DIGITAL NOTES
ON
SOFTWARE ENGINEERING

B.TECH II YEAR-I
SEM (2023-24)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MALLAREDDY COLLEGE OF ENGINEERING & TECHNOLOGY
(Autonomous Institution–UGC, Govt. of India)

(Affiliated to JNTUH ,Hyderabad ,Approved by AICTE –Accredited by NBA&NAAC –'A'Grade-ISO9001:2015 Certified) Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, INDIA.

MALLAREDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

II Year B.Tech.CSE- I Sem **L/T/P/C 3/-/-/3**

(R22A0505) SOFTWARE ENGINEERING

COURSE OBJECTIVES

- The aim of the course is to provide an understanding of the working knowledge of the techniques to understand Software development as a process.
- Various software process models and system models.
- Various software designs, Architectural, object oriented, user interface etc.
- Software testing methodologies overview: various testing techniques including white box testing black box testing regression testing etc.
- Software quality: metrics, risk management quality assurance etc.

UNIT-I

Introduction to Software Engineering: The evolving role of software, changing nature of software, software myths.

A Generic view of process: Software engineering-a layered technology, a process framework, the capability maturity model integration(CMMI).

Process models: The waterfall model, Spiral model and Agile methodology

UNIT -II

Software Requirements: Functional and non- functional requirements, user requirements, system requirements, interface specification, the software requirements document.

Requirements engineering process: Feasibility studies, requirements elicitation and analysis, requirements validation, requirements management.

UNIT-III

Design Engineering: Design process and design quality, design concepts, the design model. Creating an architectural design: software architecture, data design, architectural styles and patterns, architectural design, conceptual model of UML, basic structural modeling, class diagrams, sequence diagrams, collaboration diagrams, usecase diagrams, component diagrams.

UNIT-IV

Testing Strategies: A strategic approach to software testing, test strategies for conventional software, black-box and white-box testing, validation testing, system testing, the art of debugging.

Metrics for Process and Products: Software measurement, metrics for software quality.

UNIT-V

Risk management: Reactive Vs proactive risk strategies, software risks, risk identification, risk projection, risk refinement, RMMM.

Quality Management: Quality concepts, software quality assurance, software reviews, formal technical reviews, statistical software quality assurance, software reliability, the ISO 9000qualitystandards.

.

TEXTBOOKS:

1. Software Engineering A practitioner's Approach, RogerS Pressman,6thedition. McGraw Hill International Edition.
2. Software Engineering, IanSommerville,7th edition, Pearson education.

Course Outcomes:

- Understand software development life cycle Ability to translate end-user requirements into system and software requirements.
- Structure the requirements in a Software Requirements Document and Analyze Apply various process models for a project, Prepare SRS document for a project
- Identify and apply appropriate software architectures and patterns to carry out high level design of a system and be able to critically compare alternative choices.
- Understand requirement and Design engineering process for a project and Identify different principles to create an user interface

- Identify different testing methods and metrics in a software engineering project and Will have experience and/or awareness of testing problems and will be able to develop a simple testing report

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF CSE

INDEX

14	III	Design concepts	25
15	III	The design model	27
16	III	Creating an architectural design: software architecture, data design	28
17	III	A Conceptual Model of the UML	31
18	III	Basic structural modeling, class diagrams, sequence diagrams, collaboration diagrams, use case diagrams, component diagrams	53
19	IV	Testing Strategies : A strategic approach to software testing ,test strategies for conventional software	54
20	IV	Black-Box and White-Box testing	58
21	IV	Validation testing, System testing	61
22	IV	The art of Debugging	62
23	IV	Product metrics: Software Quality, Metrics for Analysis Model, Metrics for Design Model, Metrics for source code, Metrics for testing, Metrics for maintenance.	64
24	V	Risk management: Reactive Vs proactive risk strategies, software risks,	67
25	V	Identification, Risk projection,	68
26	V	Risk refinement, RMMM	69
27	V	Quality Management: Quality concepts, software quality assurance	65
28	V	Software reviews, formal technical reviews	71
29	V	Statistical software quality assurance,.	72

30	V	Software reliability, the ISO 9000qualitystandards	73

UNIT- I

Introduction to Software Engineering: The evolving role of software, changing nature of software, software myths.

A Generic view of process: Software engineering-a layered technology, a process framework, the capability maturity model integration (CMMI).

Process models: The waterfall model, Spiral model and Agile methodology

---

INTRODUCTION:

Software Engineering is a framework for building software and is an engineering approach to software development. Software programs can be developed without S/E principles and methodologies but they are indispensable if we want to achieve good quality software in a cost effective manner.

Software is defined as:

Instructions + Data Structures + Documents

Engineering is the branch of science and technology concerned with the design, building, and use of engines, machines, and structures. It is the application of science, tools and methods to find cost effective solution to simple and complex problems.

SOFTWARE ENGINEERING is defined as a systematic, disciplined and quantifiable approach for the development, operation and maintenance of software.

The Evolving role of software

The dual role of Software is as follows:

1. A Product-Information transformer producing, managing and displaying information.
2. A Vehicle for delivering a product-Control of computer(operating system),the communication of information(networks) and the creation of other programs.

Characteristics of software

- Software is developed or engineered, but it is not manufactured in the classical sense.
- Software does not wear out, but it deteriorates due to change.
- Software is custom built rather than assembling existing components.

THECHANGINGNATUREOFSOFTWARE

- The various categories of software are
- System software
- Application software
- Engineering and scientific software
- Embedded software
- Product-line software
- Web-applications
- Artificial intelligence software

- **System software.** System software is a collection of programs written to service other programs
- **Embedded software**–resides in read-only memory and is used to control products and systems for the consumer and industrial markets.
- **Artificial intelligence software.** Artificial intelligence (AI) software makes use of non numeric algorithms to solve complex problems that are not amenable to computation or straightforward analysis
- **Engineeringandscientificsoftware.**Engineeringandscientificsoftwarehavebeencharact erized

by "number crunching" algorithms.

LEGACYSOFTWARE

Legacy software is older programs that are developed decades ago. The quality of legacy software is poor because it has in extensible design, convoluted code, poor and nonexistent documentation, test cases and results that are not achieved.
As time passes legacy systems evolve due to following reasons:

- Thesoftwaremustbeadaptedtomeettheneedsofnewcomputingenvironmentortechnology.
- The software must be enhanced to implement new business requirements.
- Thesoftwaremustbeextendedtomakeitinteroperablewithmoremodernsystemsordatabase
- The software must be re architected to make it viable within a network environment.

SOFTWAREMYTHS

Myths are widely held but false beliefs and views which propagate misinformation and confusion. Three types of myth are associated with software:

- Management myth
- Customer myth
- Practitioner's myth

MANAGEMENTMYTHS

- Myth (1)-The available standards and procedures for software are enough.
- Myth (2)-Each organization feel that they have state-of-art software development tools since they have latest computer.
- Myth (3)-Adding more programmers when the work is behind schedule can catch up.
- Myth(4)-Outsourcing the software project to third party , we can relax and let that party build it.

CUSTOMERMYTHS

- Myth (1)-General statement of objective is enough to begin writing programs, the details can be filled in later.
- Myth(2)-Software is easy to change because software is flexible

PRACTITIONER'SMYTH

- Myth (1)-Once the program is written ,the job has been done.
- Myth (2)-Until the program is running , there is no way of assessing the quality.
- Myth(3)-The only deliverable work product is the working program
- Myth (4)-Software Engineering creates voluminous and unnecessary documentation and invariably slows down software development.

SOFTWAREENGINEERING-ALAYEREDTECHNOLOGY

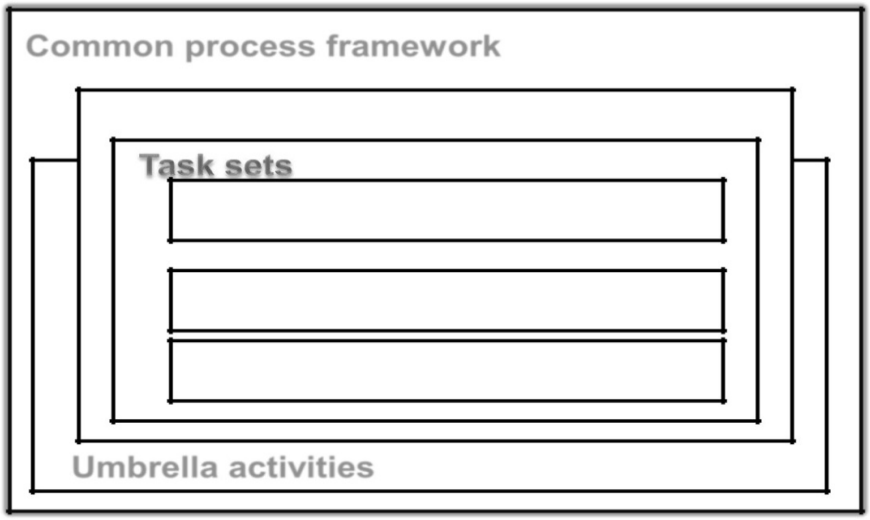
Fig: Software Engineering-A layered technology

SOFTWAREENGINEERING-ALAYEREDTECHNOLOGY

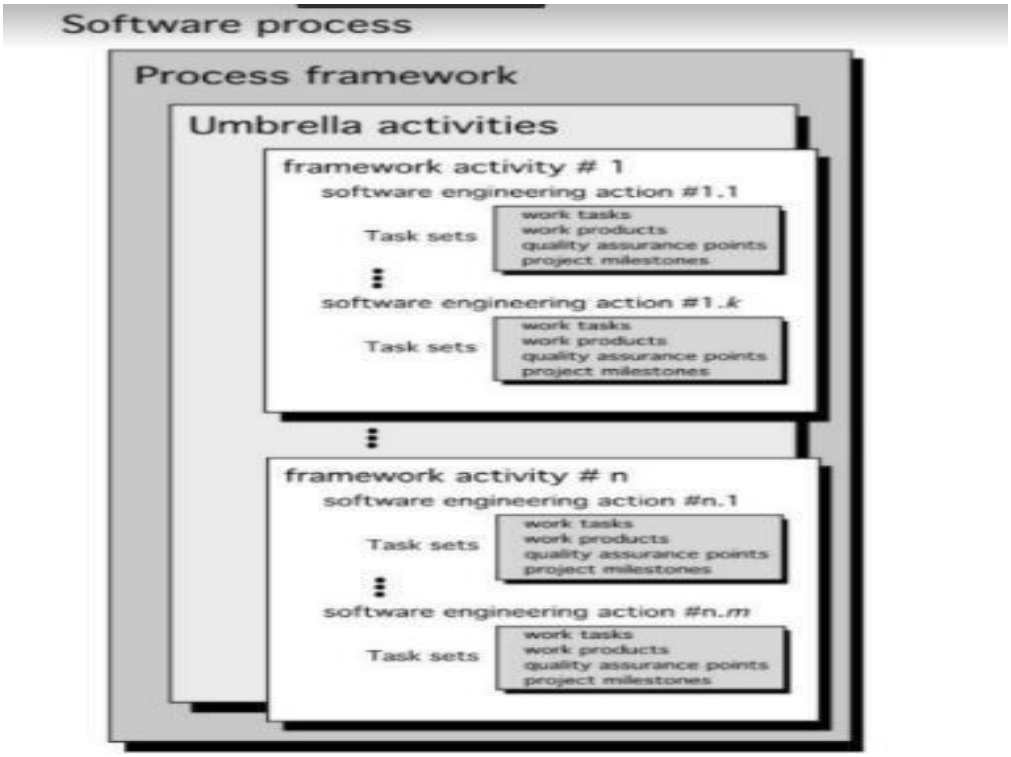
- Quality focus –Bedrock that supports Software Engineering.
- Process-Foundation for software Engineering
- Methods-Provide technical How-to's for building software
- Tools-Provide semi-automatic and automatic support to methods

APROCESSFRAMEWORK

- Establishes the foundation for a complete software process
- Identifies a number of frame work activities applicable to all software projects
- Also include a setofumbrellaactivitiesthatareapplicableacrosstheentiresoftwareprocess.



A PROCESS FRAME WORK comprises of: Common process frame work Umbrella activities Framework activities Tasks, Milestones, deliverables SQA points



A PROCESS FRAME WORK

Used as a basis for the description of process models Generic process activities

- Communication
- Planning
- Modeling
- Construction
- Deployment

APROCESSFRAMEWORK

Genericviewofengineeringcomplimentedbyanumberofumbrellaactivities

- Software project tracking and control
- Formal technical reviews
- Software quality assurance
- Software configuration management
- Document preparation and production
- Reusability management
- Measurement
- Risk management

CAPABILITY MATURITY MODEL INTEGRATION (CMMI)

- Developed by SEI (Software Engineering institute)
- Assess the process model followed by an organization and rate the organization with different levels
- A set of software engineering capabilities should be present as organizations reach

different levels of process capability and maturity.

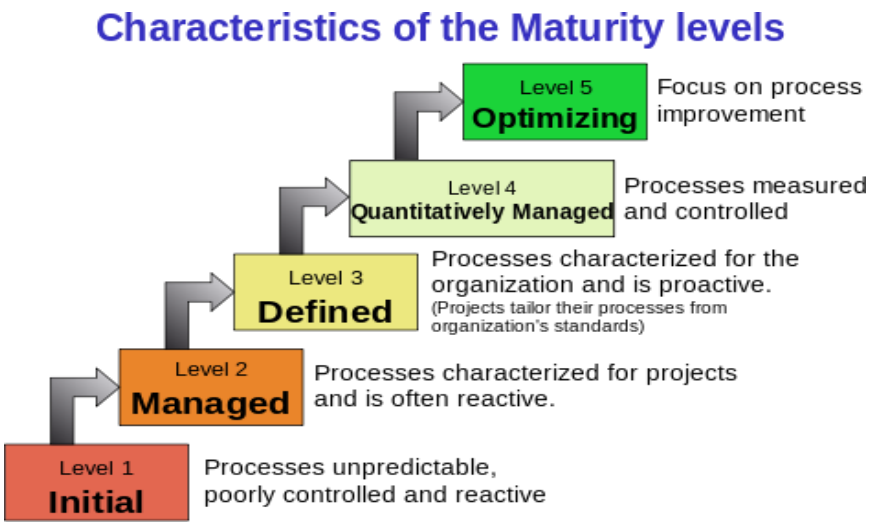
CMMI process Meta model can be represented in different ways
1. A continuous model
2. A staged model

Continuous model:

-Lets organizations elect specific improvement that best meet its business objectives and minimize risk- Levels are called capability levels.
-Describes a processin2 dimensions
-Each process area is assessed against specific goals and practices and israted according to the following capability levels.

