory **classes** under it. Java classes will go under this directory.



### 5. Creating ServletDemoServlet

There are three different ways to create a servlet.

- By implementing **Servlet** interface
- By extending **GenericServlet** class
- By extending **HttpServlet** class

### 6. Compile Servlet and save the class file in **classes** folder.

### 7. Create a Deployment Descriptor

- The **deployment descriptor** is an xml file, from which Web Container gets the information about the servlet to be invoked.
  - The web container uses the Parser to get the information from the web.xml file.
  - Add a servlet entry and a servlet-mapping entry for each servlet for Tomcat to run.
- Add entries after <web-app> tag inside web.xml

```
<web-app>
 <servlet>
 <servlet-name>Demo</servlet-name>
 <servlet-class>DemoServlet</servlet-class>
 </servlet>
 <servlet-mapping>
 <servlet-name>Demo</servlet-name>
 <url-pattern>/welcome</url-pattern>
 </servlet-mapping>
</web-app>
```

### 8. Start Tomcat server

### 9. Open browser and type <http://localhost/MyApp/DemoServlet>

## **Servlet API**

Servlet API consists of two important packages that encapsulates all the important classes and interface, namely :

1. **javax.servlet**
2. **javax.servlet.http**

### **L. javax.servlet**

#### **Interfaces**

1. Servlet – Declares life cycle methods for a servlet.
2. ServletConfig – To get initialization parameters
3. ServletContext- To log events and access information
4. ServletRequest- To read data from a client request
5. ServletResponse – To write data from client response

#### **Classes**

1. GenericServlet – Implements Servlet and ServletConfig
2. ServletInputStream – Provides an input stream for reading client requests.
3. ServletOutputStream - Provides an output stream for writing responses to a client.
4. ServletException – Indicates servlet error occurred.
5. UnavailableException - Indicates servlet is unavailable

## **Interfaces**

### **1.Servlet Interface**

Declares life cycle methods for a servlet.

- Servlet interface **provides common behaviour to all the servlets.**
- Servlet interface needs to be implemented for creating any servlet (either directly or indirectly).

It provides 3 life cycle methods that are used to initialize the servlet, to service the requests, and to destroy the servlet and 2 non-life cycle methods.

public void service(ServletRequest request, ServletResponse response)	provides response for the incoming request. It is invoked at each request by the web container.
public ServletConfig getServletConfig()	returns the object of ServletConfig.

```
import java.io.*;
import javax.servlet.*;
public class First implements Servlet{
 ServletConfig config=null;
 public void init(ServletConfig config){
```

```
 this.config=config;
 System.out.println("servlet is initialized");
 }
 public void service(ServletRequest req,ServletResponse res)
 throws IOException,ServletException{
 res.setContentType("text/html");
 PrintWriter out=res.getWriter();
 out.print("<html><body>");
 out.print("hello KALPANA");
 out.print("</body></html>");
 }
 public void destroy(){
 System.out.println("servlet is destroyed");
 }
 public ServletConfig getServletConfig(){
 return config;
 }
 public String getServletInfo(){
 return "copyright 2007-1010";
 }
}
```

## 2. ServletConfig interface

- To get initialization parameters
- When the **Web Container** initializes a servlet, it creates a **ServletConfig** object for the servlet. ServletConfig object is used to pass information to a servlet during initialization by getting configuration information from **web.xml**(Deployment Descriptor).

### Methods

- getInitParameter(String name): -->returns a String value initialized parameter
- getInitParameterNames():---> returns the names of the servlet's initialization parameters as an Enumeration of String objects
- getServletContext():---> returns a reference to the ServletContext
- getServletName(): -->returns the name of the servlet instance

## 3. ServletContext Interface - To log events and access information

- For every **Web application** a **ServletContext** object is created by the web container.
- ServletContext object is used to get configuration information from **Deployment Descriptor**(web.xml) which will be available to any servlet.

### Methods :

- getAttribute(String name) - returns the container attribute with the given name
- getInitParameter(String name) - returns parameter value for the specified parameter name
- getInitParameterNames() - returns the names of the context's initialization parameters as an Enumeration of String objects

- `setAttribute(String name, Object obj)` - set an object with the given attribute name in the application scope
- `removeAttribute(String name)` - removes the attribute with the specified name from the application context

#### **4. Servlet RequestInterface-** To read data from a client request

- True job of a Servlet is to handle client request.
- Servlet API provides two important interfaces **`javax.servlet.ServletRequest`** to encapsulate client request.
- Implementation of these interfaces provides important information about client request to a servlet.

#### **Methods**

- `getAttribute(String name)`, `removeAttribute(String name)`, `setAttribute(String name, Object o)`, `getAttributeName()` – used to store and retrieve an attribute from request.
- **`getParameter(String name)`** - returns value of parameter by name
- `getParameterNames()` - returns an enumeration of all parameter names
- `getParameterValues(String name)` - returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist

#### **5. Servlet ResponseInterface** To write data from client response

- Servlet API provides `ServletResponse` to assist in sending response to client.

#### **Methods**

- **`getWriter()`**- returns a `PrintWriter` object that **can send character text to the client**.
- **`setContentType(String type)`**- **sets the content type of the response** being sent to the client before sending the respond.

### **Classes of `javax.servlet` package:**

#### **1.GenericServlet** class - Creates a `GenericServlet`.

- `GenericServlet` class implements **`Servlet`**, **`ServletConfig`** and **`Serializable`** interfaces.
- `GenericServlet` class can handle any type of request so it is protocol- independent.  
You may create a generic servlet by inheriting the `GenericServlet` class and providing the implementation of the service method.

#### **Methods**

**`public void init(ServletConfig config)`** is used to initialize the servlet.

**`public abstract void service(ServletRequest request, ServletResponse response)`** provides service for the incoming request. It is invoked at each time when user requests for a servlet.

**`public void destroy()`** is invoked only once throughout the life cycle and indicates that servlet is being destroyed.

- **`public ServletConfig getServletConfig()`** returns the object of `ServletConfig`.
- **`public String getServletInfo()`** returns information about servlet such as writer, copyright, version etc.

**`public void init()`** it is a convenient method for the servlet programmers, now there is no need to call `super.init(config)`

- **`public ServletContext getServletContext()`** returns the object of `ServletContext`.
- **`public String getInitParameter(String name)`** returns the parameter value for the given

parameter name.

**public Enumeration getInitParameterNames()** returns all the parameters defined in the web.xml file.

**public String getServletName()** returns the name of the servlet object.

## **2. ServletInputStream Class**

It provides stream to **read binary data** such as image etc. from the request object. It is an abstract class.

The **getInputStream()** method of **ServletRequest** interface returns the instance of ServletInputStream class

**intreadLine(byte[] b, int off, intlen)** it reads the input stream.

## **3. ServletOutputStream Class**

- It provides a stream to write binary data into the response. It is an abstract class.
  - The **getOutputStream()** method of **ServletResponse** interface returns the instance of ServletOutputStream class.
- ServletOutputStream class provides **print()** and **println()** methods that are overloaded.

## **4. ServletException and UnavailableException**

ServletException is a general exception that the servlet container will catch and log. The cause can be anything.

- The exception contains a root cause exception.
- Defines an exception that a servlet or filter throws to indicate that it is permanently or temporarily unavailable.

When a servlet or filter is permanently unavailable, something is wrong with it, and it cannot handle requests until some action is taken. For example, a servlet might be configured incorrectly, or a filter's state may be corrupted.

## **2. javax.servlet.http**

1. HttpServletRequest
2. HttpServletResponse
3. HttpSession

### **Classes of javax.servlet.http package**

1. HttpServlet
2. Cookie

### **Interfaces of javax.servlet.http package**

#### **1. HttpServletRequest - reads the data from a client**

**HttpServletRequest** Extends the ServletRequest interface to provide request information for HTTP servlets.

The servlet container creates an HttpServletRequest object and passes it as an argument to the servlet's service methods (doGet, doPost, etc).

METHODS:

```
getParameter(),
getParameterValues()
```



`getParameterNames()` are offered as ways to access the arguments.

### **2. *HTTPServletResponse*- Used to create and send a HTTP response**

Extends the `ServletResponse` interface and can perform these tasks

#### **1. Set Response Codes –**

The response code for a request is a numeric value that represents the status of the response. For example, 200 represents a successful response, 404 represents a file not found.

#### **2. Set Headers –**

Headers for the response can be set by calling `setHeader`, specifying the name and value of the header to be set.

#### **3. Send Redirects –**

The `sendRedirect` method is used to issue a redirect to the browser, causing the browser to issue a request to the specified URL. The URL passed to `sendRedirect` must be an absolute URL—it must include protocol, machine, full path, and so on.

METHODS:

```
public int getStatus()
public void sendRedirect()
public void setHeader();
public String getHeader();
```

### **3. HttpSession**

**HttpSession** object is used to store entire session with a specific client.

We can store, retrieve and remove attribute from **HttpSession** object.

Any servlet can have access to **HttpSession** object throughout the `getSession()` method of the **HttpServletRequest** object.

METHODS:

```
public String getId()
public void setAttribute()
```

## **Classes of `javax.servlet.http` package**

### **1. HTTPServlet: used to create a http servlet**

`HttpServlet` extends from `GenericServlet` and does not override `init`, `destroy` and other methods.

It implements the `service()` method which is an abstract method in `GenericServlet`.

A subclass of `HttpServlet` must override at least one method, usually one of these:

- **`doGet()`, if the servlet supports HTTP GET requests**
- **`doPost()`, for HTTP POST requests**
- `init()` and `destroy()`, to manage resources that are held for the life of the servlet
- `getServletInfo()`, which the servlet uses to provide information about itself

## 2. Cookie

- A **cookie** is a small piece of information that is persisted between the multiple client requests.
- **javax.servlet.http.Cookie** class provides the functionality of using cookies. It provides a lot of useful methods for cookies.
- **public void addCookie(Cookie ck):** method of HttpServletResponse interface is used to add cookie in response object.
- **public Cookie[] getCookies():** method of HttpServletRequest interface is used to return all the cookies from the browser.

### Reading Servlet Parameters(or) Handling HTTPRequest and HTTPResponse

- The parameters are the way in which a client or user can send information to the Http Server.
- The **HttpServletRequest** interface includes methods that allow you to read the names and values of parameters that are included in a client request.
- The **HttpServletResponse** Interface provides functionality for sending response to client.
- The browser uses two methods to pass this information to web server. These methods are GET Method and POST Method.

#### GET method:

- The GET method sends the encoded user information appended to the page request.
- The page and the encoded information are separated by the ? character as follows:

**http://www.test.com/hello?key1=value1&key2=value2**

- The GET method is the default method to pass information from browser to web server.
- Never use the GET method if you have password or other sensitive information to pass to the server.
- The GET method has size limitation: only 1024 characters can be in a request string.
- This information is passed using QUERY\_STRING header and will be accessible through QUERY\_STRING environment variable.
- Servlet handles this type of requests using **doGet()** method.