CMMI

- Six levels of CMMI
  - Level0:Incomplete
  - Level1:Performed
  - Level2:Managed
  - Level3:Defined
  - Level4:Quantitativelymanaged
  - Level5:Optimized



CMMI

- Incomplete-Process is adhoc. Objective and goal of process areas are not known
- Performed-Goal, objective, work tasks, work products and other activities of software process are carried out
- Managed-Activities are monitored, reviewed, evaluated and controlled
- Defined-Activities are standardized, integrated and documented

- Quantitatively Managed-Metrics and indicators are available to measure the process and quality
- Optimized-Continuous process improvement based on quantitative feedback from the user

-Use of innovative ideas and techniques, statistical quality control and other methods for process improvement.

CMMI-Staged model

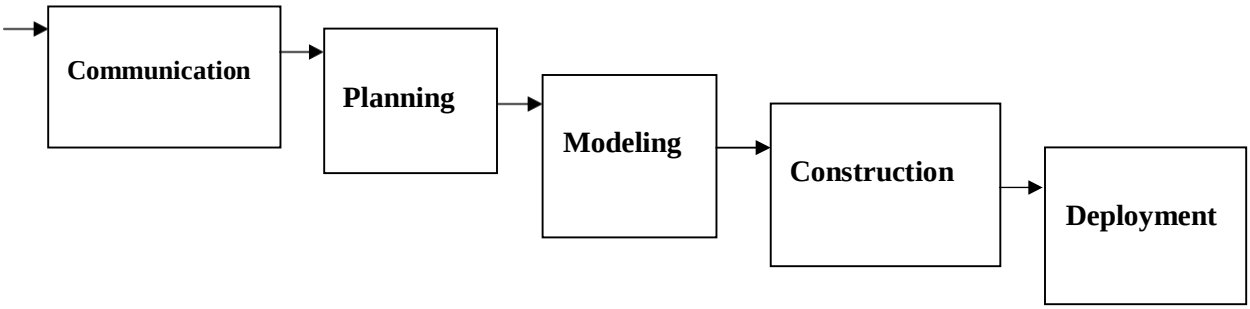
- This model is used if you have no clue of how to improve the process for quality software.
- It gives a suggestion of what things other organizations have found helpful to work first
- Levels are called maturity levels

PROCESS MODELS

- Help in the software development
- Guide the software team through a set of frame work activities
- Process Models may be linear, incremental or evolutionary

THE WATERFALL MODEL

- Used when requirements are well understood in the beginning
- Also called classic life cycle
- A systematic, sequential approach to Software development
- Begins with customer specification of Requirements and progresses through planning, modeling, construction and deployment.



This Model suggests a systematic, sequential approach to SW development that begins at the system level and progresses through analysis, design, code and testing

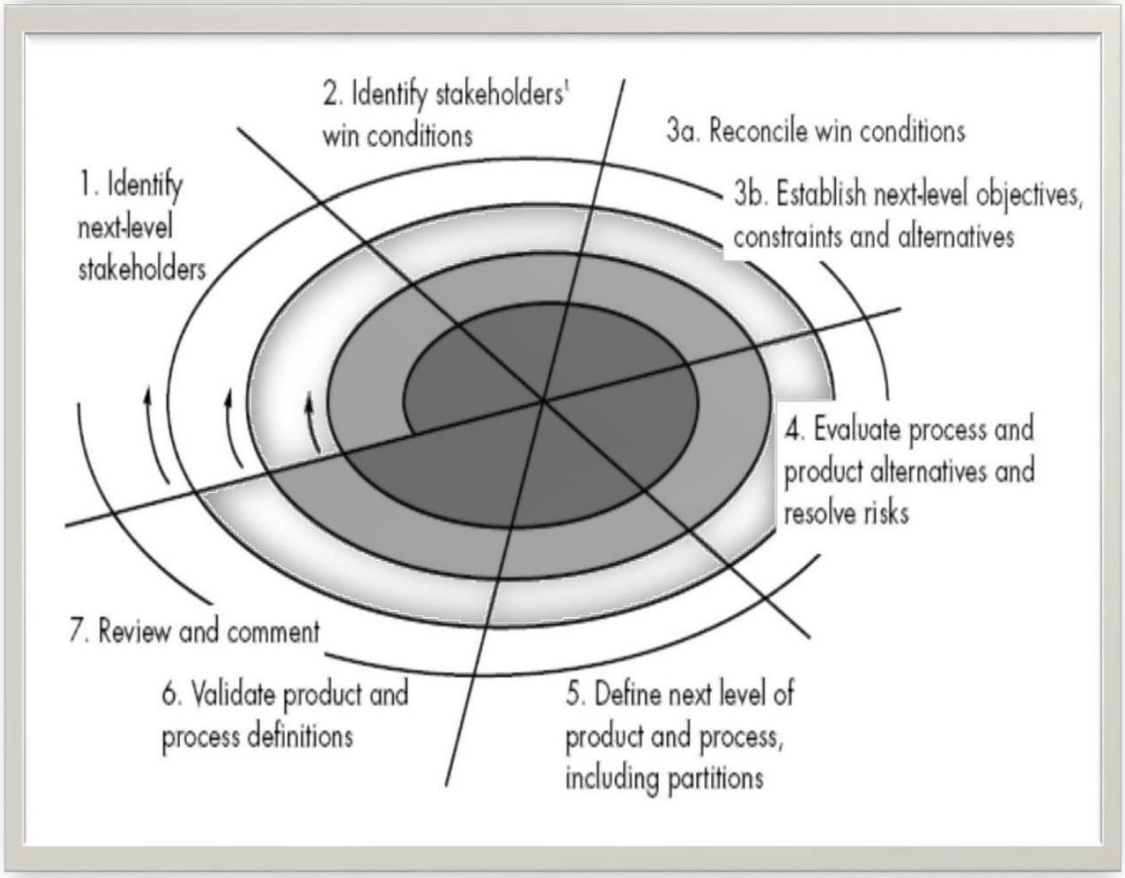
PROBLEMS IN WATER FALL MODEL

- Real projects rarely follow these sequential flow since they are always iterative
- The model requires requirements to be explicitly spelled out in the beginning, which is often difficult

- A working model is not available until late in the project time plan.

THE SPIRAL MODEL

An evolutionary model which combines the best feature of the classical lifecycle and The iterative nature of prototype model. Include new element: Risk element. Starts in middle and continually visits the basic tasks of communication, planning, modeling, construction and deployment



THE SPIRAL MODEL

- Realistic approach to the development of large scale system and software
- Software evolves as process progresses
- Better understanding between developer and customer
- The first circuit might result in the development of a product specification
- Sub sequent circuits develop a prototype
- And sophisticated version of software

THE CONCURRENT DEVELOPMENT MODEL

- Also called concurrent engineering
- Constitutes a series of framework activities ,software engineering action ,tasks and their associated states
- All activities exist concurrently but reside in different states
- Applicable to all types of software development
- Event generated at one point in the process trigger transitions among the states

A FINAL COMMENT ON EVOLUTIONARY PROCESS

- Difficult in project planning
- Speed of evolution is not known does not focus on flexibility and extensibility (more emphasis on high quality)
- Requirement is balance between high quality and flexibility and extensibility

Agility and Agile Process model

The meaning of Agile is swift or versatile."**Agile process model**" refers to a software development approach based on iterative development. Agile methods break tasks into smaller iterations, or parts do not directly involve long term planning. The project scope and requirements are laid down at the beginning of the development process. Plans regarding the number of iterations, the duration and the scope of each iteration are clearly defined in advance

Fig. Agile Model

Phases of Agile model:

1. Requirements gathering
2. Design the requirements

3. Construction/iteration
4. Testing/Quality assurance
5. Deployment
6. Feedback

1. **Requirements gathering:** In this phase, you must define the requirements. You should explain business opportunities and plan the time and effort needed to build the project. Based on this information, you can evaluate technical and economic feasibility.

2. **Design the requirements:** When you have identified the project, work with stakeholders to define requirements. You can use the user flow diagram or the high-level UML diagram to show the work of new features and show how it will apply to your existing system.

3. **Construction/ iteration:** When the team defines the requirements, the work begins. Designers and developers start working on their project, which aims to deploy a working product. The product will undergo various stages of improvement, so it includes simple, minimal functionality.

4. **Testing: In** this phase, the Quality Assurance team examines the product's performance and looks for the bug.

5. **Deployment:** In this phase, the team issues a product for the user's work environment.

6. Feedback: After releasing the product, the last step is feedback. In this, the team receives feedback about the product and works through the feedback.

Advantages:

1. Frequent Delivery
2. Face-to-Face Communication with clients.
3. Efficient design and fulfils the business requirement.
4. Any time changes are acceptable.
5. It reduces total development time.

Disadvantages:

1. Due to the shortage of formal documents, it creates confusion and crucial decisions taken throughout various phases can be misinterpreted at any time by different team members.
2. Due to the lack of proper documentation, once the project completes and the developers allotted to another project, maintenance of the finished project can

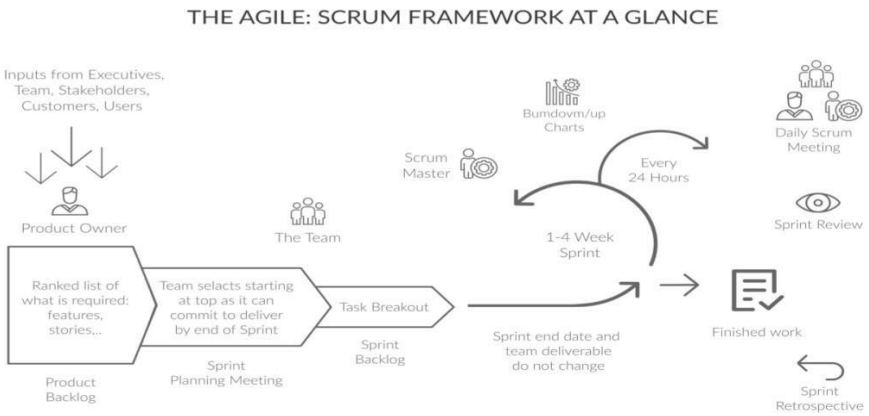
become a difficulty.

Other process models of Agile Development and Tools

- Crystal
- Scrum

Scrum

Scrum is aimed at sustaining strong collaboration between people working on complex products, and details are being changed or added. It is based upon the systematic interactions between the three major roles: Scrum Master, Product Owner, and the Team.



- **Scrum Master** is a central figure within a project. His principal responsibility is to eliminate all the obstacles that might prevent the team from working efficiently.
- **Product Owner**, usually a customer or other stakeholder, is actively involved throughout the project ,conveying the global vision of the product and providing timely feedback on the job done after every sprint.
- **Scrum Team** is a cross-functional and self-organizing group of people that is responsible for the product implementation. It should consist of up to 7 team members, in order to stay flexible and productive.

Crystal

Crystal is an agile methodology for software development. It places focus on people over processes, to empower teams to find their own solutions for each project rather than being constricted with rigid methodologies.

Crystal methods focus on:-

- People involved
- Interaction between the teams/Community
- Skills of people involved Their Talents

UNIT-II

SOFTWAREREQUIREMENTS: Functional and non- functional requirements, user requirements, system requirements, interface specification, the software requirements document.

Requirements engineering process: Feasibility studies, requirements elicitation and analysis, requirements validation, requirements management.

Software Requirements:

IEEE defines Requirements:

1. A condition or capability needed by a user to solve a problem or achieve an objective
2. A condition or capability that must be met or possessed by a system or a system component to satisfy construct, standard, specification or formally imposed document
3. A documented representation of a condition nor capability as in 1 or 2

SOFTWARE REQUIREMENTS

- Encompasses both the User's view of the requirements(the external view)and the Developer's view(inside characteristics)

User's Requirements

--Statements in a natural language plus diagram, describing the services the system is expected to provide and the constraints

- System Requirements–Describe the system's function ,services and operational condition

SOFTWARE REQUIREMENTS

- **System Functional Requirements**

--Statement of services the system should provide
--Describe the behavior in particular situations
--Defines the system reaction to particular inputs

- Non functional Requirements
- Constraints on the services or functions offered by the system

--Include timing constraints ,constraints on the development process and standards
--Apply to system as a whole

- Domain Requirements

--Requirements relate to specific application of the system
--Reflect characteristics and constraints of that system

FUNCTIONALREQUIREMENTS

- Should be both complete and consistent
- Completeness

--All services required by the user should be defined

- Consistent

--Requirements should not have contradictory definition

- Difficult to achieve completeness and consistency for large system

NON-FUNCTIONAL REQUIREMENTS

Types of Non-functional Requirements

1. Product Requirements

-Specify product behavior

-Include the following

- Usability
- Efficiency
- Reliability
- Portability

2. Organizational Requirements

--Derived from policies and procedures

--Include the following:

- Delivery
- Implementation
- Standard

3. External Requirements

--Derived from factors external to the system and its development process

--Includes the following

- Interoperability
- Ethical
- Legislative

PROBLEMSFACEDUSINGTHENATURALLANGUAGE

1. Lack of clarity–Leads to misunderstanding because of ambiguity of natural language
2. Confusion–Due to over flexibility, some time difficult to find whether requirements are same or distinct.
3. Amalgamation problem–Difficult to modularize natural language requirements

STRUCTURED LANGUAGE SPECIFICATION

- Requirements are written in a standard way
- Ensures degree of uniformity
- Provide templates to specify system requirements
- Include control constructs and graphical highlighting to partition the specification

SYSTEM REQUIREMENTS STANDARD FORM

- Function
- Description
- Inputs
- Source
- Outputs
- Destination
- Action
- Precondition
- Post condition
- Side effects

Interface Specification

- Working of new system must match with the existing system
- Interface provides this capability and precisely specified

Three types of interfaces

1. Procedural interface–Used for calling the existing programs by the new programs
2. Data structures-Provide data passing from one sub-system to another
3. Representations of Data
   --Ordering of bits to match with the existing system
   --Most common in real-time and embedded system

The Software Requirements document

The requirements document is the official statement of what is required of the system developers. Should include both a definition of user requirements and a specification of the system requirements. It is NOT a design document. As far as possible ,it should set of" WHAT" the system should do rather than HOW it should do it

The Software Requirements document
Heninger suggests that here are 6requirements that requirement document should satisfy. It should

- Specify only external system behavior
- Specify constraints on the implementation.
- Be easy to change
- Serve as reference tool for system maintainers
- Record for thought about the lifecycle of the system.
- Characterize acceptable responses to undesired events

Purpose of SRS

- Communication between the Customer, Analyst, system developers,

maintainers
- Firm foundation for the design phase
- Support system testing activities
- Support project management and control
- Controlling the evolution of the system

IEEE requirements standard

Defines a generic structure for a requirements document that must be instantiated for each specific system.
- Introduction.
- General description.
- Specific requirements.
- Appendices.
- Index.