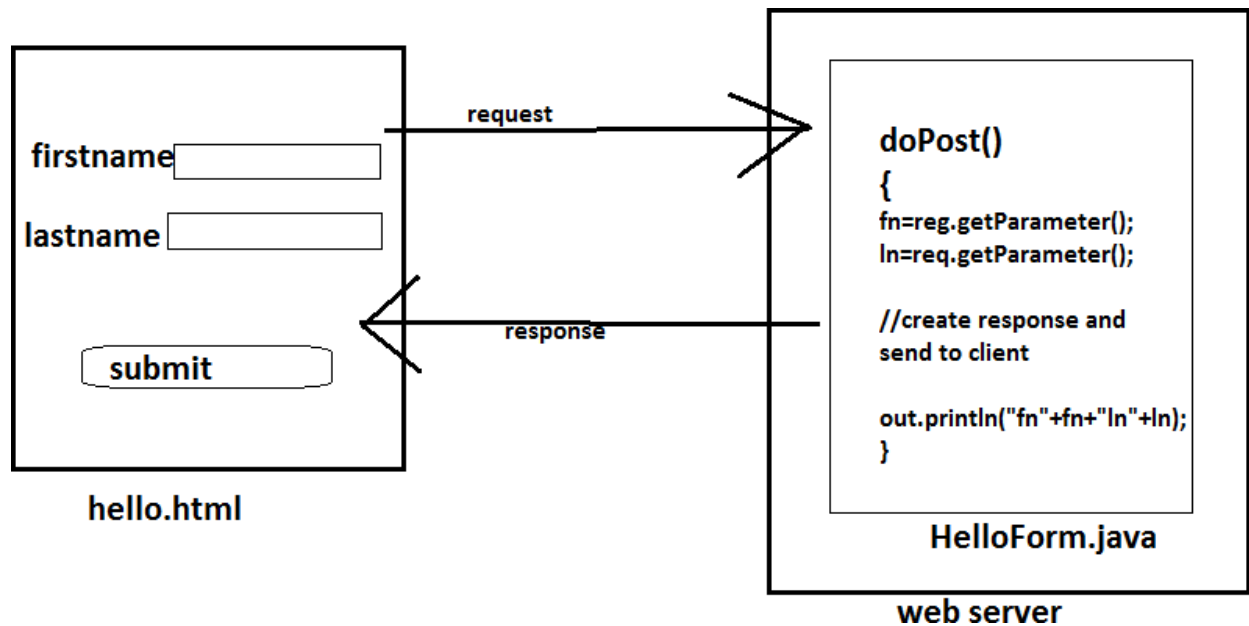
#### POST method:

- A generally more reliable method of passing information to a backend program is the POST method.
- This message comes to the backend program in the form of the standard input which you can parse and use for your processing.
- Servlet handles this type of requests using **doPost()** method.

## Reading Form Data using Servlet:

Servlets handles form data parsing automatically using the following methods depending on the situation:

- **getParameter():** You call request.getParameter() method to get the value of a form parameter.
- **getParameterValues():** Call this method if the parameter appears more than once and returns multiple values, for example checkbox.
- **getParameterNames():** Call this method if you want a complete list of all parameters in the current request.



- First create a **HTML page** Hello.html and put it in <Tomcat- installation-directory>/webapps/ROOT directory

```
<html>
<body>
<form action="HelloForm"method="GET">
 First Name:
 <inputtype="text"name="first_name">
 Last Name:
 <inputtype="text"name="last_name"/></br>
 <inputtype="submit"value="Submit"/>
</form>
</body>
</html>
```

### POST method example

Let us consider **HelloForm.java**

```
import java.io.*;
import java.util.*;
import javax.servlet.http.*;
public class HelloForm extends HttpServlet {
 public void doPost(HttpServletRequest request,HttpServletResponse response)throws
 IOException,ServletException{
 PrintWriter pw = response.getWriter();
 String fn=request.getParameter("first_name");
 String dln=request.getParameter("last_name")
 pw.print(" your first name is"+fn);
 pw.print(" your last name is"+ln);
 pw.close();
 }
}
```

- Compile HelloForm.java as follows: `$javac HelloForm.java`
- Compilation would produce **HelloForm.class** file.
- Next you would have to copy this class file in  
    <Tomcat- installation-directory>/webapps/ROOT/WEB-INF/classes
- Create following entries in **web.xml** file located in  
    <Tomcat- installation-directory>/webapps/ROOT/WEB-INF/  
    <servlet>  
    <servlet-name>HelloForm</servlet-name>  
    <servlet-class>HelloForm</servlet-class>  
    </servlet>  
  
    <servlet-mapping>  
    <servlet-name>HelloForm</servlet-name>  
    <url-pattern>/HelloForm</url-pattern>  
    </servlet-mapping>
- When you access ***http://localhost:8080/Hello.html***, then output of the above form.

First Name:

Last Name:

- Start Tomcat Server and open browser.
- Now enter firstname and lastname, Click Submit
- It will generate result

your firstname is: naveen  
your firstname is: Mrcet

## Reading Initialization Parameters

Reading initialization can be done by using

### 1. ServletConfig

### 2. ServletContext

#### 1. Using Servlet Config:

- An object of **ServletConfig** is created by the web container for each servlet. This object can be used to get configuration information from web.xml file.
- If the configuration information is modified from the web.xml file, we don't need to change the servlet. So it is easier to manage the web application if any specific content is modified from time to time.

#### Methods

- `getInitParameter(String name)`: returns a String value initialized parameter
- `getInitParameterNames()`: returns the names of the servlet's initialization parameters as an Enumeration of String objects
- `getServletContext()`: returns a reference to the ServletContext
- `getServletName()`: returns the name of the servlet instance

#### Syntax to provide the initialization parameter for a servlet

The `init-param` sub-element of `servlet` is used to specify the initialization parameter for a servlet.

```

<web-app>
 <servlet>

 <init-param>
 <param-name>email</param-name>
 <param-value>kalpana@gamil.com</param-value>
 </init-param>

 </servlet>
</web-app>

```

**Retrieve ServletConfig**

```

ServletConfigsc = getServletConfig();
out.println(sc.getInitParameter("email"));

```

**Ex: we b.xml**

```

<web-app>
 <servlet>
 <servlet-name>TestInitParam</servlet-name>
 <servlet-class>TestInitParam</servlet-class>
 <init-param>
 <param-name>email</param-name>
 <param-value>kalpana@gmail.com</param-value>
 </init-param>
 </servlet>
 <servlet-mapping>
 <servlet-name>TestInitParam</servlet-name>
 <url-pattern>/TestInitParam</url-pattern>
 </servlet- mapping>
</web-app>

```

**TestInitParam.java**

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class TestInitParam extends HttpServlet {
 protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
 response.setContentType("text/html;charset=UTF-8");
 PrintWriter out = response.getWriter();
 ServletConfigsc=getServletConfig();
 out.print("<html><body>");
 out.print(""+sc.getInitParameter("email")+"");
 out.print("</body></html>");
 out.close();
 }
}

```

- It will generate result

## 2. Using ServletContext

- An object of ServletContext is created by the web container at time of deploying the project.
- This object can be used to get configuration information from web.xml file.
- There is only one ServletContext object per web application.
- If any information is shared to many servlet, it is better to provide it from the web.xml file using the **<context-param>** element.

### Advantage

- **Easy to maintain** if any information is shared to all the servlet, it is better to make it available for all the servlet.
- We provide this information from the web.xml file, so if the information is changed, we don't need to modify the servlet. Thus it removes maintenance problem.

### Uses

1. The object of ServletContext provides an interface between the container and servlet.
2. The ServletContext object can be used to get configuration information from the web.xml file.
3. The ServletContext object can be used to set, get or remove attribute from the web.xml file.
4. The ServletContext object can be used to provide inter-application communication.

### Methods:

- `getAttribute(String name)` - returns the container attribute with the given name
- `getInitParameter(String name)` - returns parameter value for the specified parameter name
- `getInitParameterNames()` - returns the names of the context's initialization parameters as an Enumeration of String objects
- `setAttribute(String name, Object obj)` - set an object with the given attribute name in the application scope
- `removeAttribute(String name)` - removes the attribute with the specified name from the application context

### Retrieve ServletContext

`ServletContextapp = getServletContext();`

*OR*

`ServletContextapp = getServletConfig().getServletContext();`

### Ex: web.xml

```
<web-app>
 <context-param>
 <param-name>driverName</param-name>
 <param-value>sun.jdbc.JdbcOdbcDriver</param-value>
 </context-param>

 <servlet>
 <servlet-name>TestServletContext</servlet-name>
```

```
<servlet-class>TestServletContext</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>TestServletContext</servlet-name>
 <url-pattern>/TestServletContext</url-pattern>
</servlet-mapping>
</web-app>
```

**TestServletContext.java**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

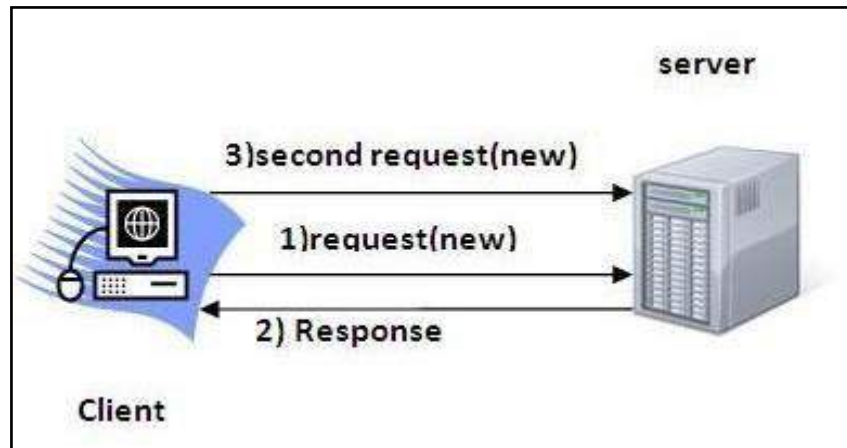
public class TestServletContext extends HttpServlet {
 protected void doGet(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {
 response.setContentType("text/html;charset=UTF-8");
 PrintWriter out = response.getWriter();
 ServletContextsc = getServletContext();
 out.println(sc.getInitParameter("driverName"));
 }
}
```

- It will generate result

sun.jdbc.JdbcOdbcDriver

## **Session Tracking**

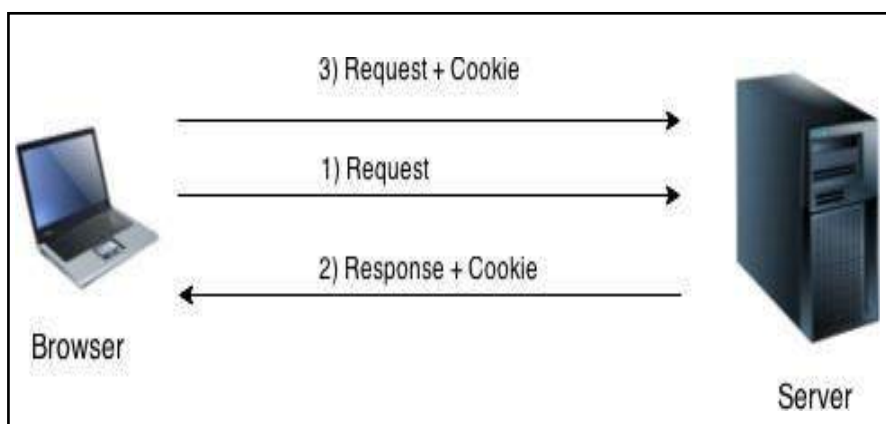
- Session simply means a **particular interval of time**.
- Session Tracking is a way to maintain state (data) of an user.
- Http protocol is a stateless, each request is considered as the new request, so we need to maintain state using session tracking techniques.
- Each time user requests to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize to particular user.