IEEE requirements standard
1. Introduction :
2. Purpose
3. Scope
4. Definitions,
5. Acronyms and Abbreviations
6. References
7. Overview
8. General Description
9. Product perspective
10. Product function summary
11. User characteristics
12. General constraints
13. Assumptions and dependencies
14. Specific Requirements
15. Functional requirements

-External interface requirements
- Performance requirements
- Design constraints
- Attributeseg.security,availability,maintainability,transferability/conversion
- Other requirements
- Appendices
- Index

REQUIREMENTS ENGINEERING PROCESS

To create and maintain a system requirement document .The overall process includes four high level requirements engineering sub-processes:
Feasibility study:--Concerned with assessing whether the system is useful to the business

2.Elicitation and analysis:-- Discovering requirements

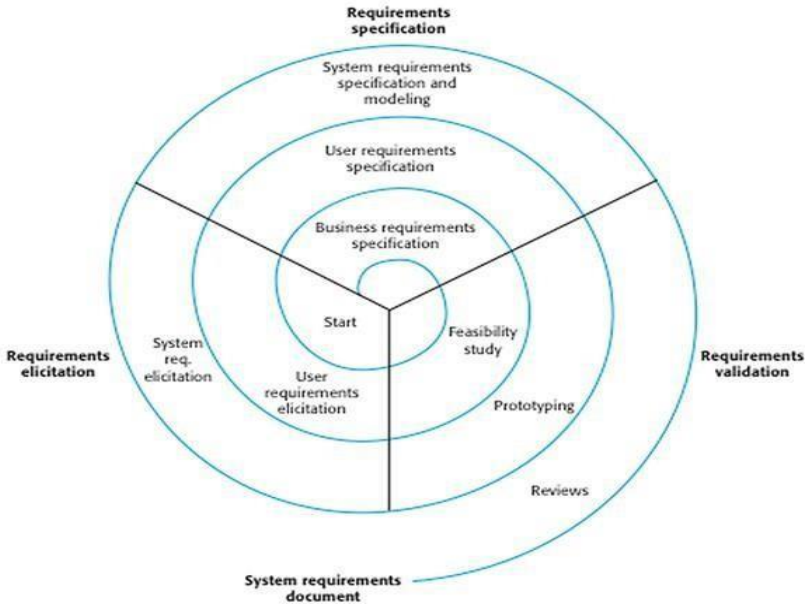
3. Specifications:-- Converting the requirements into a standard form
4. Validation:-- Checking that the requirements actually define the system that the customer wants

SPIRAL REPRESENTATION OF REQUIREMENTS ENGINEERING PROCESS

Process represented as three stage activity. Activities are organized as an iterative process around a spiral. Early in the process, most effort will be spent on understanding high-level business and the use requirement. Later in the outer rings, more effort will be devoted to system requirements engineering and system modeling

Three level process consists of :

- Requirements elicitation
- Requirements specification
- Requirements validation



FEASIBILITYSTUDIES

Starting point of the requirements engineering process

• Input :Set of preliminary business requirements, an outline description of the system and how the system is intended to support business processes

• Output : Feasibility report that recommends whether or not it is worth carrying out further Feasibility report answers a number of questions:

1. Does the system contribute to the overall objective
2. Can the system be implemented using the current technology and with in given cost and schedule
3. Can the system be integrated with other system which is already in place.

REQUIREMENTS ELICITATION ANALYSIS

Involves a number of people in an organization.

Stakeholder definition–Refers to any person or group who will be affected by the system directly or indirectly i.e. End users, Engineers, business managers, domain experts.

- Reasons why eliciting is difficult
- Stakeholder often don't know what they want from the computer system
- A stakeholder expression of requirements in natural language is sometimes difficult to understand.
- Different stakeholders express requirements differently
- Influences of political factors, Change in requirements due to dynamic environments.

REQUIREMENTS ELICITATION PROCESS

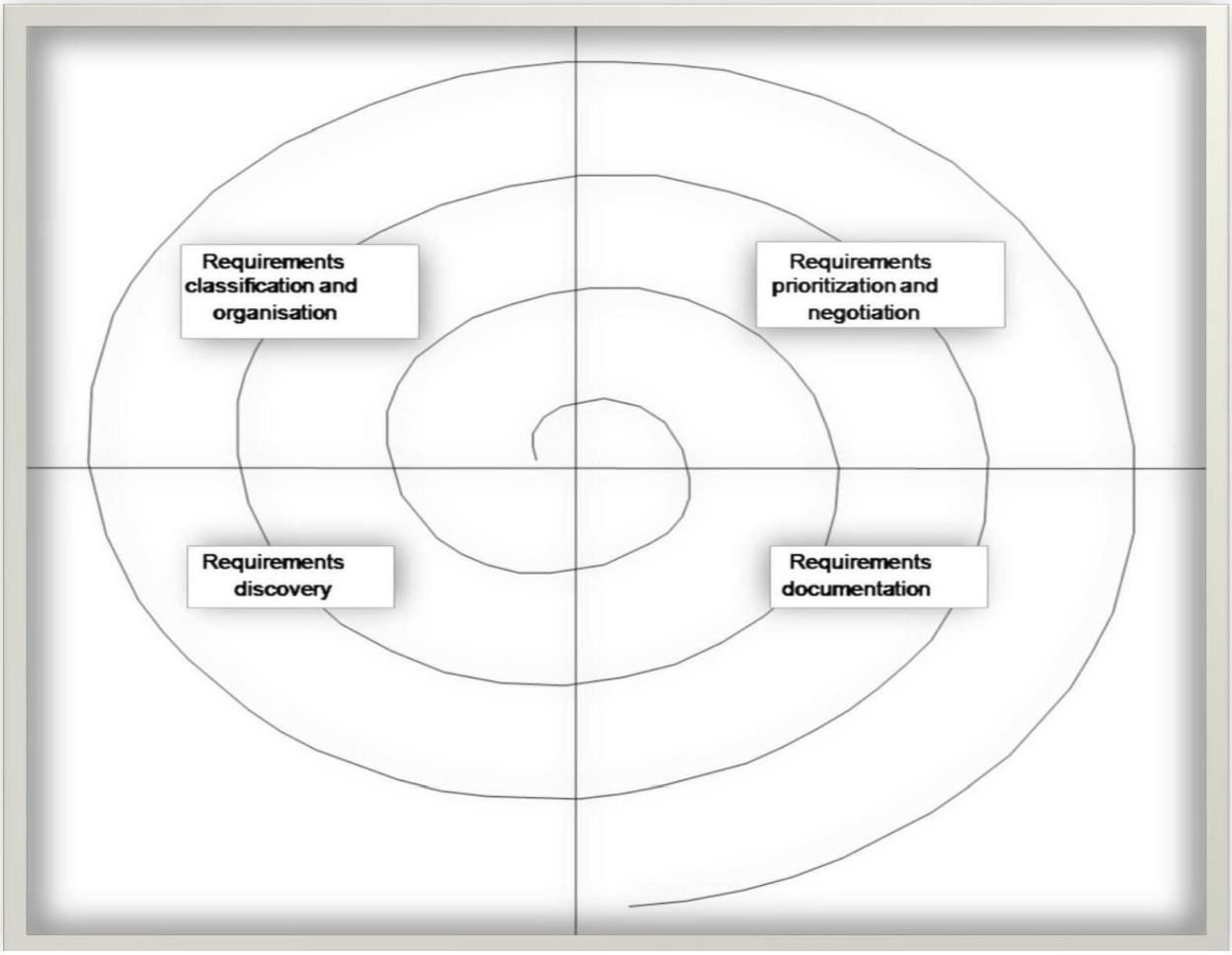
Process activities

1. Requirement Discovery–Interaction with stakeholder to collect their requirements including domain and documentation
2. Requirements classification and organization–Coherent clustering of requirements from unstructured collection of requirements
3. Requirements prioritization and negotiation–Assigning priority to requirements

--Resolves conflicting requirements through negotiation

4. Requirements documentation–Requirements be documented and placed in the next round of spiral

The spiral representation of Requirements Engineering



REQUIEMENTSDICOVERYTECHNIQUES

1. Viewpoints -Based on the viewpoints expressed by the stakeholder
--Recognizes multiple perspectives and provides a framework for discovering conflicts in the requirements proposed by different stakeholders
Three Generic types of viewpoints

- Inter actor viewpoint–Represents people or other system that interact directly with the system
- Indirect viewpoint–Stakeholders who influence the requirements ,but don't use the system
- Domain viewpoint–Requirements domain characteristics and constraints that influence the requirements.
- Inter viewing--Puts questions to stakeholders about the system that they use and the system to be developed.

Requirements are derived from the answers.

Two types of interview

- Closed interviews where the stakeholder's answers pre-defined set of questions.
- Open interviews discuss a range of issues with the stakeholders for better understanding their needs.

Effective interviewers

a) Open-minded : no pre-conceived ideas
b) Prompter: prompt the interview to start discussion with a question or a proposal

2. Scenarios --Easier to relate to real life examples than to abstract description. Starts with an outline of the interaction and during elicitation ,details are added to create a complete description of that interaction

Scenario includes:

1. Description at the start of the scenario
2. Description of normal flow of the event
3. Description of what can go wrong and how this is handled
4. Information about other activities parallel to the scenario
5. Description of the system state when the scenario finishes

LIBSYS scenario

- **Initial assumption:** The user has logged onto the LIBSYS system and has located the journal containing the copy of the article.
- **Normal**: The user selects the article to be copied. He or she is then prompted by the system to either provide subscriber information for the journal or to indicate how they will pay for the article. Alternative payment methods are by credit card or by quoting an organizational account number.
- The user is then asked to fill in a copyright form that maintains details of the transaction and they then submit this to the LIBSYS system.
- The copyright form is checked and, if OK, the PDF version of the article is downloaded to the LIBSYS working area on the user's computer and the user is informed that it is available. The user is asked to select a printer and a copy of the article is printed

LIBSYS scenario

- **What can go wrong**: The user may fail to fill in the copyright form correctly. In this case, the form should be re-presented to the user for correction. If there submitted form is still in correct then the user's request for the article is rejected.
- The payment may be rejected by the system. The user's request for the article is rejected.
- The article download may fail. Retry until successful or the user terminates the session.
- **Other activities:** Simultaneous downloads of other articles.
- **System state on completion**: User is logged on. The downloaded article has been deleted from LIBSYS workspace if it has been flagged as print-only.

Use cases–scenario based technique for requirement elicitation. A fundamental feature of UML, notation for describing object-oriented system models. Identifies a type of interaction and the actors involved. Sequence diagrams are used to add information to a Use case

Article printing use-case

Article printing LIBSYS use cases

Article printing Article search
User administration Supplier Catalogue services Library
User Library Staff

REQUIREMENTSVALIDATION

Concerned with showing that the requirements define the system that the customer wants. Important because errors in requirements can lead to extensive rework cost
Validation checks

- Validity checks –Verification that the system performs the intended function by the user
- Consistency check --Requirements should not conflict
- Completeness checks–Includes requirements which define all functions and constraints intended by the system user

Realism checks –Ensures that the requirements can be actually implemented
Verifiability –Testable to avoid disputes between customer and developer.

VALIDATION TECHNIQUES

- **REQUIREMENTS REVIEWS :** Reviewers check the following:

- Verifiability: Testable
- Comprehensibility
- Traceability
- Adaptability

- **PROTO TYPING**

- **TEST-CASE GENERATION**

Requirements management
Requirements are likely to change for large software systems and as such requirements management process is required to handle changes.
Reasons for requirements changes

(a) Diverse Users community where users have different requirements and priorities
(b) System customers and end users are different
(c) Change in the business and technical environment after installation two classes of requirements

(d) Enduring requirements : Relatively stable requirements
(e) Volatile requirements : Likely to change during system development process or during operation

Requirements management planning
An essential first stage in requirement management process. Planning process consists of the following

i. Requirements identification–Each requirement must have unique tag for cross reference and traceability

ii. Change management process–Set of activities that assess the impact and cost of changes

iii. Traceability policy--A matrix showing links between requirements and other elements of software development

iv. CASE tool support–Automatic tool to improve efficiency of change management process. Automated tools are required for requirements storage, change management and traceability management

Traceability
Maintains three types of trace ability information.
1. Source traceability–Links the requirements to the stakeholders
2. Requirements traceability–Links dependent requirements within the requirements document
3. Design traceability–Links from the requirements to the design module

Req. id	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		D	R					
1.2			D			D		D
1.3	R			R				
2.1			R		D			D
2.2								D
2.3		R		D				
3.1								R
3.2							R	

A traceability matrix Requirements change management consists of three principal stages:

1. Problem analysis and change specification--Process starts with a specific change proposal and analyzed to verify that it is valid
2. Change analysis and costing–Impact analysis in terms of cost, time and risks
3. Change implementation–Carrying out the changes in requirements document, system design and its implementation

UNIT III

Design Engineering: Design process and design quality, design concepts, the design model. Creating an architectural design: software architecture, data design, architectural styles and patterns, architectural design, conceptual model of UML, basic structural modeling, class diagrams, sequence diagrams, collaboration diagrams, use case diagrams, component diagrams

---

DESIGN PROCESS AND DESIGNQUALITY

Encompasses the set of principles, concepts and practices that lead to the development of high quality system or product. Design creates a representation or model of the software. Design model provides details about S/W architecture, interfaces and components that are necessary to implement the system. Quality is established during Design. Design should exhibit firmness, commodity and design. Design sits at the kernel of S/W Engineering. Design sets the stage for construction.