- We use session tracking to recognize the user. It is used to recognize the particular user.
- Session Tracking Techniques
  - Cookies
  - Hidden Form Field
  - URL Rewriting
  - HttpSession

## 1. Using Cookies

- Cookies are text files stored on the client computer and they are kept for various information tracking purpose.
- There are three steps involved in identifying returning users:
  - Server script sends a set of cookies to the browser in response header.
  - Browser stores this information on local machine for future use.
  - When next time browser sends any request to web server then it sends those cookies information to the server in request header and server uses that information to identify the user.
- Cookies are created using **Cookie** class present in Servlet API.
- For adding cookie or getting the value from the cookie, we need some methods provided by other interfaces. They are:
  - a. **public void addCookie(Cookie ck):** method of HttpServletResponse interface is used to add cookie in response object.
  - b. **public Cookie[] getCookies():** method of HttpServletRequest interface is used to return all the cookies from the browser.





**Disadvantage of Cookies**

- It will not work if cookie is disabled from the browser.
- Only textual information can be set in Cookie object.

**Methods**

	Sets the maximum age of the cookie in seconds.
	Returns the name of the cookie. The name cannot be changed after creation.
	Returns the value of the cookie.
	changes the name of the cookie.
	changes the value of the cookie.

**Create Cookie**

```
Cookie ck=new Cookie("user","kalpana ");//creating cookie object
response.addCookie(ck);//adding cookie in the response
```

**Delete Cookie**

It is mainly used to logout or signout the user.

```
Cookie ck=new Cookie("user","");//deleting value of cookie
ck.setMaxAge(0);//changing the maximum age to 0 seconds
response.addCookie(ck);//adding cookie in the response
```

**Get Cookies**

```
Cookie ck[]=request.getCookies();
for(int i=0;i<ck.length;i++)
 out.print("
" +ck[i].getName()+" "+ck[i].getValue());

//printing name and value of cookie
```

**Sending the Cookie into the HTTP response headers:**

We use **response.addCookie** to add cookies in the HTTP response header as follows:

```
response.addCookie(cookie);
```

**Ex: List and AddCookie.java**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class ListandAddCookieextends HttpServlet {
 protected void doGet(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {
 response.setContentType("text/html");
 PrintWriter out = response.getWriter();
 Cookie cookie = null;
 out.println("<html><body>" +
 "<form method='get' action='/mrcet/CookieLab'>" +
 "Name:<input type='text' name='user' />
" +
```

```
"Password:<input type='text' name='pass' >
" +
"<input type='submit' value='submit'>" +
"</form>");
```

```
String name = request.getParameter("user");
String pass = request.getParameter("pass");
```

```
if(!pass.equals("") || !name.equals("")) {
 Cookie ck = new Cookie(name,pass);
 response.addCookie(ck);
}
```

```
Cookie[] cookies = request.getCookies();
if(cookies != null){
 out.println("<h2> Found Cookies Name and Value</h2>");
 for (inti = 0; i<cookies.length; i++){
 cookie = cookies[i];
 out.print("Cookie Name : " + cookie.getName() + ", ");
 out.print("Cookie Value: " + cookie.getValue()+"
");
 }
}
out.println("</body></html>");
```

```
}
```

**we b.xml**

```
<web-app>
 <servlet>
 <servlet-name>ListandAddCookie</servlet-name>
 <servlet-class>ListandAddCookie</servlet-class>
 </servlet>
 <servlet-mapping>
 <servlet-name>ListandAddCookie</servlet-name>
 <url-pattern>/ListandAddCookie</url-pattern>
 </servlet- mapping>
</web-app>
```



**Found Cookies Name and Value**

Cookie Name : Kalpana, Cookie Value: 123456

## 2.Session

- HttpSession Interface provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user.
- Web container creates a session id for each user. The container uses this id to identify the particular user.
- The servlet container uses this interface to create a session between an HTTP client and an HTTP server.
- The session persists for a specified time period, across more than one connection or page request from the user.

### Get the HttpSession object

The HttpServletRequest interface provides two methods to get the object of HttpSession:

1. **public HttpSession getSession():** Returns the current session associated with this request, or if the request does not have a session, creates one.
2. **public HttpSession getSession(boolean create):** Returns the current HttpSession associated with this request or, if there is no current session and create is true, returns a new session.

### Destroy Session

```
session.invalidate();
```

### Set/Get data in session

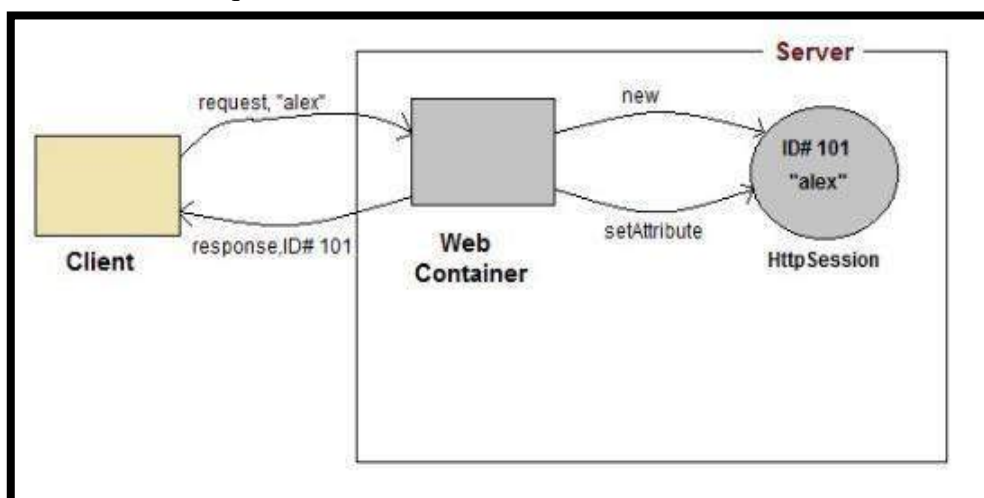
```
session.setAttribute(name,value);
session.getAttribute(name);
```

### Methods

1. **public String getId():** Returns a string containing the unique identifier value.
2. **public long getCreationTime():** Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
3. **public long getLastAccessedTime():** Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
4. **public void invalidate():** Invalidates this session then unbinds any objects bound to it.

### Steps

- On client's first request, the **Web Container** generates a unique session ID and gives it back to the client with response. This is a temporary session created by webcontainer.
- The client sends back the session ID with each request. Making it easier for the web container to identify where the request is coming from.
- The **Web Container** uses this ID, finds the matching session with the ID and associates the session with the request.



**Ex: SessionTrack.java**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionTrack extends HttpServlet {

 public void doGet(HttpServletRequest request,
 HttpServletResponse response) throws ServletException, IOException {
 // Create a session object if it is already not created.
 HttpSession session = request.getSession(true);

 String title = "Welcome to my website";
 String userID = "";
 Integer visitCount = new Integer(0);

 if (session.isNew())
 {
 userID = "Kalpana";
 session.setAttribute("UserId", "Kalpana");
 }
 else {
 visitCount = (Integer)session.getAttribute("visitCount");
 visitCount = visitCount + 1;
 userID = (String)session.getAttribute("UserId");
 }
 session.setAttribute("visitCount", visitCount);

 response.setContentType("text/html");
 PrintWriter out = response.getWriter();

 out.println("<html>" +
 "<body>" +
 "<h1>Session Infomation</h1>" +
 "<table border='1'>" +
 "<tr><th>Session info</th><th>value</th></tr>" +
 "<tr><td>id</td><td>" + session.getId() + "</td></tr>" +
 "<tr><td>User ID</td><td>" + userID + "</td></tr>" +
 "<tr><td>Number of visits</td><td>" + visitCount + "</td></tr>" +
 "</table></body></html>");
 }
}
```

**web.xml**

```
<web-app>
 <servlet>
 <servlet-name>SessionTrack</servlet-name>
 <servlet-class>SessionTrack</servlet-class>
 </servlet>
 <servlet-mapping>
 <servlet-name>SessionTrack</servlet-name>
 <url-pattern>/SessionTrack</url-pattern>
 </servlet-mapping>
</web-app>
```

**Output:**

The screenshot shows a web browser window with the title 'Session Information'. The browser's address bar displays 'localhost:8080/SessionTrack'. The main content area features a table with session data.

Session info	value
id	8B74921FDAE0591A0F0ECE9DECC5B516
User ID	Kalpana
Number of visits	6

## UNIT IV

### JAVA SERVER PAGES

**Introduction to JSP: The Problem with Servlet. The Anatomy of a JSP Page, JSP Processing. JSP Application Design with MVC Setting Up and JSP Environment, JSP Declarations, Directives, Expressions, Code Snippets, implement objects, Requests, Using Cookies and Session for Session**

The Servlet technology and JavaServer Pages (JSP) are the two main technologies for developing java Web applications. When first introduced by Sun Microsystems in 1996, the Servlet technology was considered superior to the reigning Common Gateway Interface (CGI) because servlets stay in memory after they service the first requests. Subsequent requests for the same servlet do not require instantiation of the servlet's class therefore enabling better response time.

Servlets are Java classes that implement the `javax.servlet.Servlet` interface. They are compiled and deployed in the web server. The problem with servlets is that you embed HTML in Java code. If you want to modify the cosmetic look of the page or you want to modify the structure of the page, you have to change code. Generally speaking, this is left to the better hands (and brains) of a web page designer and not to a Java developer.

```
PrintWriter pw = response.getWriter();
pw.println("<html><head><title>Testing</title></head>"); pw.println("<body
bgcolor=\"# ffdddd\">");
```

As seen from the example above this method presents several difficulties to the web developer:

1. The code for a servlet becomes difficult to understand for the programmer.
2. The HTML content of such a page is difficult if not impossible for a web designer to understand or design.
3. This is hard to program and even small changes in the presentation, such as the page's background color, will require the servlet to be recompiled. Any changes in the HTML content require the rebuilding of the whole servlet.
4. It's hard to take advantage of web-page development tools when designing the application interface. If such tools are used to develop the web page layout, the generated HTML must then be manually embedded into the servlet code, a process which is time consuming, error prone, and extremely boring.
5. In many Java servlet-based applications, processing the request and generating the response are both handled by a single servlet class.
6. The servlet contains request processing and business logic (implemented by methods), and also generates the response HTML code, are embedded directly in the servlet code.

JSP solves these problems by giving a way to include java code into an HTML page using scriptlets. This way the HTML code remains intact and easily accessible to web designers, but the page can still perform its task.

In late 1999, Sun Microsystems added a new element to the collection of Enterprise Java tools: JavaServer Pages (JSP). JavaServer Pages are built on top of Java servlets and designed to increase the efficiency in which programmers, and even nonprogrammers, can create web content.

Instead of embedding HTML in the code, you place all static HTML in a JSP page, just as in a regular web page, and add a few JSP elements to generate the dynamic parts of the page. The request processing can remain the domain of the servlet, and the business logic can be handled by JavaBeans and EJBcomponents.

A JSP page is handled differently compared to a servlet by the web server. When a servlet is deployed into a web server in compiled (bytecode) form, then a JSP page is deployed in its original, human-readable form.

When a user requests the specific page, the web server compiles the page into a servlet and from there on handles it as a standard servlet.

This accounts for a small delay, when a JSP page is first requested, but any subsequent requests benefit from the same speed effects that are associated with servlets.

### **The Problem with Servlet**

- Servlets are difficult to code which are overcome in JSP. Other way, we can say, JSP is almost a replacement of Servlets, (by large, the better word is extension of Servlets), where coding decreases more than half.

- In Servlets, both static code and dynamic code are put together. In JSP, they are separated. For example, In Servlets:

```
out.println(-Hello Mr. || + str + || you are great man||);
```

where str is the name of the client which changes for each client and is known as dynamic content. The strings, -Hello Mr. || and -you are great man|| are static content which is the same irrespective of client. In Servlets, in println(), both are put together.

- In JSP, the static content and dynamic content is separated. Static content is written in HTML and dynamic content in JSP. As much of the response comprises of static content (nearly 70%) only, the JSP file more looks as a HTML file.
- Programmer inserts, here and there, chunks of JSP code in a running HTML developed by Designer. As much of the response delivered to client by server comprises of static content (nearly 70%), the JSP file more looks like a HTML file. Other way we can say, JSP is nothing but Java in HTML (servlets are HTML in Java); java code embedded in HTML.
- When the roles of Designer and Programmer are nicely separated, the product development becomes cleaner and fast. Cost of developing Web site becomes cheaper as Designers are much paid less than Programmers, especially should be thought in the present competitive world.
- Both presentation layer and business logic layer put together in Servlets. In JSP, they can be separated with the usage of JavaBeans.
- The objects of PrintWriter, ServletConfig, ServletContext, HttpSession and RequestDispatcher etc. are created by the Programmer in Servlets and used. But in JSP, they are builtin and are known as "implicit objects". That is, in JSP, Programmer never creates these objects and straightaway use them as they are implicitly created and given by JSP container. This decreases lot of coding.
- JSP can easily be integrated with JavaBeans.
- JSP is much used in frameworks like Struts etc.
- With JSP, Programmer can build custom tags that can be called in JavaBeans directly. Servlets do not have this advantage. Reusability increases with tag libraries and JavaBean etc.
- Writing alias name in <url-pattern> tag of web.xml is optional in JSP but mandatory in Servlets.
- A Servlet is simply a Java class with extension .java written in normal Javacode.
- A Servlet is a Java class. It is written like a normal Java. JSP is comes with some elements that are easy to write.

- JSP needs no compilation by the Programmer. Programmer deploys directly a JSP source code file in server where as incase of Servlets, the Programmer compiles manually a Servlet file and deploys a .class file in server.
- JSP is so easy even a Web Designer can put small interactive code (not knowing much of Java) in static Web pages.
- First time when JSP is called it is compiled to a Servlet. Subsequent calls to the same JSP will call the same compiled servlet (instead of converting the JSP to servlet), Ofcourse, the JSP code would have not modified. This increases performance.

### Anatomy of JSP





# Anatomy of a jsp page

```
<%@page contentType = "text/html" language = "java%">
<%@page import = "java.util.Date" session = "false"%>
```

Jsp elements

**%@ is jsp directive**

```
<html>
<head>
<title> simple jsp page demo</title>
</head>
<body>
<h3> current time is : </h3>
```



Template data

```
<%= new Date()%> -> jsp elements
```

**%= is jsp element**

```
</body>
```

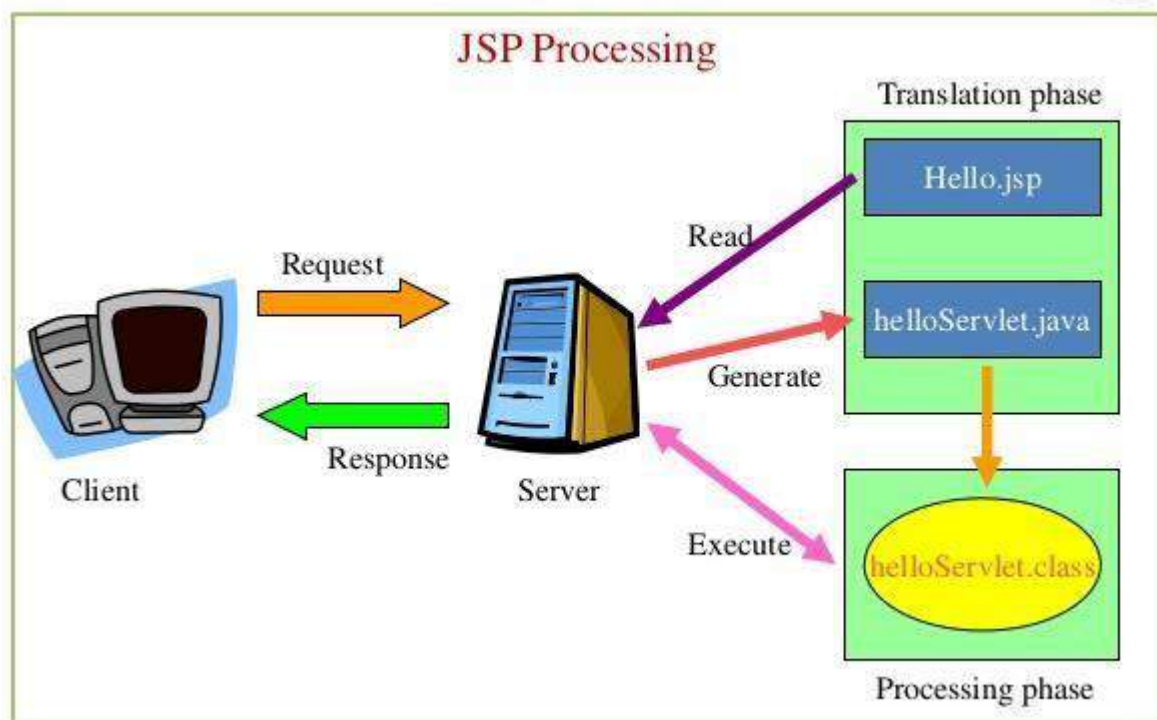
```
</html>
```

Template data

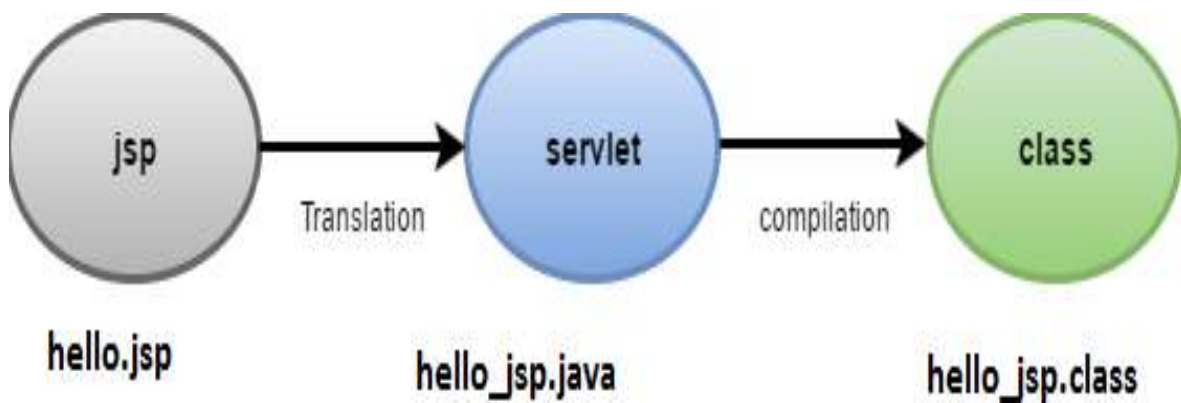
## JSP Processing

JSP Processing is done two phases:

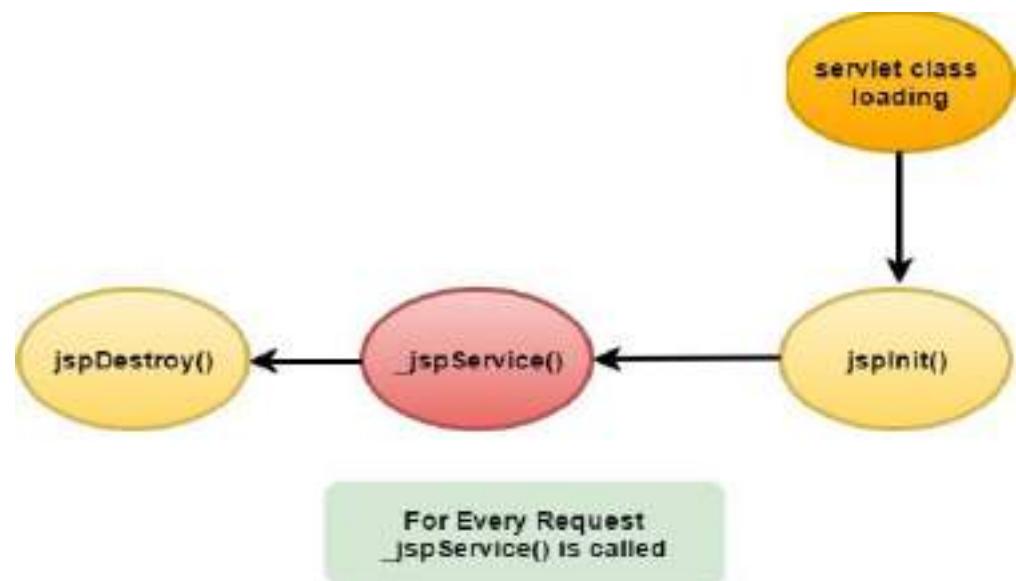
Translation Phase and Request Processing phase



- **Translation phase:** It is process of converting jsp into an equivalent Servlet and then generating class file of the Servlet.



**Request processing phase:** It is process of **executing service()** of jsp and generating response data on to the browser.



Once you have a JSP capable web-server or application server, you need to know the following information about it:

- Where to place the files
- How to access the files from your browser (with an http: prefix, not as file:)

You should be able to create a simple file, such as

```
<HTML>
<BODY>
Hello, world
</BODY> </HTML>
```

Know where to place this file and how to see it in your browser with an http:// prefix.

Since this step is different for each web-server, you would need to see the web-server documentation to find out how this is done. Once you have completed this step, proceed to the next.

**Your first JSP**

JSP simply puts Java inside HTML pages. You can take any existing HTML page and change its extension to ".jsp" instead of ".html". In fact, this is the perfect exercise for your first JSP.

Take the HTML file you used in the previous exercise. Change its extension from ".html" to ".jsp". Now load the new file, with the ".jsp" extension, in your browser.

**You will see the same output, but it will take longer! But only the first time. If you reload it again, it will load normally.**

What is happening behind the scenes is that your JSP is being turned into a Java file, compiled and loaded. This compilation only happens once, so after the first load, the file doesn't take long to load anymore. (But everytime you change the JSP file, it will be re-compiled again.)