QUALITYGUIDELINES

- Uses recognizable architectural styles or patterns
- Modular that is logically partitioned into elements or subsystems
- Distinct representation of data, architecture, interfaces and components
- Appropriate data structures for the classes to be implemented
- Independent functional characteristics for components
- Interfaces that reduces complexity of connection
- Repeatable method

QUALITY ATTRIBUTES

FURPS quality attributes

- **Functionality**
  - Features and capabilities of programs
  - Security of the overall system
- **Usability**
  - user-friendliness
  - Aesthetics
  - Consistency
  - Documentation
- **Reliability**
  - Evaluated by measuring the frequency and severity of failure
  - MTTF
- **Supportability**
  - Extensibility
  - Adaptability
  - Serviceability

DESIGN CONCEPTS

1. Abstractions

2. Architecture
3. Patterns
4. Modularity
5. Information Hiding
6. Functional Independence
7. Refinement
8. Re-factoring
9. Design Classes

DESIGN CONCEPTS
ABSTRACTION
Many levels of abstraction.
Highest level of abstraction: Solution is slated in broad terms using the language of the problem environment
Lower levels of abstraction: More detailed description of the solution is provided

- Procedural abstraction–Refers to a sequence of instructions that a specific and limited function
- Data abstraction–Named collection of data that describe a data object

DESIGN CONCEPTS

ARCHITECTURE–Structure organization of program components (modules) and their interconnection Architecture Models
(a) Structural Models–An organized collection of program components
(b) Framework Models–Represents the design in more abstract way
(c) Dynamic Models–Represents the behavioral aspects indicating changes as a function of external events
(d).Process Models–focus on the design of the business or technical process

PATTERNS
Provides a description to enables a designer to determine the followings:
(a). Whether the pattern is applicable to the current work
(b). Whether the pattern can be reused
(c). Whether the pattern can serve as a guide for developing a similar but functionally or structurally different pattern

MODULARITY
Divides software into separately named and addressable components, sometimes called modules. Modules are integrated to satisfy problem requirements. Consider two problems p1 andp2. If the complexity of p1 iscp1 and of p2 is cp2 then effort to solve p1=cp1 and effort

t

o solve p2=cp2 If cp1>cp2 then ep1>ep2
The complexity of two problems when they are combined is often greater than the sum of the perceived complexity when each is taken separately.
Based on Divide and Conquer strategy:
it is easier to solve a complex problem when broken into sub-modules

INFORMATION HIDING

Information contained within a module is inaccessible to other modules who do not need such information. Achieved by defining a set of Independent modules that communicate with one another only that information necessary to achieve S/W function. Provides the greatest benefits when modifications are required during testing and later. Errors introduced during modification are less likely to propagate to other location within the S/W.

FUNCTIONALINDEPENDENCE

A direct outgrowth of Modularity. Abstraction and information hiding. Achieved by developing a module with single minded function and an aversion to excessive interaction with other modules. Easier to develop and have simple interface. Easier to maintain because secondary effects caused by design or code modification are limited, error propagation is reduced and reusable modules are possible. Independence is assessed by two quantitative criteria:

(1) Cohesion
(2) Coupling

Cohesion-- Performs a single task requiring little interaction with other components Coupling--Measure of interconnection among modules.
Coupling should be low and cohesion should be high for good design.

REFINEMENT & REFACTORING

REFINEMENT -- Process of elaboration from high level abstraction to the lowest level abstraction. High level abstraction begins with a statement of functions. Refinement causes the designer to elaborate providing more and more details at successive level of abstractions Abstraction and refinement are complementary concepts.

REFACTORING -- Organization technique that simplifies the design of a component without changing its function or behavior. Examines for redundancy, unused design elements and inefficient or unnecessary algorithms.

DESIGN CLASSES-- Class represents a different layer of design architecture.

Five types of Design Classes

1. User interface class–Defines all abstractions that are necessary for human computer interaction
2. Business domain class --

Refinementoftheanalysisclassesthatidentityattributesandservicesto implement some of business domain

3. Process class –implement slower level business abstractions required to fully manage the business domain classes
4. Persistent class -- Representdatastoresthatwillpersistbeyondtheexecutionofthesoftware
5. System class --Implements management and control functions to operate and communicate within the computer environment and with the outside world.

THE DESIGN MODEL

Analysis viewed in two different dimensions as process dimension and abstract dimension. Process dimension indicates the evolution of the design model as design tasks are executed as part of software process.
Abstraction dimension represents the level of details as each element of the analysis model is transformed into design equivalent
Data Design elements
--Data design creates a model of data that is represented at a high level of abstraction
--Refined progressively to more implementation-specific representation for processing by the computer base system
--Translation of data model into a data base is pivotal to achieving business objective of a system

THE DESIGN MODEL

Architectural design elements .Derived from three sources

- Information about the application domain of the software
- Analysis model such as data flow diagrams or analysis classes.
- Architectural pattern and styles Interface Design elements Set of detailed drawings constituting:
- User interface
- External interfaces to other systems, devices etc
- Internal interfaces between various components

THE DESIGN MODEL

Deployment level design elements. Indicate show software functionality and sub system will be allocated within the physical computing environment. UML deployment diagram is developed and refined Component level design elements fully describe the internal details of each software

component. UML diagram can be used

CREATING ANARCHITECTURAL DESIGN

What is SOFTWARE ARCHITECTURE… The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationship among them.

Software Architecture is not the operational software. It is a representation that enables a software engineer to

- Analyze the effectiveness of the design in meeting its stated requirements.
- Consider architectural alternative at a stage when making design changes is still relatively easy.
- Reduces the risk associated with the construction of the software.

Why Is Architecture Important? Three key reasons
--Representations of software architecture enable communication and understanding between stakeholders
--Highlights early design decisions to create an operational entity.
--constitutes a model of software components and their interconnection

Data Design
The data design action translates data objects defined as part of the analysis model into data structures at the component level and database architecture at application level when necessary.

DATA DESIGN ATARCHITECTURE LEVEL

- Data structure at programming level
- Database at application level
- Data warehouse at business level.

DATA DESIGN AT COMPONENT LEVEL

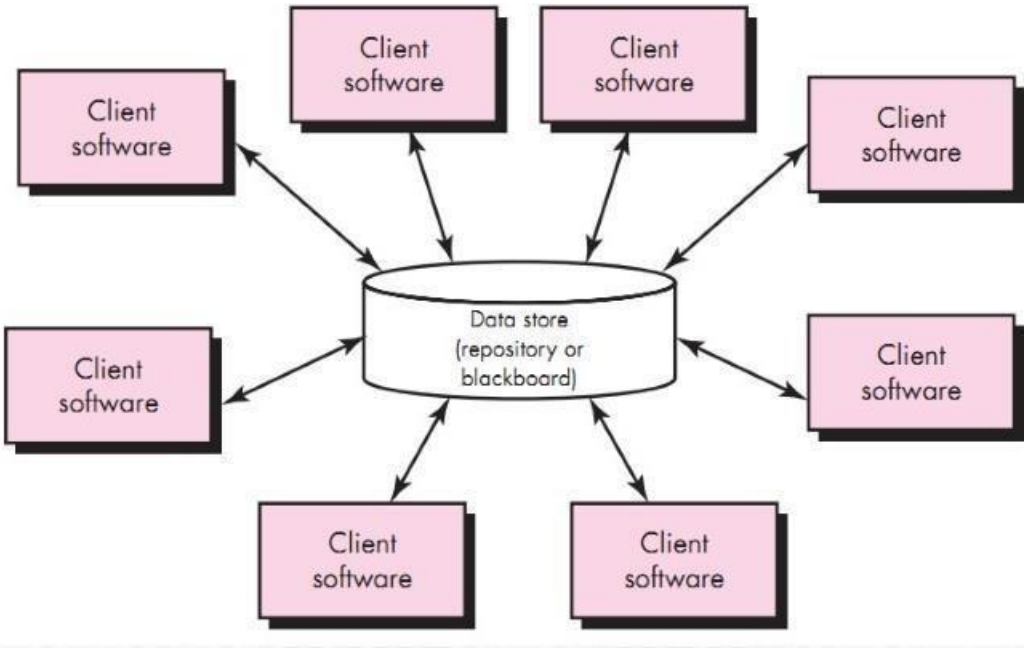
Principles for data specification:

- Proper selection of data objects and data and data models
- Identification of attribute and functions and their encapsulation of these with in a class
- Mechanism for representation of the content of **each** data object. Class diagrams may be used
- Refinement of data design elements from requirement analysis to component level design.
- Information hiding
- A library of useful data structures and operations be developed.
- Software design and PL should support the specification and

realization of abstract data types.

ARCHITECTURAL STYLES

Describes a system category that encompasses:
(1) a set of *components*
(2) a set of *connectors* that enables "communication and coordination
(3) C*onstraints* that define how components can be integrated to form the system
(4) S*emantic models* to understand the overall properties of a system



Data-flow architectures
Shows the flow of input data ,its computational components and output data. Structure is also called pipe and Filter. Pipe provides path for flow of data. Filters manipulate data and work independent of its neighboring filter. If data flow degenerates into a single line of transform, it is termed as batch sequential.
Call and return architectures
Achieves a structure that is easy to modify and scale . Two sub styles
(1) Main program/sub program architecture
--Classic program structure
--Main program invokes a number of components, which in turn invoke still

other components

(2) Remote procedure call architecture
--Components of main program/sub program are distributed across computers over network

Object-oriented architectures
The components of a system encapsulate data and the operations. Communication and coordination between components is done via message

Layered architectures
A number of different layers are defined Inner Layer(interface with OS)

- Intermediate Layer Utility services and application function)Outer Layer(User interface)

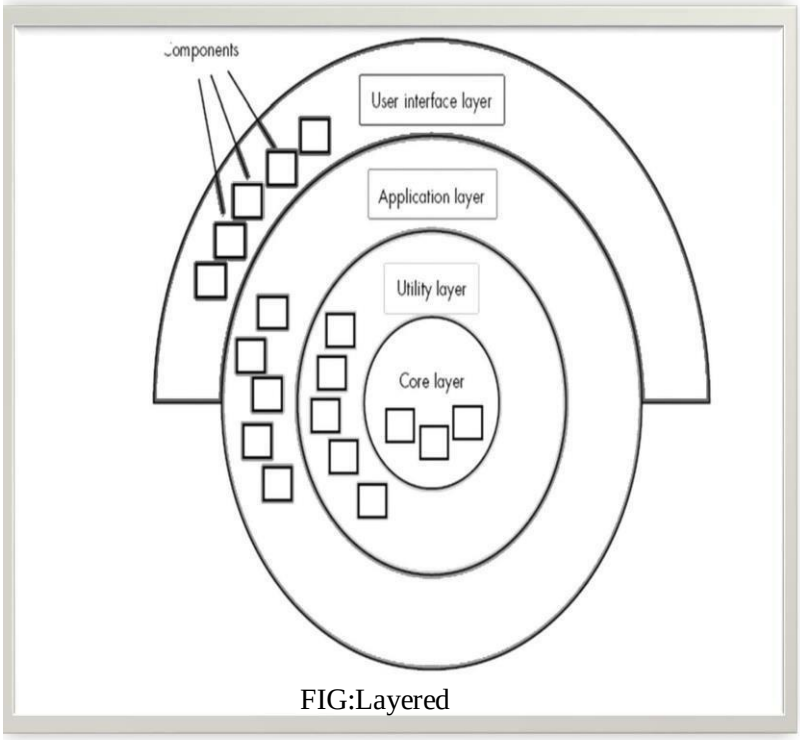


FIG:Layered

ARCHITECTURAL PATTERNS
A template that specifies approach for some behavioral characteristics of the system Patterns are imposed on the architectural styles
Pattern Domains
1. Concurrency
--Handles multiple tasks that simulate parallelism.

--Approaches (Patterns)
(a) Operating system process management pattern
(b) A task scheduler pattern
(c) Persistence
--Data survives past the execution of the process
--Approaches (Patterns)
(a) Data base management system pattern
(b) Application Level persistence Pattern(word processing software)

1. Distribution
--Addresses the communication of system in a distributed environment
--Approaches (Patterns)
(a)Broker Pattern
--Acts as middleman between client and server.

Conceptual model of UML, basic structural modeling, class diagrams, sequence diagrams, collaboration diagrams, use case diagrams, component diagrams

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

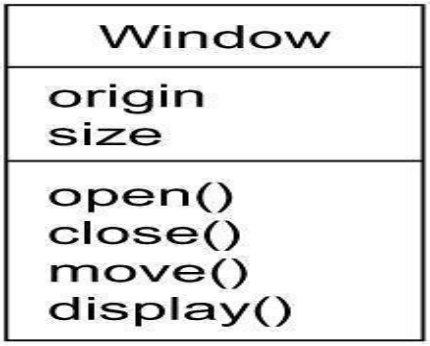
These things are the basic object-oriented building blocks of the UML. You use them to write well-formed models.

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

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 Interfaces

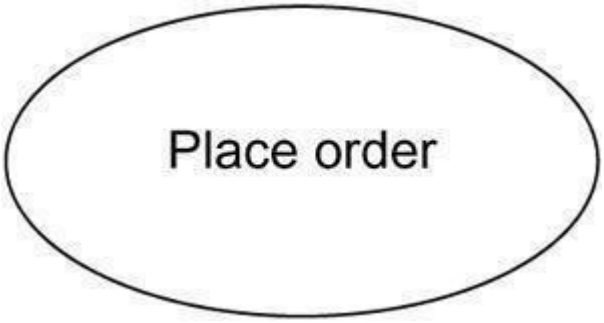
Third, collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class might participate 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

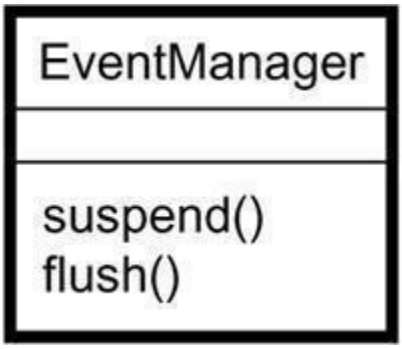
Fourth, a ***use case*** is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor. A use case is used to structure the behavioral things in a model. A use case is realized by collaboration. Graphically, a use case is rendered 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.

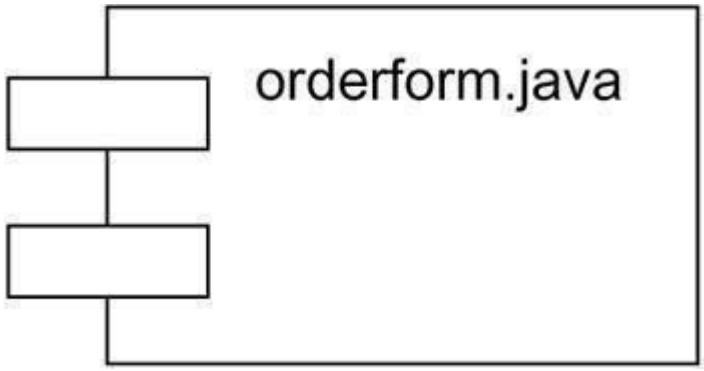
Figure Active Classes



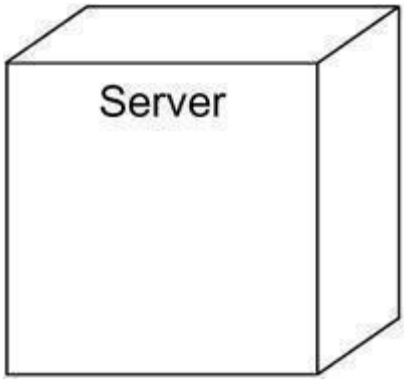
The remaining two elements• component, and nodes• are also different. They represent physical things, whereas the previous five things represent conceptual or logical things.

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 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.

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 sub states, if any.

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.

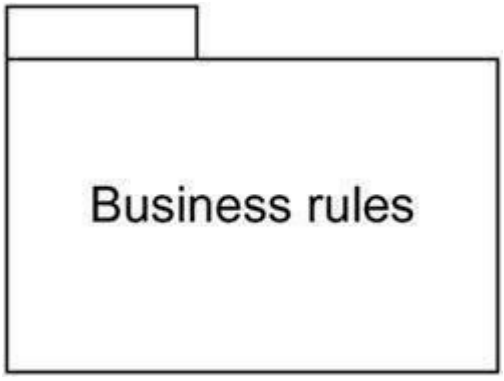
Grouping Things

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.

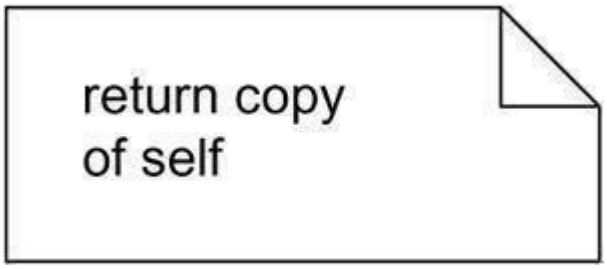
Figure Packages



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 .



Relationships in the UML

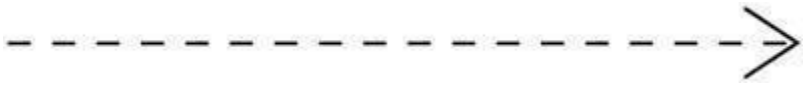
There are four kinds of relationships in the UML:

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.

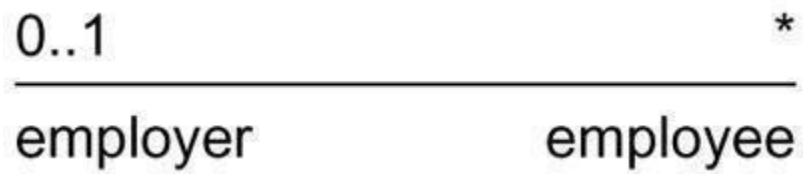
First, a *dependency* is a semantic relationship between two things in which a change to one thing (the independent thing) may affect the semantics of the other thing (the dependent thing). Graphically, a dependency is rendered as a dashed line, possibly directed, and occasionally including a label,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



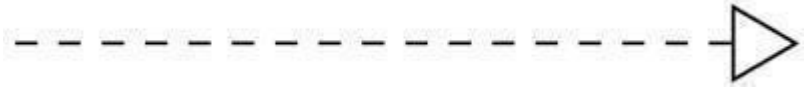
Third, a *generalization* is a specialization/generalization relationship in which objects of the specialized element (the child) are substitutable for objects of the generalized element (the parent). In this way, the child shares the structure and the behavior of the parent. Graphically, a generalization 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 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. State chart 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 *state chart diagram* shows a state machine, consisting of states, transitions, events, and activities. State chart 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 state chart 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 address the static deployment view of an architecture. They are related to component diagrams in that a node typically encloses one or more components the configuration of run-time processing nodes and the components that live on 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, and inconsistent.

Common Mechanisms in the UML

It is made simpler by the presence of four common mechanisms that apply consistently throughoutthe 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

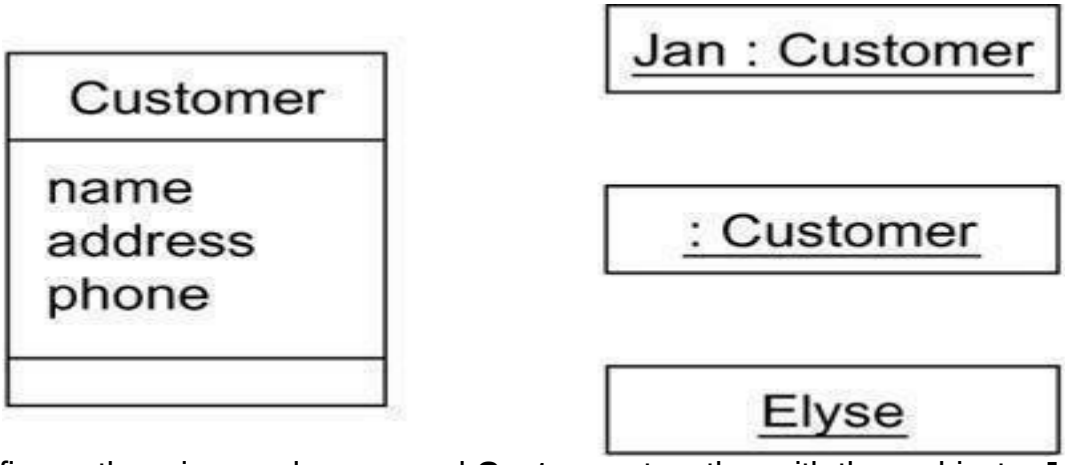
Most elements in the UML have a unique and direct graphical notation that provides a visual representation of the most important aspects of the element. For example, the notation for a class is intentionally designed to be easy to draw, because classes are the most common element found in modeling object-oriented 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 ofadornments 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 shownin Figure.

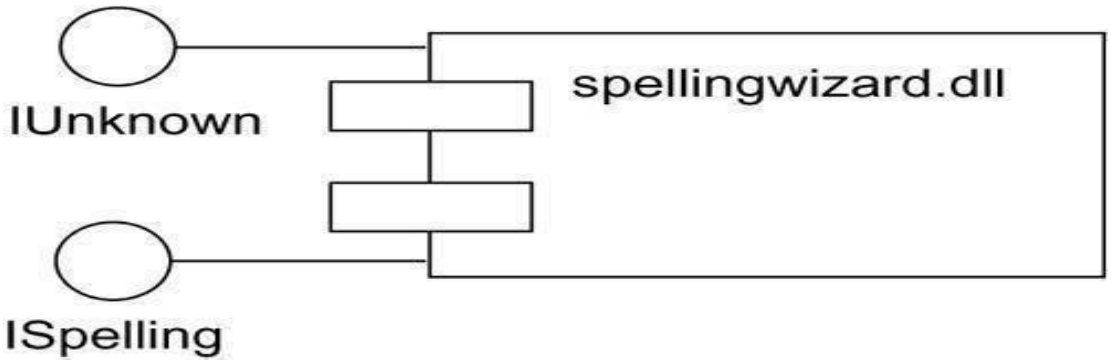
Figure Classes And Objects



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, **I Unknown** 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 .

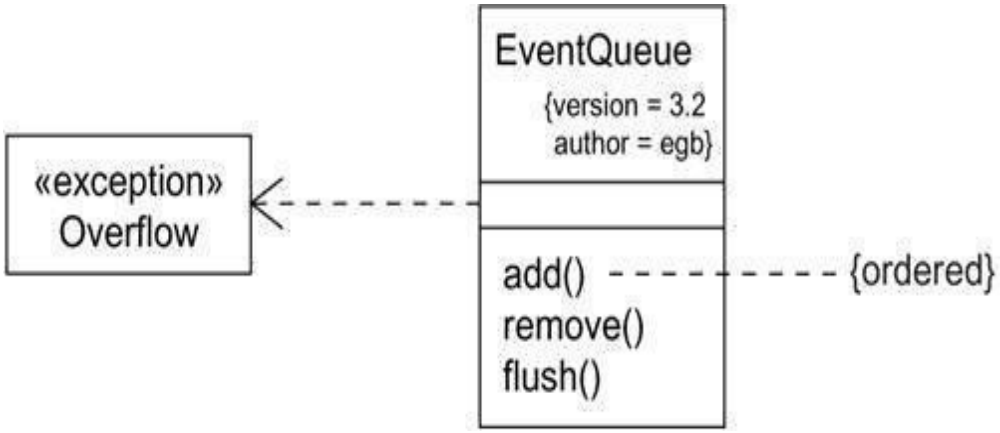


Figure Extensibility Mechanisms

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 **Event Queue** 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 **Event Queue** class so that all additions are done in order. You can add a constraint that explicitly marks these for the operation **adds.**

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, state chart 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.

Basic structural modeling:

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.

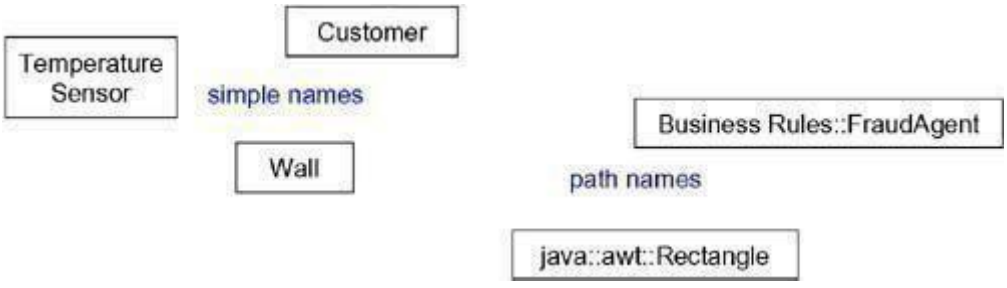
Terms and Concepts

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.

Figure Simple and Path Names



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

Operations

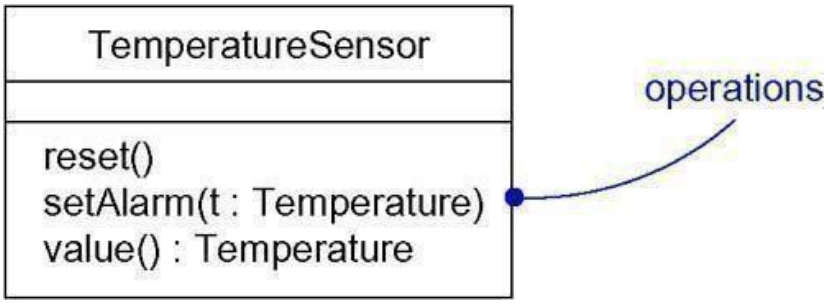
An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class. A class may have any number of operations or no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

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

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



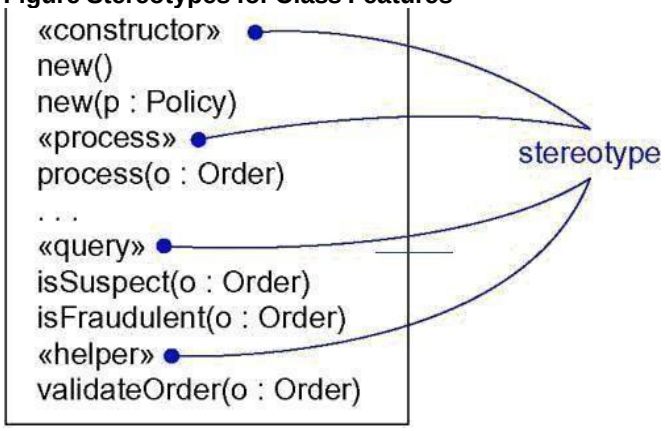
Organizing Attributes and Operations

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 witha 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 **Fraud Agent** 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 **Temperature Sensor** 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.

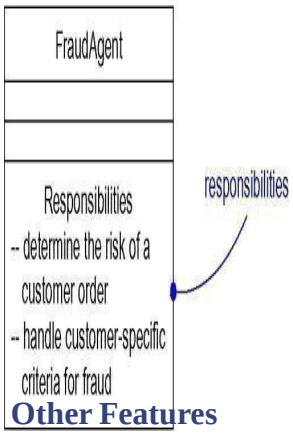


Figure Responsibilities

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 needto convey the most important semantics of your classes.

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 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.

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 **Fraud Agent** 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 **Temperature Sensor** 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

Other Features

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 needto 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).

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.

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).