Of course, it is not very useful to just write HTML pages with a .jsp extension! We now proceed to see what makes JSP so useful

Adding dynamic content via expressions

As we saw in the previous section, any HTML file can be turned into a JSP file by changing its extension to .jsp. Of course, what makes JSP useful is the ability to embed Java. Put the following text in a file with .jsp extension (let us call it hello.jsp), place it in your JSP directory, and view it in a browser.

```
<HTML>
<BODY>
Hello! The time is now <%= new java.util.Date() %>
</BODY>
</HTML>
```

Notice that each time you reload the page in the browser, it comes up with the current time. The character sequences

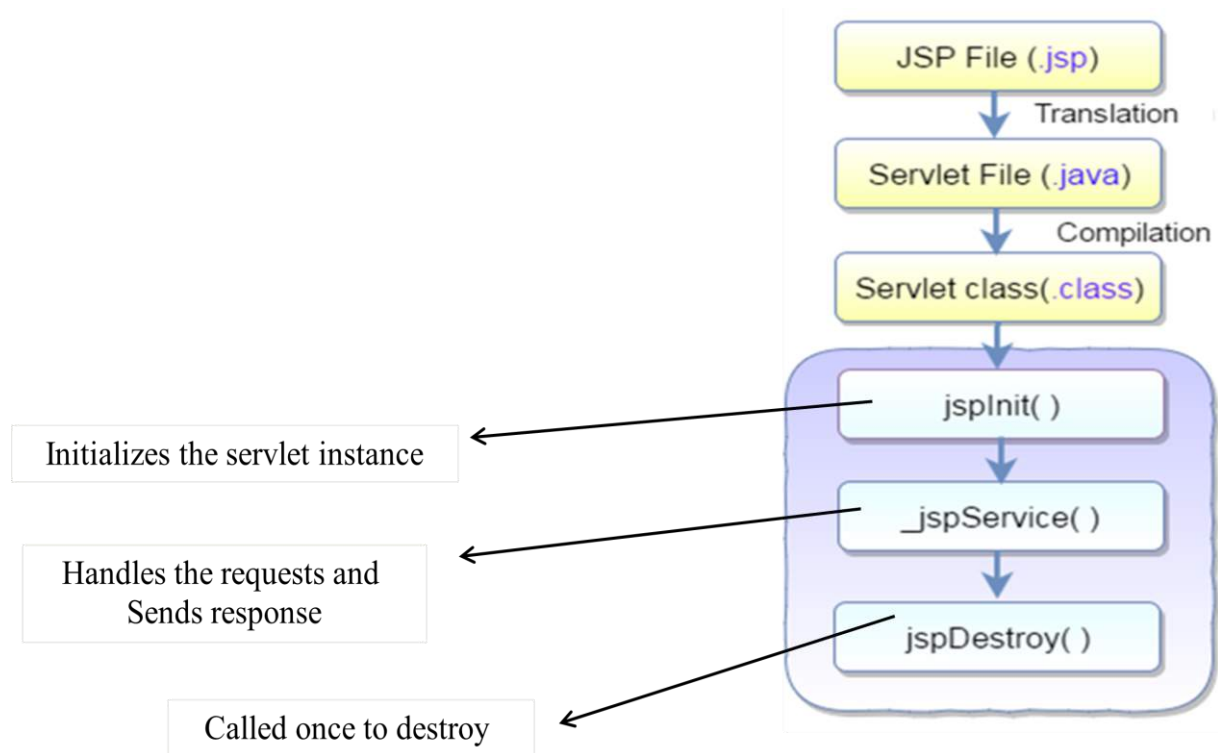
<%= and %> enclose Java expressions, which are evaluated at run time.

This is what makes it possible to use JSP to generate dynamic HTML pages that change in response to user actions or vary from user to user.

**JSP Lifecycle**

The translation of a JSP page to a Servlet is called Lifecycle of JSP.

JSP Lifecycle is exactly same as the Servlet Lifecycle, with one additional first step, which is, translation of JSP code to Servlet code.



```
public void jspInit()
{
 //code to initialize Servlet instances
}
```

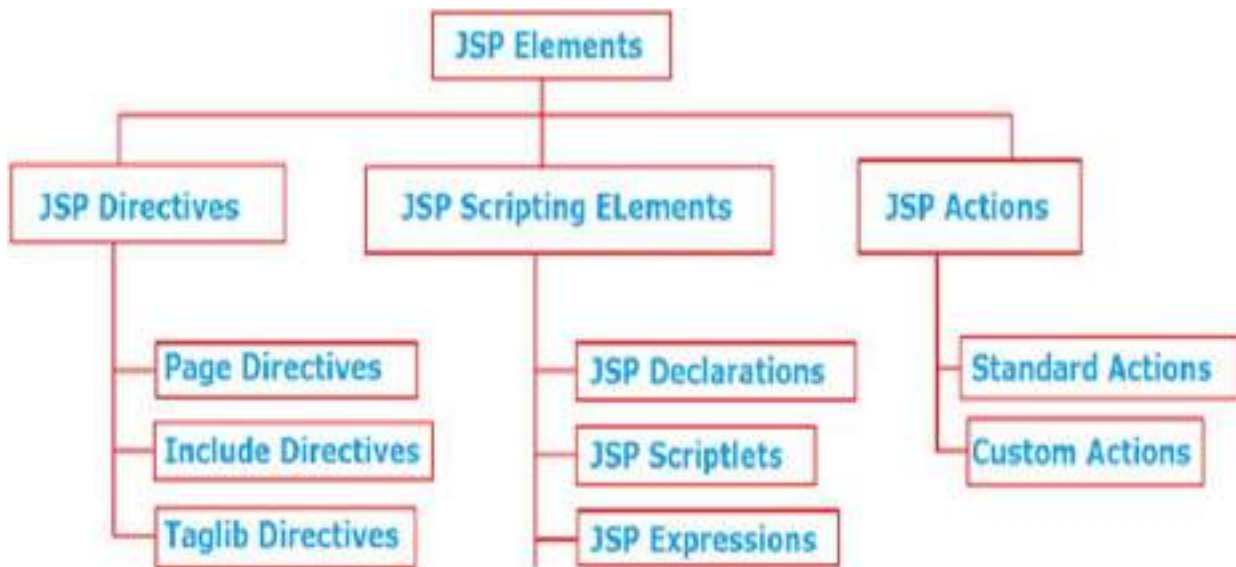
```
void _jspService(HttpServletRequest req, HttpServletResponse res)
{
 //code goes here
}
```

jspDestroy() method to destroy the instance of the servlet class

```
public void jspDestory()
{
 //code to remove the instances of servlet class
}
```

## JSP elements

JSP Scripting Elements are used for writing the Java Code inside the JSP page. There are different types of scripting elements these elements are used for various purposes



### I) JSP Directives:

JSP Directives have the mechanisms to interact with the JSP Container about translation of JSP to Servlet code and how it compiles JSP page. The entire JSP page process is controlled by this directive tags

■ **A directive has the form:**

`<%@ directive attribute="value" %>`

■ **3 types of directives**

1. **Page directive:** page is used to provide the information about it.

`<%@ page import="java.util"%>`

Ex. Write a JSP program to display date.

`<%@ page import="java.util.Date" %>`

Today is: `<%= new Date() %>`

2. **Include directive :** include is used to include a file in the JSP page.

Example: `<%@ include file="/header.jsp" %>`

```

<html>
<body>
<%@ include file="header.jsp" %>

Contact Us at: we@studytonight.com

<%@ include file="footer.jsp" %>
</body>
</html>

```

This says insert the complete content of **header.jsp** into this JSP page

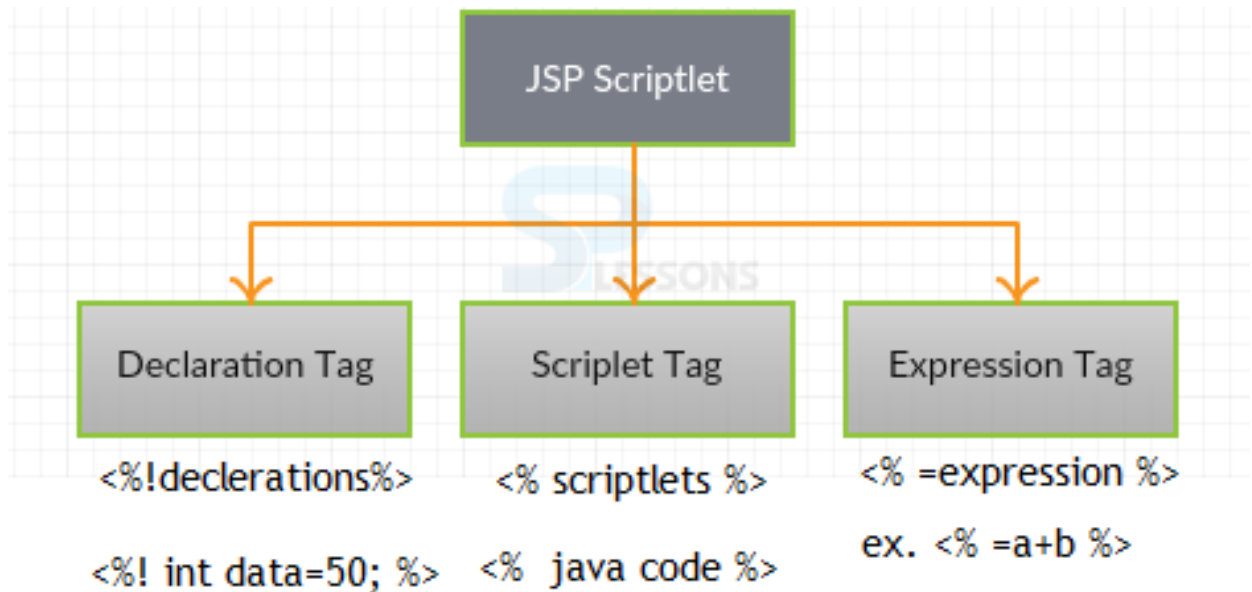
This says insert the complete content of **footer.jsp** into this JSP page

3. **taglib directive:** taglib is used to use the custom tags in the JSP pages (custom tags allows us to defined our own tags).

Example: `<%@ taglib uri="tlds/taglib.tld" prefix="mytag" %>`

## 2. JSP Scripting elements

- In JSP, script tag is used to write JAVA related code within a JSP page. There is more than one type of JSP “tag,” depending on what you want done with the Java



### 1. `<%! declarations %>`

- The JSP declaration tag is used to *declare fields and methods*

### 2. `<% code %>` scriptlet

- A scriptlet tag is used to execute java source code in JSP

### 3. `<%= expression %>`

- The *expression* is evaluated and the result is inserted into the HTML page

Ex. Declaration and expression

```

<html>
<body>
<%! int data=50; %>
<%= "Value of the variable is:"+data %>
</body>
</html>

```

Ex. scriptlet

```

<html>
<body>
<% out.print("welcome to jsp"); %>
</body>
</html>

```

## 3. Action Elements

- The action tags are used to control the flow between pages





forwards the request and response to another resource.

includes another resource.

creates or locates bean object

- The forward action terminates the action of the current page and forwards the request to another resource such as a static page, another JSP page, or a Java Servlet.
- `<jsp:forward page = "Relative URL" />`
- `<jsp:forward page = "date.jsp" />`

```

<html>
<body>
<jsp:forward page = "date.jsp" />
</body> </html>

```



```

<p>Today's date:
<%= (new java.util.Date()).toLocaleString()%>
</p>

```

### Custom actions in JSP

- Custom tags are user-defined tags. They eliminates the possibility of scriptlet tag and separates the business logic from the JSP page.
- `<prefix:tagname attr1=value1. ...attrn=valuen />`
- Ex.

```

<prefix:tagname attr1=value1. ...attrn=valuen >
body code
</prefix:tagname>

```

## Code Snippets:

### JSP program to create a String

```

<HTML>
<BODY>
<H1>Creating a String</H1>
<%
String greeting = "Hello from JSP!";
out.println(greeting);
%>
</BODY>
</HTML>

```

**Use for loop to display string array**

```
<% @ page session="false" %>
<%
String[] colors = { "red", "green", "blue" };
for (int i = 0; i < colors.length; i++) { out.print("<P>" + colors[i] + "</p>");
}
%>
```

**Creating an Array**

```
<HTML>
<BODY>
<H1>Creating an Array</H1>
<%
double accounts[];
accounts = new double[100];
accounts[3] = 119.63;
out.println("Account 3 holds " + accounts[3]);
%>
</BODY>
</HTML>
```

**Write a JSP application to print the current date and time.**

```
<h3>Current Date and Time is :</h3>
<% java.util.Date d = new java.util.Date();
out.println(d.toString()); %>
```

## JSP Implicit Objects

- These Objects are the Java objects that the JSP Container makes available to the developers in each page and the developer can call them directly without being explicitly declared.
- Implicit Objects are also called pre-defined variables.

	Object	Type
1	out	PrintWriter
2	request	HttpServletRequest
3	response	HttpServletResponse
4	config	ServletConfig
5	application	ServletContext



	tion	
6	session	HttpSession
7	pageCo ntext	PageContext
8	page	Object
9	excepti on	Throwable

## 1. The out Object

The out implicit object is an instance of a javax.servlet.PrintWriter object and is used to send content in a response.

out.print(dataType dt) -Print a data type value

out.println(dataType dt)- Print a data type value then terminate the line with new line character.