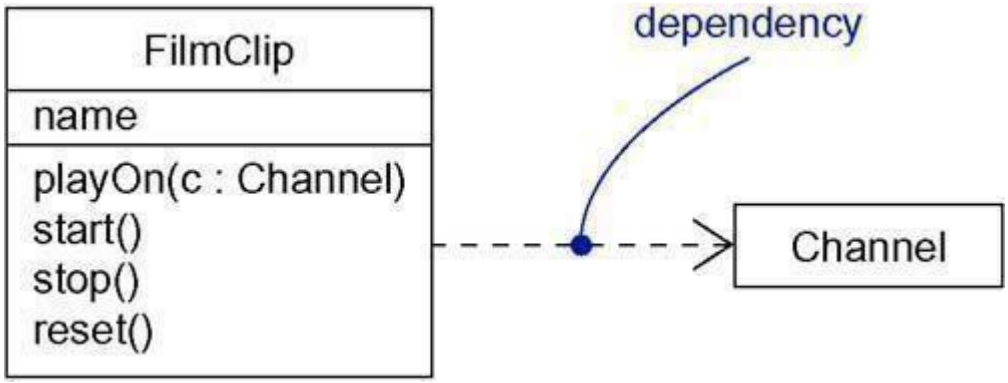
Finally, classes rarely stand alone. Rather, when you build models, you will typically focus on groups

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

Figure Dependencies



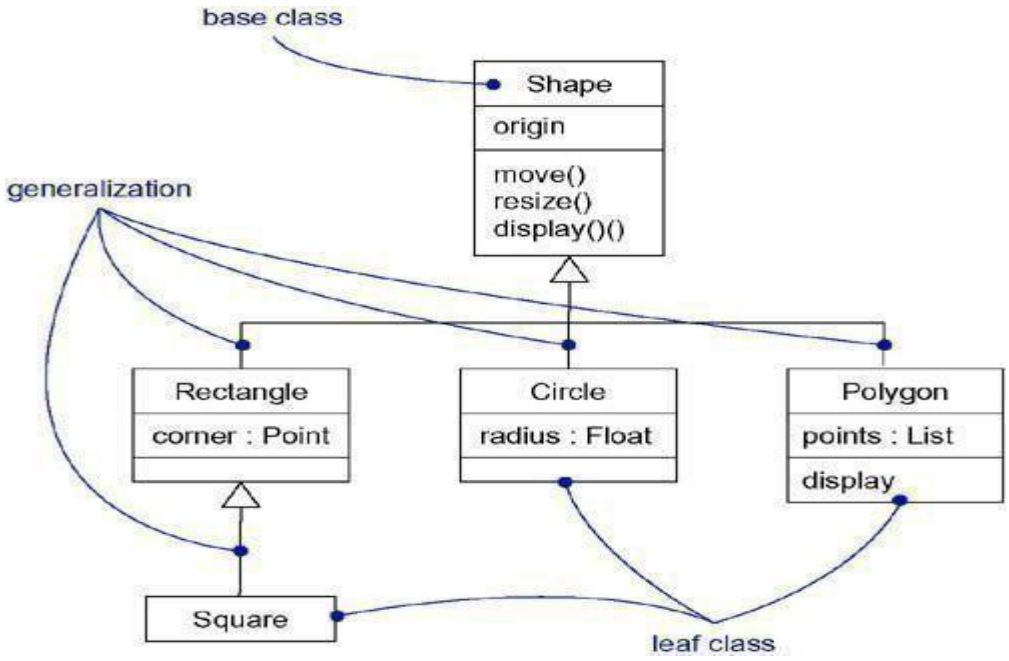
Generalization

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 attri**butes 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 objectof one class to anobject 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 as 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.

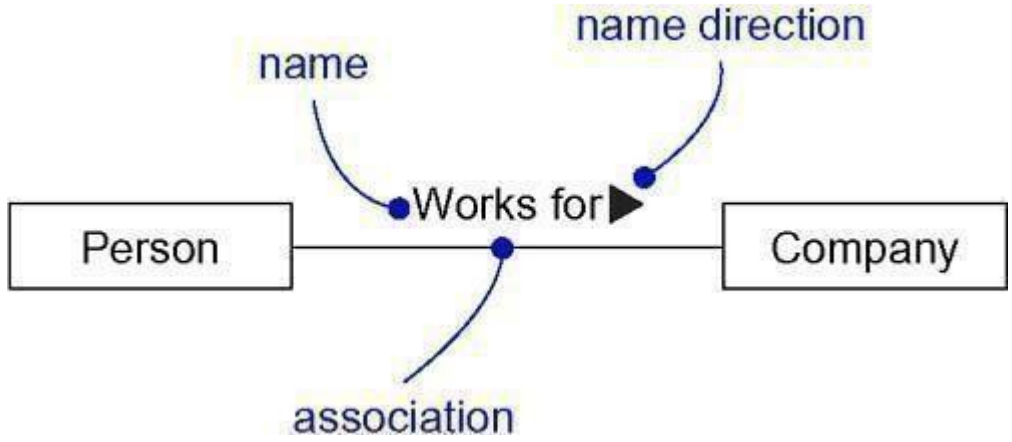
Beyond this basic form, there are four adornments that apply to associations.

Name

An association can have a name, and you use that name to describe the nature of the relationship. Sothat 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.

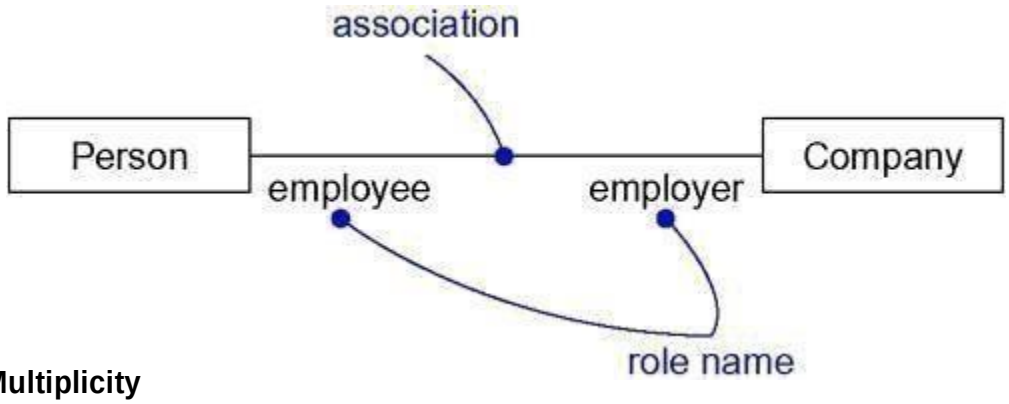
Figure Association Names



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

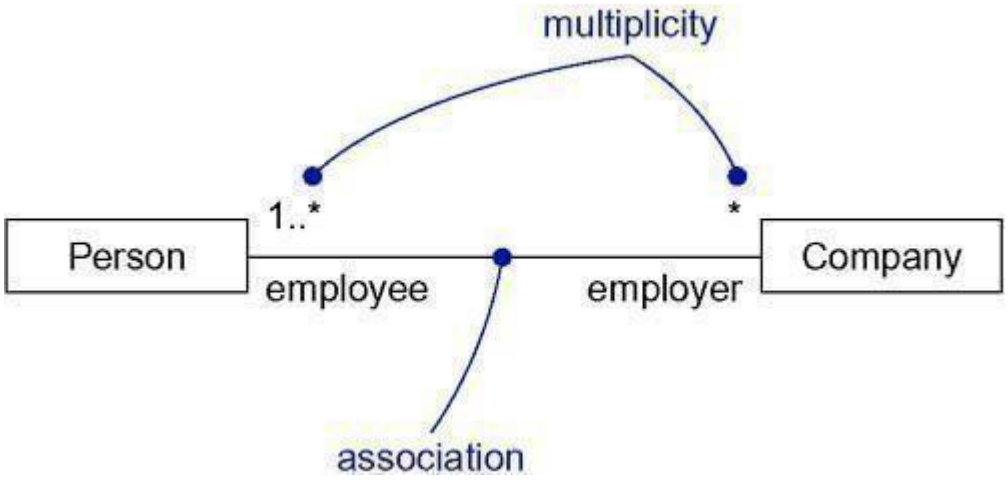


Multiplicity

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**).

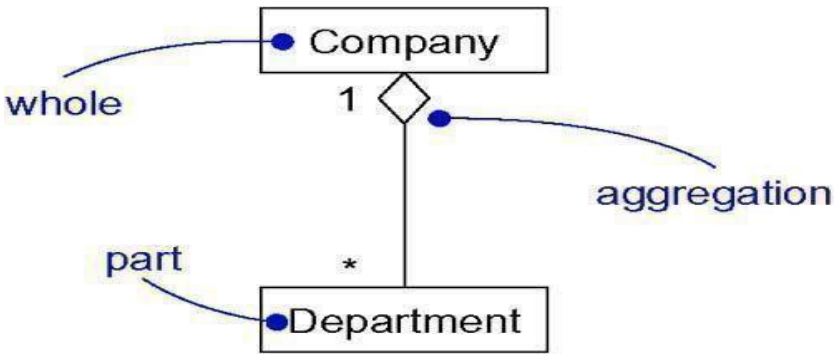
Figure Multiplicity



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.

Structural Diagrams

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

1	Class diagram	Class, interfaces and collaborations
2	Objects diagram	Objects
3	Component diagram	Components
4	Deployment diagram	Nodes

UNIT IV

Testing Strategies : A strategic approach to software testing ,test strategies for conventional software, Black-Box and White-Box testing, Validation testing, System testing, the art of Debugging.

Product metrics: Software Quality, Metrics for Analysis Model ,Metrics for Design Model, Metrics for source code, Metrics for testing, Metrics for maintenance.

Metrics for Process and Products: Software Measurement, Metrics for software quality.

Testing Strategies

Software is tested to uncover errors introduced during design and construction. Testing often accounts for more project effort than other s/e activity. Hence it has to be done carefully using a testing strategy.

The strategy is developed by the project manager, software engineers and testing specialists. Testing is the process of execution of a program with the intention of finding errors Involves 40% of total project cost

Testing Strategy provides a road map that describes the steps to be conducted as part of testing. It should incorporate test planning, test case design, test execution and resultant data collection and execution

Validation refers to a different set of activities that ensures that the software is traceable to the customer requirements. V&V encompasses a wide array of Software Quality Assurance.

The customer requirements.

A strategic Approach for Software testing

Testing is a set of activities that can be planned in advance and conducted systematically.
Testing strategy
Should have the following characteristics:
--usage of Formal Technical Reviews(FTR)
--Begins at component level and covers entire system
--Different techniques at different points
--conducted by developer and test group
--should include debugging
Software testing is one element of verification and validation.
Verification refers to these activities that ensure that software correctly implements a specific function.
(Ex: A rebuilding the product right?)
Validation refers to the set of activities that ensure that the software built is traceable to customer requirements.
(Ex:Are we building the right product?)

Testing Strategy

Testingcanbedonebysoftwaredeveloperandindependenttestinggroup.Testingand debugging are different activities. Debugging follows testing

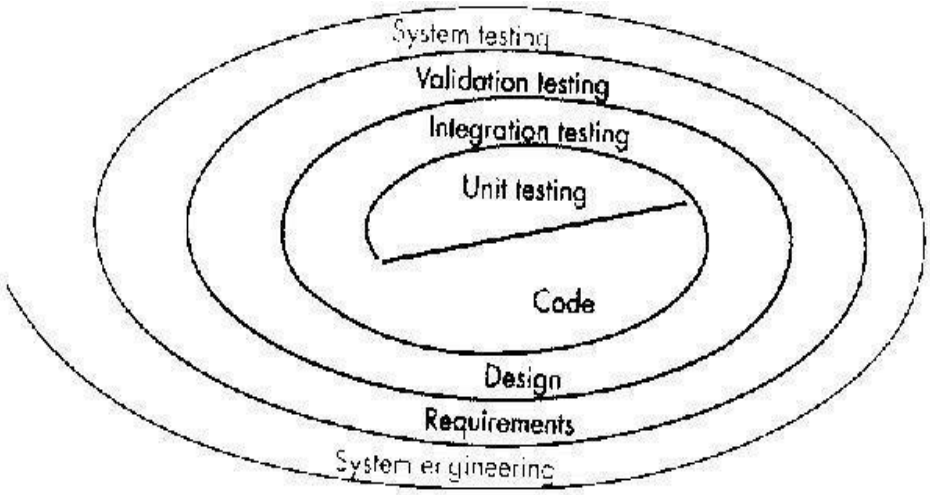
Low level tests verifies small code segments. High level tests validate major system functions against customer requirements

Test Strategies for Conventional Software:

Testing Strategies for Conventional Software can be viewed as a spiral consisting off our levels of testing:

1) Unit Testing
2)Integration Testing
3)Validation Testing and
4)System Testing

SpiralRepresentationofTesting forConventionalSoftware



Unit Testing begins at the vortex of the spiral and concentrates one a chunk of software in source code.
It uses testing techniques that exercise specific paths in a component and its control structure to ensure
Complete coverage and maximum error detection .It focuses on the internal processing logic and data structures. Test cases should uncover errors.