Ex.

```
<% out.print("Hello");%>
```

```
<% out.print("Today is:"+java.util.Calendar.getInstance().getTime()); %>
```

## 2. The request Object;

- The JSP request is an implicit object of type HttpServletRequest
- It can be used to get request information such as parameter, header information, remote address

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go">

</form>
```



```
<%
String name=request.getParameter("uname");
out.print("welcome "+name);
%>
```

Index.jsp

welcome.jsp

## 3. The response implicit object

- In JSP, response is an implicit object of type HttpServletResponse
- Just as the server creates the request object, it also creates an object to represent the response to the client.
- It can be used to add or manipulate response such as redirect response to another resource, send error etc.

```
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go">

</form>
```



```
<%
response.sendRedirect("http://www.google.com");
%>
```

Index.jsp

welcome.jsp

#### 4.JSP config implicit object:

- In JSP, config is an implicit object of type *ServletConfig*. This object can be used to get initialization parameter for a particular JSP page
- Generally, it is used to get initialization parameter from the web.xml file.

```
<%
out.print("Welcome "+request.getParameter("uname"));

String driver=config.getInitParameter("dname");
out.print("driver name is="+driver);
%>
```

welcome.jsp

#### 5.JSP application implicit object

- In JSP, application is an implicit object of type *ServletContext*.
- The instance of ServletContext is created only once by the web container when application or project is deployed on the server.
- This object can be used to get initialization parameter from configuration file (web.xml). It can also be used to get, set or remove attribute from the application scope.

```
<%

out.print("Welcome "+request.getParameter("uname"));

String driver=application.getInitParameter("dname");
out.print("driver name is="+driver);

%>
```

#### 6) session implicit object

- In JSP, session is an implicit object of type HttpSession. The Java developer can use this object to set, get or remove attribute or to get session information..

```
<% session.setAttribute("user",name); %>
```

## 7) pageContext implicit object

- In JSP, pageContext is an implicit object of type PageContext class. The pageContext object can be used to set, get or remove attribute from one of the following scopes:
- page
- request
- session
- application

## cookies:

- A **cookie** is a small piece of information created by a JSP program that is stored in the client's hard disk by the browser. Cookies are used to store various kind of information such as username, password, and user preferences, etc.
- **Different methods in cookie class are:**
  1. **String getName()**- Returns a name of cookie
  2. **String getValue()**-Returns a value of cookie
  3. **int getMaxAge()**-Returns a maximum age of cookie in millisecond
  4. **String getDomain()**-Returns a domain
  5. **boolean getSecure()**-Returns true if cookie is secure otherwise false
  6. **String getPath()**-Returns a path of cookie
  7. **void setPath(String)**- set the path of cookie
  8. **void setDomain(String)**-set the domain of cookie
  9. **void setMaxAge(int)**-set the maximum age of cookie
  10. **void setSecure(Boolean)**-set the secure of cookie.

### **Creating cookie:**

Cookie are created using cookie class constructor.

Content of cookies are added to the browser using addCookies() method.

### **Reading cookies:**

Reading the cookie information from the browser using get\_cookies() method.

Find the length of cookie class.

Retrieve the information using different method belongs to the cookie class

**PROGRAM: To create and read the cookie for the given cookie name as "EMPID" and its value as "AN2356".**

### **JSP program to create a cookie**

```
<%!
Cookie c=new Cookie("EMPID","AN2356");
response.addCookie(c);
%>
```

**JSP program to read a cookie**

```
<%!
Cookie c[]=request.getCookies();
for(i=0;i<c.length;i++)
{
String name=c[i].getName();
String value=c[i].getValue();
out.println("-name=" + name);
out.println("-value=" + value);
}
%>
```

**Session object(session tracking or session uses)**

- The HttpSession object associated to the request
- Session object has a session scope that is an instance of javax.servlet.http.HttpSession class. Perhaps it is the most commonly used object to manage the state contexts.
- This object persist information across multiple user connection.
- Created automatically by
- Different methods of HttpSession interface are as follows:

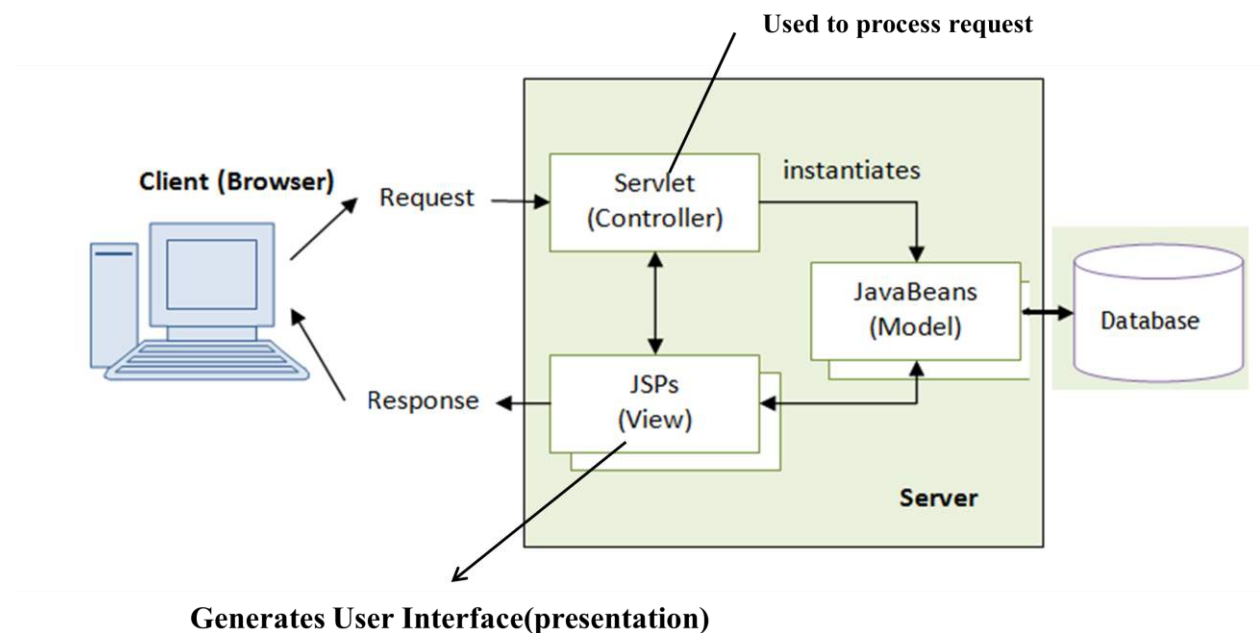
1. **object getAttribute(String)**-Returns the value associated with the name passed as argument.
2. **long getCreationTime()**-Returns the time when session created.
3. **String getID()**-Returns the session ID
4. **long getAccessedTime()**-returns the time when client last made a request for this session.
5. **void setAttribute(String,object)**-Associates the values passed in the object name passed.

**Program:**