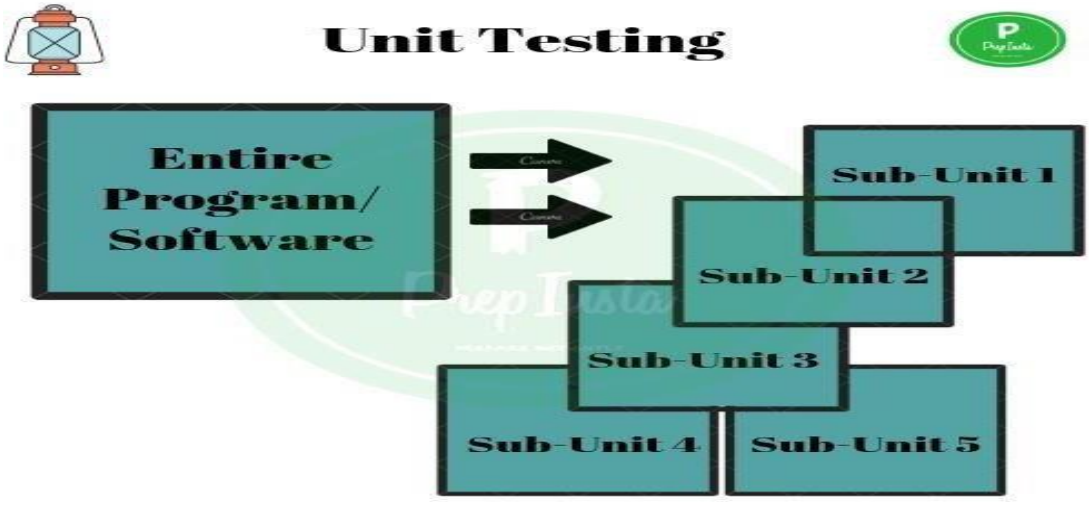


Fig:Unit Testing

Boundary testing also should be done as s/w usually fails at its boundaries. Unit tests can be designed before coding begins or after

source code is generated.

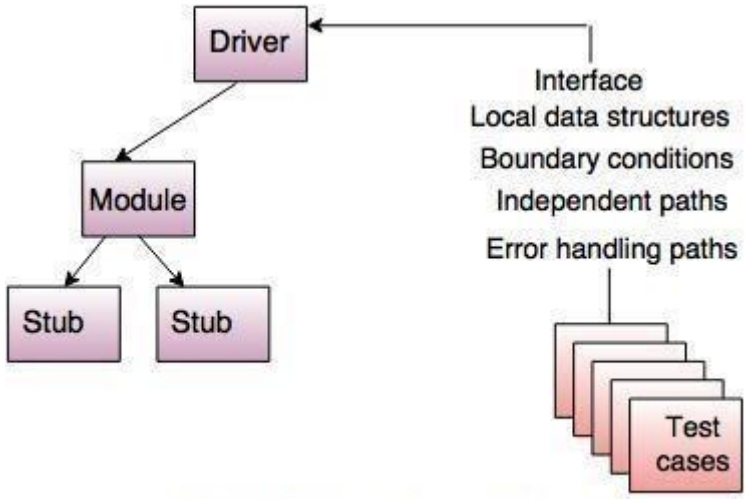


Fig. - Unit test environment

Integration testing: In this the focus is on design and construction of the software architecture. It addresses the issues associated with problems of verification and program construction by testing inputs and outputs. Though modules function independently problems may arise because of interfacing. This technique uncovers errors associated with interfacing. We can use top-down integration wherein modules are integrated by moving downward through the control hierarchy, beginning with the main control module. The other strategy is bottom -up which begins construction and testing with atomic modules which are combined into clusters as we move up the hierarchy. A combined approach called Sandwich strategy can be used i.e., top- down for higher level modules and bottom-up for lower level modules.

Testing Tactics:

The goal of testing is to find errors and a good test is one that has a high probability of finding an error.

A good test is not redundant and it should be neither too simple nor too complex. Two major categories of software testing

- Black box testing: It examines some fundamental aspect of a system, tests whether each function of Product is fully operational.
- White box testing: It examines the internal operation sofa system and examines the procedural detail.

Black box testing

This is also called behavioral testing and focuses on the functional requirements of software. It fully Exercises all the functional requirements for a program and finds incorrect or missing functions, interface Errors, database errors etc. This is performed in the later stages in the testing process. Treats the system as Black box whose behavior can be determined by studying its input and related output not concerned with the internal. The various testing methods employee adheres are:

Graph based testing method: Testing begins by creating a graph of important objects and their relationships And then devising a series of tests that will cover the graph so that each object and relationship is exercised And errors are uncovered.

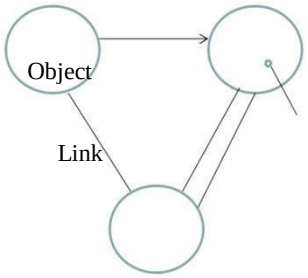


Fig:O-Rgraph.

1)Equivalence partitioning : This divides the input domain of a program into classes of data from which test cases can be derived. Define test cases that uncover classes of errors so that no. of test cases are reduced. This is based on equivalence classes which represent a set of valid or invalid states for input conditions. Reduces the cost of testing

Example

Input consists of1to10

Then classes are n<1,1<=n<=10,n>10

Choose one valid class with value with in the allowed range and two invalid classes where values are greater than maximum value and smaller than minimum value.

2) Boundary Value analysis

Select input from equivalence classes such that the input lies at the edge of the equivalence classes. Set of data lies on the edge or boundary of a class of input data or generates the data that lies at the boundary of a class of output data .Test cases exercise boundary values to uncover errors at the boundaries of the input domain.

Example

If0.0<=x<=1.0

Then test cases are(0.0,1.0)for valid input and(-0.1and1.1)for invalid input

3) Orthogonal array Testing

This method is applied to problems in which input domain is relatively small but too large for exhaustive testing

Example

Three inputs A,B,C each having three values will require 27 test cases**. O**rthogonal testing will reduce the number of test case to 9 as shown below

White Box testing

Also called glass box testing. It uses the control structure to derive test cases. It exercises all independent paths ,Involves knowing the internal working of a program, Guarantees that all independent

Paths will be exercised at least once .Exercises all logical decisions on their true and false sides**,** Executes all loops, Exercises all data structures for their validity. White box testing techniques

- Basis path testing
- Control structure testing

1. Basis path testing

Proposed by Tom Mc Cabe. Defines a basic set of execution paths based on logical complexity of a

Procedural design. Guarantees to execute every statement in the program at least once

Steps of Basis Path Testing

1. Draw the flow graph from flowchart of the program
2. Calculate the cyclomatic complexity of the resultant flow graph
3. Prepare test cases that will force execution

of each path Two methods to compute Cyclomatic complexity number

1.V(G)=E-N+2whereEisnumber of edges ,N is number of nodes

2.V(G)=Number of regions

The structured constructs used in the flow graph are:

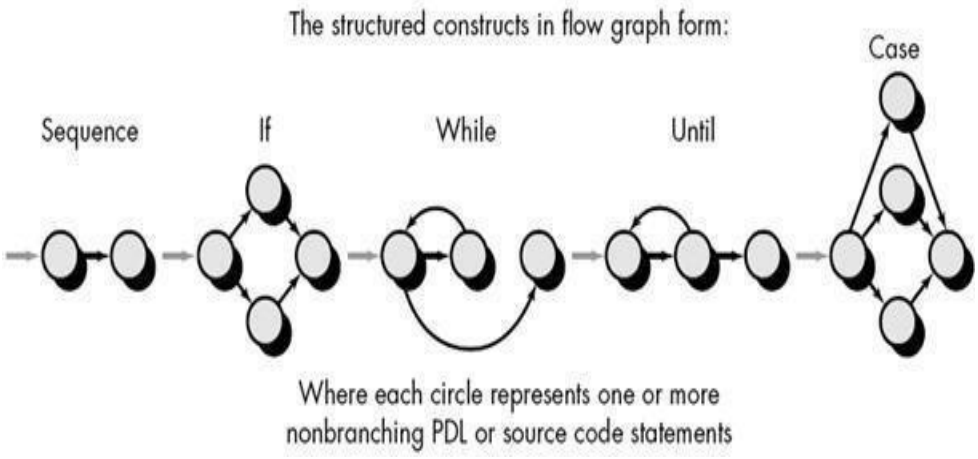


Fig:Basis path Testing

Basis path testing is simple and effective .It is not sufficient in itself

2. Control Structure testing

This broadens testing coverage and improves quality of testing. It uses the following methods:

a) Condition testing :Exercises the logical conditions contained in a program module.

Focuses on testing each condition in the program to ensure that it does not contain errors

Simple condition
E1<relation operator>E2 Compound condition
simple condition<Boolean operator>simple
condition
Types of errors include operator errors, variable errors, arithmetic expression errors etc.

b) Dataflow Testing

This selects test paths according to the locations of definitions and use of variables in a program Aims to
Ensure that the definitions of variables and sub sequent use is tested
First construct a definition-use graph from the control flow of a program
DEF(definition):definition of a variable on the left-hand side of an assignment statement
USE:Computationaluseofavariablelikeread,writeorvariableontherighthandof Assignment statement
Every DU chain be tested at least once.

c) Loop Testing

This focuses on the validity of loop constructs .Four categories can be defined
1.Simple loops
2.Nested loops
3. Concatenated loops

4. Unstructured loops

Testing of simple loops
N is the maximum number of allowable passes through the loop
1. Skip the loop entirely
2. Only one pass through the loop
3. Two passes through the loop
4. M passes through the loop where m>N
5.N-1,N,N+1passestheloop

Validation Testing:

Through Validation testing requirements are validated against s/w constructed. These
are high-order tests where validation criteria must be evaluated to assure that s/w meets all functional,
behaviouralandperformancerequirements.Itsucceedswhenthesoftwarefunctions in a manner that can be reasonably expected by the customer.
1) Validation Test Criteria
2)Configuration Review
3) Alpha And Beta Testing
The validation criteria described in SRS form the basis for this testing. Here, Alpha and Beta testing is performed. Alpha testing is performed at the developers' site beyond users in a natural setting and with a controlled environment. Beta testing is conducted at end-user sites. It is a "live" application and environment is not controlled. End-user records all problems and reports to developer. Developer then makes modifications and releases the product.

System Testing : In system testing, s/w and other system elements are tested as a whole. This is the last high-order testing step which falls in the context of computer system engineering. Software is combined with other system elements like H/W, People, Database and the overall functioning is checked by conducting a series of tests. These tests fully exercise the computer based system. The types of tests are:

1. Recovery testing: Systems must recover from faults and resume processing within a pre specified time.
It forces the system to fail in a variety of ways and verifies that recovery is properly performed. Here the Mean Time To Repair (MTTR) is evaluated to see if it is within acceptable limits.

2. Security Testing: This verifies that protection mechanisms built into a system will protect it from improper penetrations. Tester plays the role of hacker. In reality given enough resources and time it is possible to ultimately penetrate any system. The role of system designer is to make penetration cost more than the value of the information that will be obtained.

3. Stress testing : It executes a system in a manner that demands resources in abnormal quantity, frequency or volume and tests the robustness of the system.

4. Performance Testing: This is designed to test the run-time performance of s/w within the context of an integrated system. They require both h/w and s/w instrumentation.

The Art of Debugging

Debugging occurs as a consequence of successful testing. It is an action that results in the removal of errors.
It is very much an art.

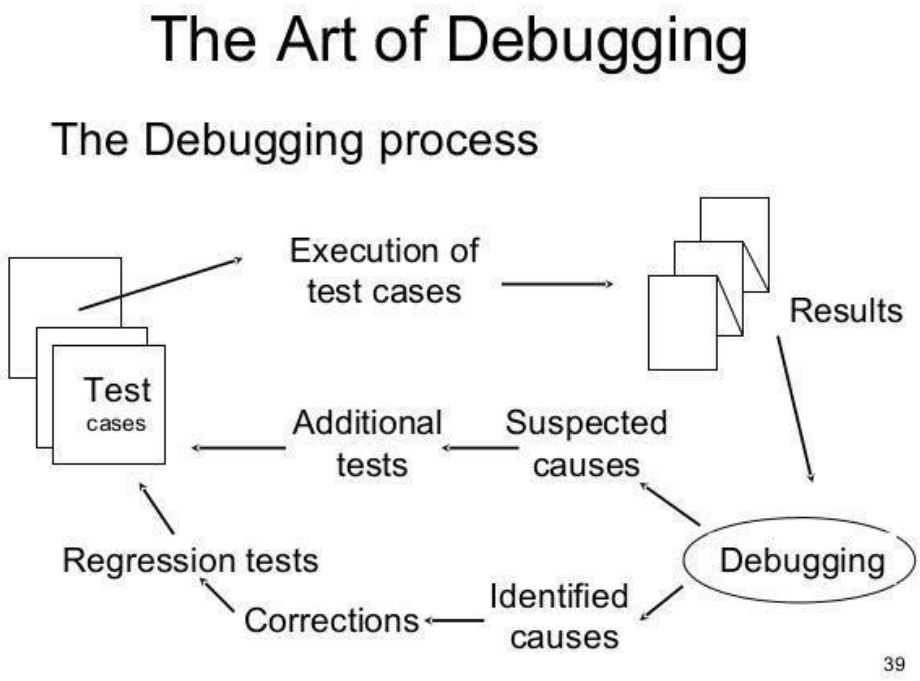


Fig:Debugging process

Debugging has two outcomes:
- Cause will be found and corrected
- Cause will not be found

Characteristics of bugs:
- Symptom and cause can be in different locations

- symptoms may be caused by human error or timing problems

Debugging is an innate human trait . Some are good at it and some

are not.

Debugging Strategies:
The objective of debugging is to find and correct the cause of a software error which is realized by a
Combination of systematic evaluation , intuition and luck. Three strategies are proposed:
1)Brute Force Method.
2)Back Tracking
3)Cause Elimination

Brute Force :Most common and least efficient method for isolating the cause of a s/w error.
This is applied when all else fails. Memory dumps are taken, run-time traces are invoked and program is loaded with output statements. Tries to find the cause from the load of information Leads to waste of time and effort.

Backtracking : Common debugging approach. Useful for small programs.
Beginning at the system where the symptom has been uncovered, the source code is traced backward until the site of the cause is found. More no. of lines implies no. of paths are unmanageable.

Cause Elimination: Based on the concept of Binary partitioning. Data related to error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and data is used to prove or disprove it. A list of all possible causes is developed and tests are conducted to eliminate each

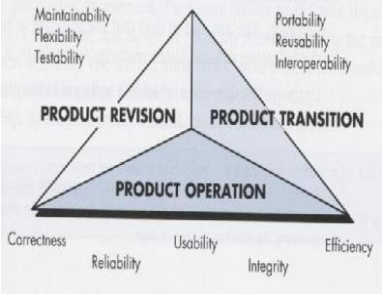
Automated Debugging: This supplements the above approaches with debugging tools that provide semi-automated support like debugging compilers, dynamic debugging aids, test case generators, mapping tools etc.