```
<%!
HttpSession h=req.getSession(true);
Date d=(Date) h.getAttribute("Date");
out.println("-last date and time"+d);
Date d1=new Date();
d1=h.setAttribute("date",d1);
out.println("-current date and time="+d1);
%>
```

## Application design with MVC

It is an architectural pattern used to develop a web appln



M stands for Model

V stands for View

C stands for controller.

Controller acts as an interface between View and Model. Controller intercepts all the incoming requests.

Model represents the state of the application i.e. data. It can also have business logic.

View represents the presentaion i.e. UI(User Interface).

## UNIT-5

### DATABASE ACCESS & JAVA BEANS

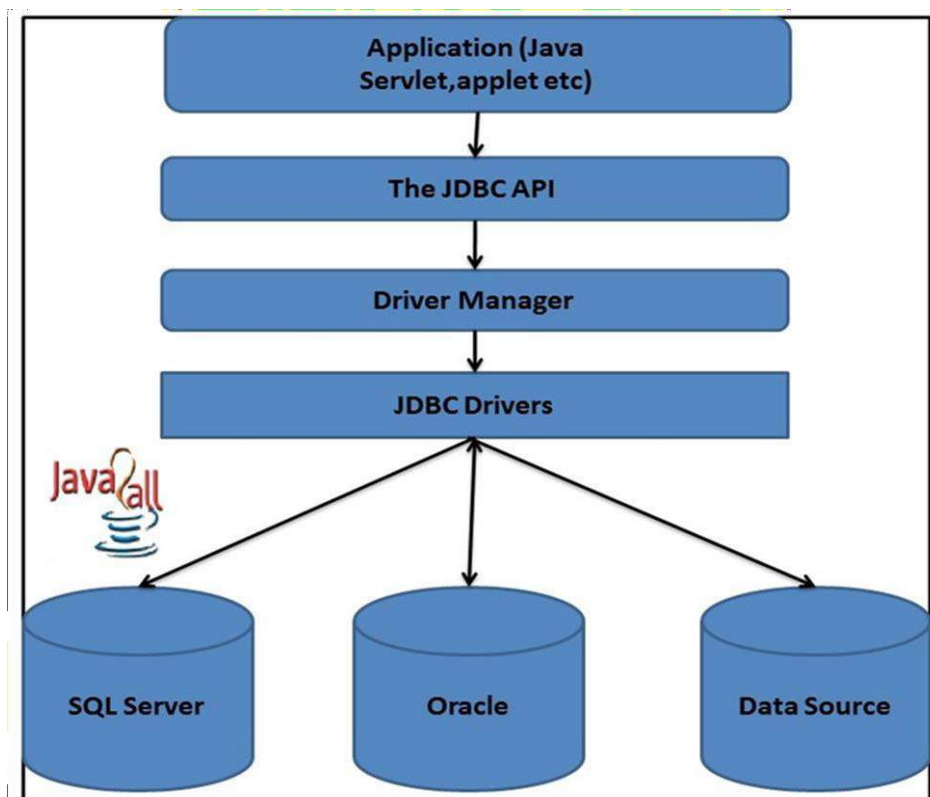
**Database Access:** Database Programming using JDBC, JDBC drivers, Studying Javax.sql.\* package, Connecting to database in PHP, Execute Simple Queries, Accessing a Database from a Servlet and JSP page.

JDBC stands for **Java Database Connectivity**. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database.



#### JDBC Driver

JDBC Driver is a software component that enables java application to interact with the database



#### JDBC drivers:

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below..

1. TYPE-1: JDBC-ODBC Bridge Driver,
2. TYPE-2: Native Driver,
3. TYPE-3: Network Protocol Driver, and

## 4. TYPE-4: Thin Driver

**Type 1: JDBC-ODBC Bridge Driver**

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.

The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

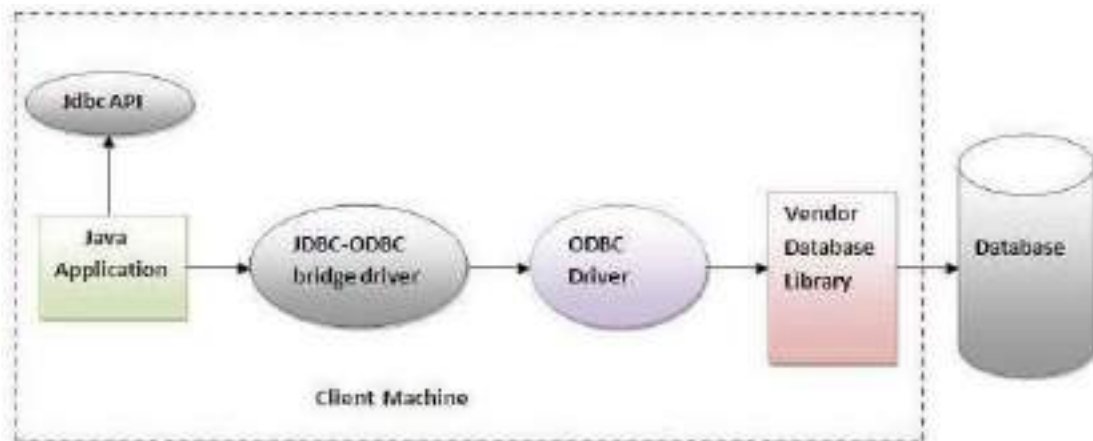


Figure- JDBC-ODBC Bridge Driver

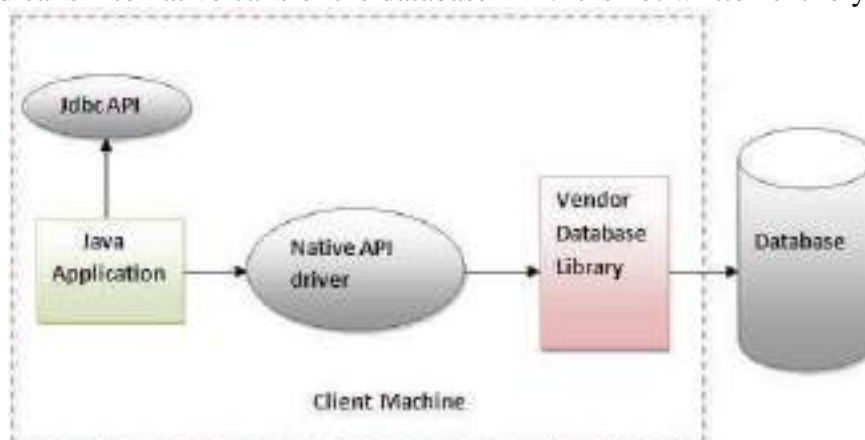
Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

**Advantages:**

- easy to use.
- can be easily connected to any database.
- Disadvantages:
  - Performance degraded because JDBC method call is converted into the ODBC function calls.
  - The ODBC driver needs to be installed on the client machine.

**Type 2: JDBC-Native API**

The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in java.



**Advantage:**

- performance upgraded than JDBC-ODBC bridge driver.

**Disadvantage:**

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

**Type 3: JDBC-Net pure Java**

In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

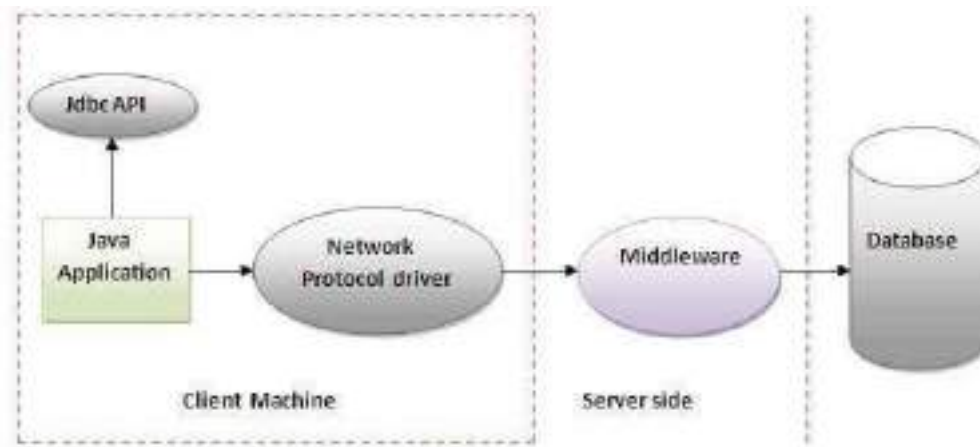


Figure- Network Protocol Driver

**Advantage:**

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

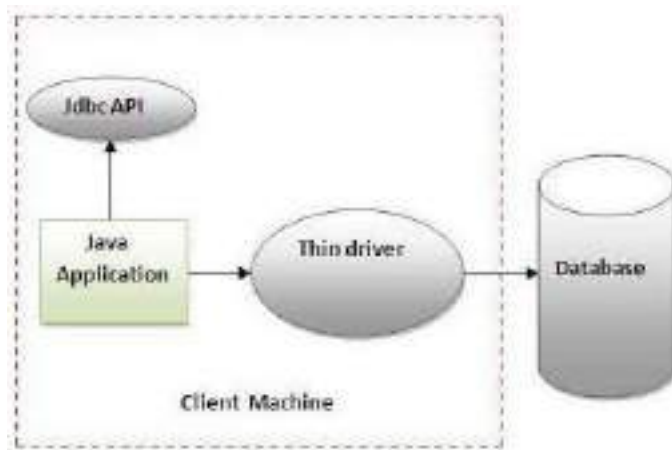
**Disadvantages:**

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

**Type 4: 100% Pure Java**

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.



**Advantage:**

- Better performance than all other drivers.
- No software is required at client side or server side.

**Disadvantage:**

- Drivers depend on the Database.

MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

**Which Driver should be Used?**

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.

If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.

Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.

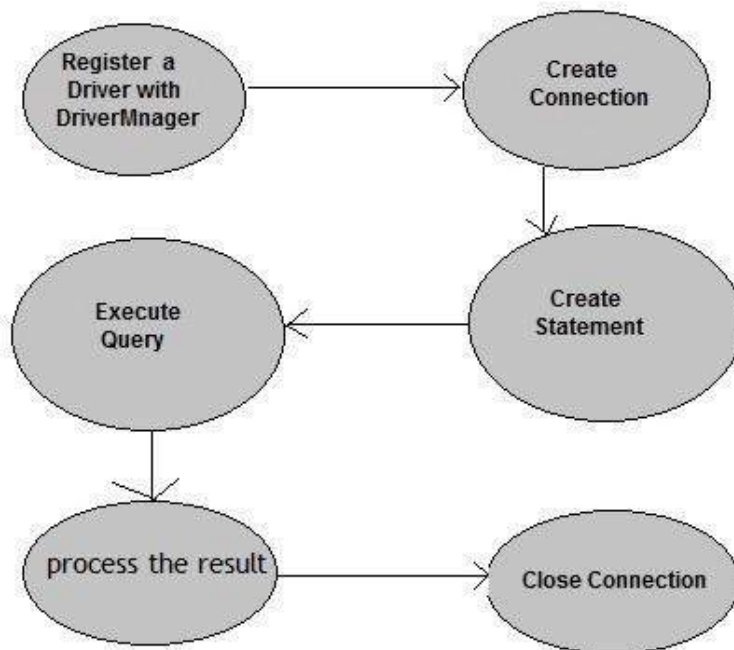
The type 1 driver is not considered a deployment- level driver, and is typically used for development and testing purposes only.

## **JDBC( Java Database Connectivity)/ JDBC Database Access**

**Steps in connecting to a Database**

There are 6 steps to connect any java application with the database using JDBC

1. Loading the driver/Register the Driver class
2. Establishing connection
3. Create statement
4. Execute queries
5. Process the result
6. Close connection



## Step 1. Register the driver class

The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax:

```
Class.forName("Driver for a Database");
```

Example

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

OR

The **registerDriver(Driver driver)** method of DriverManager class registers the given driver in the DriverManager's list.

```
Driver d=new com.mysql.jdbc.Driver();
DriverManager.registerDriver(d);
```

## Step 2. Establishing connection

The **getConnection()** method of DriverManager class is used to establish connection with the database.

Syntax: `Connection con=DriverManager.getConnection( "url","uid","pwd");`

Example `Connection con=DriverManager.getConnection(  
"jdbc:oracle:thin:@localhost:1521:xe","system","password");`

## Step 3. Create statement

The **createStatement()** method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax: `Statement stmt=con.createStatement();`

Example `Statement stmt=con.createStatement();`

## Step 4. Execute the query

The **executeQuery()** method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax: `ResultSet rs=stmt.executeQuery("select stmt");`

Example `ResultSet rs=stmt.executeQuery("select * from emp");`

Statement **executeUpdate()**(String query) is used to execute Insert/Update/Delete (DML) statements

Syntax: `ResultSet rs=stmt.executeUpdate("non select stmt");`

Example `int a=stmt.executeUpdate("delete * from table");`

## Step 5. Processing Result

The **executeQuery()** method returns the object of **ResultSet** that can be used to get all the records of a table

Ex. `ResultSet rs=stmt.executeQuery("select * from emp");`

### To read the data from ResultSet

The object of **ResultSet** maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row. Use **next()** method to read the data items from the table.

Example

```
while(rs.next()){
 System.out.println(rs.getInt(1)+" "+
 rs.getString(2));
}
```

## Step 6. Close the connection

```
con.close();
Close() method is used to close the existing connections.
```

### //PROGRAM TO CONNECT TO DB AND READ THE DATA

```
import java.sql.*;
class MysqlCon{
 public static void main(String args[])
 {
 //loading driver
 Class.forName("com.mysql.jdbc.Driver");
 //establishing connection
 Connection con=DriverManager.getConnection
 ("jdbc:mysql://localhost:3306/nkr","root","root");
 //create statement
 Statement stmt=con.createStatement();
 //executing query and processing result
 ResultSet rs=stmt.executeQuery("select * from emp");
 //read data from resultset
 while(rs.next())
 {
 System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.getString(3));
 }
 //close connection
 con.close();
 }
}
```

**JDBC Database Access**

JDBC was designed to keep simple things simple. This means that the JDBC API makes everyday database tasks, like simple SELECT statements, very easy.

**Import a package java.sql.\* :** This package provides you set of all classes that enables a network interface between the front end and back end database.

- **DriverManager** will create a Connection object.
- **java.sql.Connection** interface represents a connection with a specific database. Methods of connection is close(), createStatement(), prepareStatement(), commit(), close() and prepareCall()
- **Statement** interface used to interact with database via the execution of SQL statements. Methods of this interface are **executeQuery()**, **executeUpdate()**, **execute()** and **getResultSet()**.
- A **ResultSet** is returned when you execute an SQL statement. It maintains a pointer to a row within the tabular results. Methods of this interface are next(), getBoolean(), getByte(), getDouble(), getString(), close() and getInt().

## Examples.

**Creating JDBC Statements**

A Statement object is what sends your SQL statement to the DBMS. You simply create a Statement object and then execute it, supplying the appropriate execute method with the SQL statement you want to send. For a SELECT statement, the method to use is executeQuery. For statements that create or modify tables, the method to use is executeUpdate.

It takes an instance of an active connection to create a Statement object. In the following example, we use our Connection object con to create the Statement object stmt :

```
Statement stmt = con.createStatement();
```

At this point stmt exists, but it does not have an SQL statement to pass on to the DBMS. We need to supply that to the method we use to execute stmt.

For example, in the following code fragment, we supply executeUpdate with the SQL statement from the example above:

```
stmt.executeUpdate("CREATE TABLE STUDENT " +
"(S_NAME VARCHAR(32), S_ID INTEGER, COURSE VARCHAR2(10), YEAR
VARCHAR2(3))");
```

Since the SQL statement will not quite fit on one line on the page, we have split it into two strings concatenated by a plus sign (+) so that it will compile. Executing Statements.

Statements that create a table, alter a table, or drop a table are all examples of DDL statements and are executed with the method executeUpdate. The method executeUpdate is also used to execute SQL statements that update a table. In practice, executeUpdate is used far more often to update tables than it is to create them because a table is created once but may be updated many times.

The method used most often for executing SQL statements is executeQuery. This method is

used to execute SELECT statements, which comprise the vast majority of SQL statements.

### Entering Data into a Table

We have shown how to create the table STUDENT by specifying the names of the columns and the data types to be stored in those columns, but this only sets up the structure of the table. The table does not yet contain any data. We will enter our data into the table one row at a time, supplying the information to be stored in each column of that row. Note that the values to be inserted into the columns are listed in the same order that the columns were declared when the table was created, which is the default order.

The following code inserts one row of data,

```
Statement stmt = con.createStatement();
```

```
stmt.executeUpdate("INSERT INTO STUDENT VALUES ('xStudent', 501, =
_B.Tech','IV')");
```

Note that we use single quotation marks around the student name because it is nested within double quotation marks. For most DBMSs, the general rule is to alternate double quotation marks and single quotation marks to indicate nesting.

The code that follows inserts a second row into the table STUDENT . Note that we can just reuse the Statement object stmt rather than having to create a new one for each execution.

```
stmt.executeUpdate("INSERT INTO STUDENT " + "VALUES ('yStudent', 502,
_B.Tech','III')");
```

### Getting Data from a Table

Now that the table STUDENT has values in it, we can write a SELECT statement to access those values. The star (\*) in the following SQL statement indicates that all columns should be selected. Since there is no WHERE clause to narrow down the rows from which to select, the following SQL statement selects the whole table:

```
SQL> SELECT * FROM STUDENT;
```

### Retrieving Values from Result Sets

We now show how you send the above SELECT statements from a program written in the Java programming language and how you get the results we showed.

JDBC returns results in a ResultSet object, so we need to declare an instance of the class ResultSet to hold our results. The following code demonstrates declaring the ResultSet object rs and assigning the results of our earlier query to it:

```
ResultSet rs = stmt.executeQuery("SELECT S_NAME, YEAR FROM STUDENT");
```

The following code accesses the values stored in the current row of rs. Each time the method next is invoked, the next row becomes the current row, and the loop continues until there are no more rows in rs.

```
String query = "SELECT COF_NAME, PRICE FROM STUDENT";
```

```
ResultSet rs = stmt.executeQuery(query);
```

```
while (rs.next())
```

```
{
```

```
String s = rs.getString("S_N
AME"); Integer i =
```

```
rs.getInt("S_ID");
String c = rs.getString("COURSE");
String y = rs.getString("YEAR");
System.out.println(i + " " + s + " " + c + " " + y);
}
```

### Updating Tables

Suppose that after a period of time we want update the YEAR column in the table STUDENT. The SQL statement to update one row might look like this:

```
String updateString = "UPDATE STUDENT " +
"SET YEAR = IV WHERE S-NAME LIKE 'yStudent'";
```

Using the Statement object stmt , this JDBC code executes the SQL statement contained in updateString :

```
stmt.executeUpdate(updateString);
```

### Using try and catch Blocks:

Something else all the sample applications include is try and catch blocks. These are the Java programming language's mechanism for handling exceptions. Java requires that when a method throws an exception, there be some mechanism to handle it. Generally a catch block will catch the exception and specify what happens (which you may choose to be nothing). In the sample code, we use two try blocks and two catch blocks. The first try block contains the method Class.forName, from the java.lang package. This method throws a ClassNotFoundException, so the catch block immediately following it deals with that exception. The second try block contains JDBC methods, which all throw SQLExceptions, so one catch block at the end of the application can handle all of the rest of the exceptions that might be thrown because they will all be SQLException objects.

### Retrieving Exceptions

JDBC lets you see the warnings and exceptions generated by your DBMS and by the Java compiler. To see exceptions, you can have a catch block print them out. For example, the following two catch blocks from the sample code print out a message explaining the exception:

#### Try

```
{
// Code that could generate an exception goes here.
// If an exception is generated, the catch block below
// will print out information about it.
} catch(SQLException ex)
{
System.err.println("SQL Exception: " + ex.getMessage());
}
```

## Accessing a Database from a Servlet :

To start with interfacing Java Servlet Program with JDBC Connection: Proper JDBC Environment should set-up along with database creation

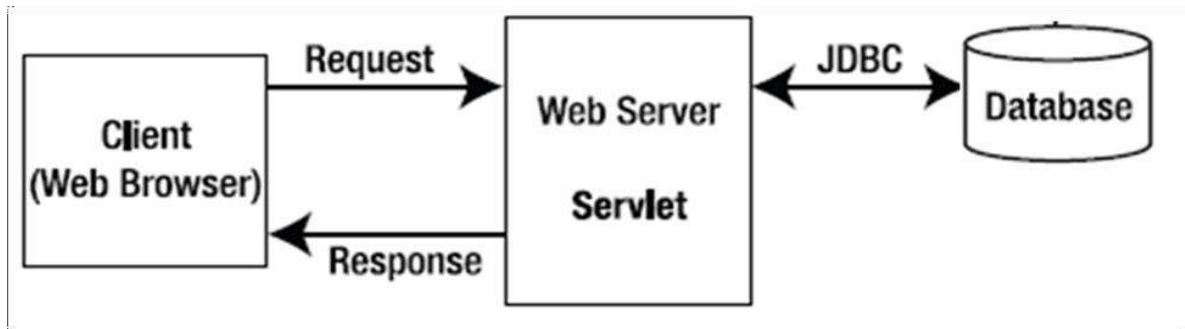


Fig. Connecting from a servlet to DB

## Accessing a Database from a Servlet

### Step1. Setup the JDBC environment for server

Download ojdbc14.jar for Oracle Database

mysqlconnector.jar for mysql database.

Copy the above in tomcat/LIB folder of server

### Step2. Create a servlet class with JDBC

```
import java.sql.*;
```

```
import java.servlet.*;
```

```
Import java.servlet.http.*;
```

```
public class ServletDatabaseDemo extends HttpServlet
```

```
{
```

```
 protected void doPost(HttpServletRequest req, HttpServletResponse res)throws
```

```
ServletException,IOException
```

```
{
```

```
 PrintWriter pw = res.getWriter();
```

```
 res.setContentType("text/html");
```

```
 Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
 Connection con =
```

```
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","scott","tiger");
```

```
Statement st=con.createStatement();
```

### Accessing a Database from a Servlet

```
ResultSet rs = st.executeQuery("Select * from STUDENT);
```

```
 pw.println("<table>");
```

```
while(rs.next())
```

```
{
```

```
 pw.println("<tr><td>"+rs.getInt(1)+"</td>
```

```
Dept. of CSE, MRCET <td>"+rs.getString(2)+"</td>
```

```
<td>"+rs.getString(3)+"</td>
```



```

 <td>"+rs.getString(4)+"</td></tr>");
 }
 pw.println("</table>");
 con.close();
}
}

```

### Step 3. Compile and configure in web.xml

```

<web-app>
 <servlet>
 <servlet-name>ServletDBConnection</servlet-name>
 <servlet-class>ServletDatabaseDemo</servlet-class>
 </servlet>
 <servlet-mapping>
 <servlet-name>ServletDBConnection</servlet-name>
 <url-pattern>/showrecord</url-pattern>
 </servlet-mapping>
</web-app>

```

### Step 4. Run the servlet

<http://localhost:8080/myproject/showrecord>

## Accessing a Database from a JSP

To access DB from a JSP the following steps are required

### Step1. Setup the JDBC environment for server

Download ojdbc14.jar for Oracle Database

mysqlconnector.jar for mysql database.

Copy the above in tomcat/LIB folder of server

### Step2. Create a JSP page with JDBC

```

<% @ page import = "java.io.*,java.util.*,java.sql.*"%>
<% @ page import = "javax.servlet.http.*,javax.servlet.*" %>
<% code for connection to db
 code to read the data from db
%>

```

### Step 4. Run the JSP

<http://localhost:8080/myproject/mydb.jsp>

## Accessing a Database from a php

With PHP, we can connect to and manipulate databases.

MySQL is the most popular database system used with PHP



In PHP

Dept. of CSE, MRCET  
 The `mysql_connect()` function is used to connect with MySQL database

In PHP

mysqli\_close() function is used to disconnect with MySQL database

Connecting to a Database in PHP

In PHP mysqli\_connect() function is used to connect with MySQL database

**Syntax:**

**mysqli\_connect (server, username, password)**

```
<?php
$servername = "localhost";
$username = "username";
$password = "password";

// Create connection
$conn = mysqli_connect ($servername, $username, $password);

// Check connection
if(! $conn)
{
 die("Connection failed" . mysql_error());
}
echo "Connected successfully";
?>
```

### **Executing simple Queries**

PHP uses mysqli\_query( ) to execute a query.

CREATING A DATABASE

```
$sql = 'CREATE DATABASE mydb';
$retval = mysqli_query($sql, $conn);

<?php
$servername = "localhost";
$username = "username";
$password = "password";

// Create connection
$conn = mysqli_connect ($servername, $username, $password);
```

### **Creating a Database**

```
$sql = 'CREATE Database mydb';
if(mysqli_query($conn,$sql)){
 echo "Database mydb created successfully.";
}
else{
 echo "Sorry, database creation failed ".mysql_error($conn);
}

?>
```

### **PHP MySQL Creating a Table**

PHP uses mysqli\_query( ) is used create DB Table

```
$sql = 'CREATE Table..';
$retval = mysqli_query($sql, $conn);
```

```

<?php
$servername = "localhost";
$username = "username";
$password = "password";
// Create connection
$conn = mysqli_connect ($servername, $username, $password);

$sql = "create table emp5(id INT AUTO_INCREMENT,name VARCHAR(20) NOT NULL,
emp_salary INT NOT NULL,primary key (id))";
if(mysqli_query($conn, $sql)){
 echo "Table emp5 created successfully";
}else{
 echo "Could not create table: ". mysqli_error($conn);
}
?>

```

### PHP MySQL Insert Record

PHP uses mysqli\_query( ) to insert a record into DB

```

$sql = 'INSERT INTO emp4(name,salary) VALUES ("nkr", 9000)';
$retval = mysqli_query($sql, $conn);

```

```

<?php
$servername = "localhost";
$username = "username";
$password = "password";

// Create connection
$conn = mysqli_connect ($servername, $username, $password);
$sql = 'INSERT INTO emp4(name,salary) VALUES ("nkr", 9000)';
if(mysqli_query($conn, $sql)){
 echo "Record inserted successfully";
}else{
 echo "Could not insert record: ". mysqli_error($conn);
} ?>

```

### PHP MySQLi Select Query

PHP mysqli\_query() function is used to execute select query

There are two other MySQLi functions used in select query.

mysqli\_num\_rows(mysqli\_result \$result): returns number of rows.

mysqli\_fetch\_assoc(mysqli\_result \$result): returns row as an associative array. Each key of the array represents the column name of the table. It return NULL if there are no more rows.

```

<?php
$sql = 'SELECT * FROM emp';
$retval=mysqli_query($conn, $sql);

```

```

if(mysqli_num_rows($retval) > 0)

```

```
 {
 while($row = mysqli_fetch_assoc($retval))
 {
 echo "EMP ID :{$row['id']}
 ".
 "EMP NAME : {$row['name']}
 ".
 "EMP SALARY : {$row['salary']}
 ".
 "-----
";
 } //end of while
 }else{
 echo "0 results";
 }
 ?>
```

# **WEB TECHNOLOGIES**

## **Assignment-II**

1. Define Servlet. Explain the life cycle methods of a Servlet and Write a program by using Servlet.
2. Define JDBC. Explain JDBC Drivers with a neat diagram.
3. Explain accessing a database from a Servlet with example.
4. Illustrate connection to MySQL database from a PHP program.
5. List out the implicit objects in JSP. Explain about each one with an example?