Regression Testing: When a new module is added as part of integration testing the software changes.
This may cause problems with the functions which worked properly before. This testing is the re-execution of some subset of tests that are already conducted to ensure that changes have not propagated unintended side effects. It ensures that changes do not introduce unintended behavior or errors. This can be done manually or automated. Software Quality Conformance to explicitly stated functional and performance requirements,
Explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software. Factors that affect software quality can be categorized in two broad groups: Factors that can be directly measured (e.g. defects uncovered during testing) Factors that can be measured only indirectly(e.g. usability or maintainability)

McCall's quality factors

1.Product operation Correctness
Reliability
Efficiency
Integrity
Usability
2. Product Revision
Maintainability
Flexibility
3. Product Transition
Portability
Reusability
Interoperability

ISO 9126 Quality Factors



1. Functionality
2. Reliability
3. Usability
4. Efficiency
5. Maintainability
6. Portability

Metrics for Process And Product

Software Measurement:
Software measurement can be categorized as
1) Direct Measure and
2) Indirect Measure

Metrics for Process And Product

Direct Measurement
Direct measure of software process include co-stand effort
Direct measure of product include lines of code , Execution speed, memory size ,defects per reporting time period.

Indirect Measurement

Indirect measure examines the quality of software product itself(e.g.:- Functionality, complexity, efficiency, reliability and maintainability)

Reasons for measurement

To gain base line for comparison with future assessment to determine status with respect to plan.

To predict the size,cost and duration estimate.

To improve the product quality and process improvement.

Software Measurement

The metrics in software Measurement are

Size oriented metrics

Function oriented metrics

Object oriented metrics

Web based application metric

Size Oriented Metrics

It totally concerned with the measurement of software.

A software company maintains a simple record for calculating the size of the software. It includes LOC, Effort,$$,PP document ,Error ,Defect ,People.

Function oriented metrics

Measures the functionality derived by the application

The most widely used function oriented metric is Function point

Function point is independent of programming language Measures functionality from user point of view

Object oriented metric

Relevant for object oriented programming

Based on the following

- Number of scenarios (Similar to use cases)
- Number of key classes
- Number of support classes
- Number of average support class per key class

Number of subsystem

Web based application metric

Metrics related to web based application measure the following

1. Number of static pages(NSP)
2. Number of dynamic pages (NDP) Customization(C) = NSP/NSP+NDP C should approach 1

Metrics for Software Quality

Measuring Software Quality

1. Correctness=defects/KLOC
2. Maintainability=MTTC(Mean-time to change)
3. Integrity=Sigma[1-(threat(1-security))]

Threat : Probability that an attack of specific type will occur with in a given time Security : Probability that an attack of a specific type will be repelled Metrics for Software

Quality Usability: Ease of use Defect Removal Efficiency(DRE)

DRE=E/(E+D)

E is the no. of errors found before delivery and D is no. of defects reported after delivery Ideal value of DRE is 1

UNIT– V

Risk management: Reactive Vs proactive risk strategies, software risks, risk identification, risk projection, risk refinement, RMMM.

Quality Management: Quality concepts, software quality assurance, software reviews, formal technical reviews, statistical software quality assurance, software reliability, the ISO 9000qualitystandards.

Risk Management

Risk is an undesired event or circumstance that occur while a project is underway It is necessary for the project manager to anticipate and identify different risks that a project may be susceptible to Risk Management .It aims at reducing the impact of all kinds of risk that may effect a project by identifying, analyzing and managing them

Reactive Vs Proactive risk

Reactive: It monitors the projects likely risk and resources are set aside.
Proactive: Risk are identified,their probability and impact is accessed

Software Risk

It involve 2 characteristics
Uncertainty : Risk may or may not happen
Loss: If risk is reality unwanted loss or consequences will occur It includes

- Project Risk
- Technical Risk
- Business Risk
- Known Risk
- Unpredictable Risk
- Predictable risk

Project risk: Threaten the project plan and affect schedule and resultant cost
Technical risk: Threaten the quality and time lines of software to be produced
Business risk: Threaten the viability of software to be built
Known risk: These risks can be recovered from careful evaluation
Predictable risk: Risks are identified by past project experience
Unpredictable risk: Risks that occur and may be difficult to identify

Risk Identification

It concerned with identification of risk
Step1: Identify all possible risks
Step2: Create item check list
Step3: Categorize into risk components-Performance risk, cost risk, support risk and schedule risk
Step4: Divide the risk into one of 4 categories
Negligible-0
Marginal-1
Critical-2

Risk Identification
Risk Identification includes
Product size
Business impact Development environment
Process definition Customer characteristics
Technology to be built Staff size and experience

Risk Projection

Also called risk estimation .It estimates the impact of risk on the project and the product.

Estimation is done by using Risk Table. Risk projection addresses risk in 2ways

Risk	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger no. of Users than planned	PS	30%	3	
Less reuse Than planned	PS	70%	2	
End user Resist system	BU	40%	3	

Likelihood or probability that the risk is real(L_i)
Consequences(X_i)

Risk Projection
Steps in Risk projection
1. Estimate L_i for each risk
2. Estimate the consequence X_i
3. Estimate the impact
4. Draw the risk table

Ignore the risk where the management concern is low i.e., risk having impact high or low with low probability of occurrence
Consider all risks where management concern is high i.e., high impact with high or moderate probability of occurrence or low impact with high probability of occurrence

Risk Projection
Projection

The impact of each risk is assessed by Impact values
Catastrophic-1
Critical-2
Marginal-3
Negligible-4

Risk Refinement
Also called Risk assessment
Refines the risk table in reviewing the risk impact based on the following three factors
a. Nature: Likely problems if risk occurs
b. Scope: Just how serious is it?
c. Timing: When and how long

It is based on Risk Elaboration
Calculate Risk exposure
RE=P*C
Where P is probability and C is cost of project if risk occurs Risk Mitigation Monitoring And Management (RMMM)
Its goal is to assist project team in developing a strategy for dealing with risk There are three issues of RMMM
1) Risk Avoidance
2) Risk Monitoring and
3) Risk Management

Risk Mitigation Monitoring And Management(RMMM)
Risk Mitigation
Proactive planning for risk avoidance
Risk Monitoring Assessing whether predicted risk occur or not Ensuring risk a version steps are being properly applied
Collection of information for future risk analysis Determine which risks caused which problems
Risk Mitigation Monitoring And Management(RMMM)
Risk Management
Contingency planning
Actions to be taken in the event that mitigation step have failed and the risk has

become a live problem

Devise RMMP(Risk Mitigation Monitoring And Management Plan)

RMMM plan

It documents all work performed as a part of risk analysis.
Each risk is documented individually by using a Risk Information Sheet.
RIS is maintained by using a database system Quality Management

Quality Concepts

Variation control is the heart of quality control
Form one project to another ,we want to minimize the difference between the predicted resources needed to complete a project and the actual resources used, including staff, equipment, and calendar time
Quality of design
Refers to characteristics that designers specify for the end product Quality Management
Quality of conformance
Degree to which design specifications are followed in manufacturing the product
Quality control
Series of inspections, reviews, and tests used to ensure conformance of a work product to its specifications
Quality assurance
Consists of a set of auditing and reporting functions that assess the effectiveness and completeness of quality control activities

Cost of Quality

Prevention costs
Quality planning, formal technical reviews ,test equipment ,training
Appraisal costs
In-process and inter-process inspection, equipment calibration and maintenance, testing Failure costs
Rework ,repair ,failure mode analysis External failure costs
Complaint resolution ,product return and replacement ,help line support, warranty work

Software Quality Assurance

Software quality assurance (SQA) is the concern of every software engineer to reduce cost and improve product time-to-market.

A Software Quality Assurance Plan is not merely another name for a test plan, though test plans are

Included in an SQA plan.
SQA activities are performed on every software project.

Use of metrics is an important part of developing a strategy to improve the quality of both software processes and work products.

Software Quality Assurance

Definition of Software Quality serves to emphasize:
Conformance to software requirements is the foundation from which software quality is measured.

Specified standards are used to define the development criteria that are used to guide the manner in which software is engineered.
Software must conform to implicit requirements(ease of use, maintainability ,reliability, etc.) as well as its explicit requirements.

SQA Activities

Prepare SQA plan for the project.

Participate in the development of the project's software process description.

Review software engineering activities to verify compliance with the defined software process.

Audit designated software work products to verify compliance with those defined as part of the software process.

Ensure that any deviations in software or work products are documented and handled according to a documented procedure.

Record any evidence of noncompliance and reports them to management.

Software Reviews

Purpose is to find errors before they are passed onto an other software engineering activity or released to the customer.

Software engineers(and others)conduct formal technical reviews(FTRs)for software quality assurance.

Using formal technical reviews(walk through or inspections) is an effective means for improving software quality.

Formal Technical Review

AFTR is a software quality control activity performed by software engineers and others. The objectives are:

To uncover errors in function, logic or implementation for any representation of the software.

To verify that the software under review meets its requirements.

To ensure that the software has been represented according to predefined standards. To achieve software that is developed in a uniform manner and

To make projects more manageable.

Review meeting in FTR

The Review meeting in a FTR should abide to the following constraints Review meeting members should be between three and five.
Every person should prepare for the meeting and should not require more than two hours of work for each person.
The duration of the review meeting should be less than two hours.

The focus of FTR is on a work product that is requirement specification ,a detailed component design, a source code listing for a component.

The individual who has developed the work product i.e, the producer informs the project leader that the work product is complete and that a review is required.

The project leader contacts a review leader, who evaluates the product for readiness, generates copy of product material and distributes them to two or three review members for advance preparation .

Each reviewer is expected to spend between one and two hours reviewing the product, making notes

There view leader also reviews the product and establish an agenda for the review meeting The review meeting is attended by review leader, all reviewers and the producer.

One of the reviewer act as a recorder, who notes down all important points discussed in the meeting.

The meeting(FTR) is started by introducing the agenda of meeting and then **the producer introduces his product. Then the producer "walkthrough" the** product, the reviewers raise issues which they have prepared in advance.

If errors are found there coder notes down

Review reporting and Record keeping

During the FTR, are viewer(recorder)records all issues that have been raised A review summary report answers three questions

What was reviewed?

Who reviewed it?

What were the findings and conclusions?

Review summary report is a single page form with possible attachments

There view issues list serves two purposes To identify problem areas in the product.

To serve as an action item check list that guides the producer as corrections are made.

Review Guidelines

Review the product ,not the producer

Set an agenda and maintain it

Limit debate and rebuttal

Enunciate problem areas, but don't attempt to solve **every problem** noted

Take return notes

Limit the number of participants and insist upon advance preparation. Develop a checklist for each product i.e likely to be reviewed.

Allocate resources and schedule time for FTRS

Conduct meaningful training for all reviewer

Review your early reviews

Software Defects

Industry studies suggest that design activities introduce 50-65% of all defects or errors during the software process

Review techniques have been shown to be up to 75% effective in uncovering design flaws which ultimately reduces the cost of subsequent activities in the software process

Statistical Software Quality Assurance

Information about software defects is collected and categorized. Each defect is traced back to its cause

Using the Pareto principle(80%ofthedefectscanbetracedto 20%ofthecauses) isolate the "vital few" defect causes.

Move to correct the problems that caused the defects in the "vital few"

Six Sigma for Software Engineering

The most widely used strategy for statistical quality assurance

Three core steps:

1. Define customer requirements, deliverables, and project goals via well-defined methods of customer communication.
2. Measure each existing process and its output to determine current quality performance (e.g., compute defect metrics)
3. Analyzed effect metrics and determine vital few causes.

For an existing process that needs improvement

1. Improve process by eliminating the root causes for defects
2. Control future work to ensure that future work does not reintroduce causes of defects

If new processes are being developed

1. Design each new process to avoid root causes of defects and to meet customer requirements
2. Verify that the process model will avoid defects and meet customer requirements

Software Reliability

Defined as the probability of failure free operation of a computer program in a specified environment for a specified time period

Can be measured directly and estimated using historical and developmental data

Software reliability problems can usually be traced back to errors in design or implementation.

Measures of Reliability

Mean time between failure(MTBF)=MTTF+MTTR

MTTF = mean time to failure

MTTR=mean time to repair

Availability= [MTTF/(MTTF+ MTTR)]x 100%

ISO 9000 Quality Standards

ISO (International Standards Organization) is a group or consortium of 63 countries established to plan and fosters standardization. ISO declared its 9000 series of standards in 1987. It serves as a reference for the contract between independent parties. The ISO 9000 standard determines the guidelines for maintaining a quality system. The ISO standard mainly addresses operational methods and organizational methods such as responsibilities, reporting, etc. ISO 9000 defines a set of guidelines for the production process and is not directly concerned about the product itself.

Types of ISO 9000 Quality Standards

The ISO 9000 series of standards is based on the assumption that if a proper stage is followed for production ,then good quality products are bound to follow automatically .The types of industries to which the various ISO standards apply are as follows.

1. **ISO9001:** This standard applies to the organizations engaged indesign,development, production, and servicing of goods. This is the standard that applies to most software development organizations.
2. **ISO 9002:** This standard applies to those organizations which do not design products but are only involved in the production. Examples of these category industries contain steel and car manufacturing industries that buy the product and plants designs from external sources and are engaged in only manufacturing thoseproducts.Therefore,ISO9002doesnot apply to software development organizations.
3. **ISO9003:** This standard applies to organizations that are involved only in the installation and testing of the products. For example, Gas companies.

An organization determines to obtain ISO9000 certification appliesto ISO registrar office for registration. The process consists of the following stages:

1. Application: Once an organization decided to go for ISO certification, it applies to the registrar for registration.
2. Pre-Assessment: During this stage ,the registrar makes a rough assessment of the organization.
3. Document review and Adequacy of Audit: During this stage ,the registrar reviews the document submitted by the organization and suggest an improvement.
4. Compliance Audit: During this stage ,the registrar checks whether the organization has compiled the suggestion made by it during the review or not.
5. Registration: The Registrar awards the ISO certification after the successful completion of all the phases.

6.**Continued Inspection:**The registrar continued to monitor the organization time bytime.