DIGITAL NOTES

ON

SOFTWARE ENGINEERING

[R22A0505]

B.TECH II YEAR – I SEM

(2023-2024)

PREPARED BY
K.SWETHA
K.N.KOUSHIL REDDY

DEPARTMENT OF INFORMATION TECHNOLOGY

MALLA REDDY COLLEGE OF ENGINEERING &TECHNOLOGY
(Autonomous Institution – UGC, Govt. of India)
(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015Certified) Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, INDIA.

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

II Year B. TECH IT – 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 9000 quality standards.

TEXTBOOKS:

1. Software Engineering, A practitioner’s Approach-RogerS.Pressman,6thedition, McGraw-Hill International Edition.

2. Software Engineering-Sommerville,7thedition, 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

INDEX

UNIT - I

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.

THE CHANGING NATURE OF SOFTWARE
The various categories of software are
1. System software
2. Application software
3. Engineering and scientific software
4. Embedded software
5. Product-line software
6. Web-applications
7. 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 forthe consumer and industrial markets.
Artificial intelligence software. Artificial intelligence (AI) software makes use of nonnumeric algorithms to solve complex problems that are not amenable to computation or straightforward analysis
Engineering and scientific software. Engineering and scientific software have been characterized by "number crunching" algorithms.

Software Myths

Software myths are preconceived notions about software and its creation that people hold to be true but are in fact untrue. Professionals in Software Engineering have now identified the software myths that have persisted throughout the years.

These fallacies are common knowledge to managers and software developers. However, it might be challenging to change old behaviours.

Types of Software Myths

There are three kinds of software myths that are busted down in the article.

- Management Myths
- Customer Myths
- Practitioner's Myths

Management Myths

Managers are often under pressure for software development under a tight budget, improved quality, and a packed schedule, often believing in some software myths. Following are some management myths.

Myth 1

Manuals containing simple procedures, principles, and standards are enough for developers to acquire all the information they need for software development.

Reality 1

Standards discussed in modules are often outdated, inadaptable, and incomplete. Not all the standards in the manual are known to developers as not all means tend to decrease delivery time and maintain high quality. Most of the time, developers are unaware of these standards.

Myth 2

Falling behind on schedule could be taken care of by adding more programmers.

Reality 2

Adding more human resources to already late projects worsens the problem. Developers working on the project have to educate the newcomers, further delaying the project. Also, newcomers are far less productive than developers already working on them. As a result, time spent on educating newcomers could not meet the immediate reduction in work.

Myth 3

If a project is outsourced to a third party, we could just relax and wait for them to build it.

Reality 3

If an organisation is not able to manage and control software projects internally, then the organisation will suffer invariably when they outsourced the project.

Customer Myths

Customer Myths are generally due to false expectations by customers, and these myths end up leaving customers with dissatisfaction with the software developers. Following are some customer myths.

Myth 1

Not only detailed conditions a vague collection of software objectives is enough to begin programming with.

Reality 1

If the objectives of software are vague enough to become ambiguous, then it's inevitable that software will not do what the customer wants. Often when software development starts without a complete picture
in mind,
 it results in software failure.

Myth 2

Softwares are flexible, and developers could accommodate any change later. Developers can quickly take care of these changes in requirements.

Reality 2

Longer the time for which software has proceeded for development, it becomes more and more difficult to accommodate any changes. Any change causes an increase in additional costs because incorporating changes at later stages needs redesigning and extra resources.

Practitioner's Myths

Developers often work under management pressure to complete software within a timeframe, with fewer resources often believing in these software myths. Following are some practitioners' myths.

Myth 1

Once the software is developed or the code is delivered to the customer, the developer's work ends.

Reality 1

A significant chunk of developers' work, i.e., 50-60 % of all the efforts expended on software, will be spent after the customer provides the software. Major requirements would get found missing, and new bugs may get discovered, and so on.

Myth 2

Software testing could only be possible when the software program starts running.

Reality 2

Quality of software could be measured at any phase of development by applying some QA mechanism.

Myth 3

Unnecessary Documentation slows down the process of software development.

Reality 3

Software engineering is about creating a quality product at every level and not about adding unnecessary work. Proper documentation of software helps us guide the user and enhance the quality, which reduces the amount of rework.

SOFTWARE APPLICATIONS

System Software –
System Software is necessary to manage the computer resources and support the execution of application programs. Software like operating systems, compilers, editors and drivers, etc., come under this category. A computer cannot function without the presence of these. Operating systems are needed to link the machine-dependent needs of a program with the capabilities of the machine on which it runs. Compilers translate programs from high-level language to machine language.

Application Software –
Application software is designed to fulfill the user's requirement by interacting with the user directly. It could be classified into two major categories:- generic or customized. Generic Software is the software that is open to all and behaves the same for all of its users. Its function is limited and not customized as per the changing requirements of the user. However, on the other hand, Customized software the software products which are designed as per the client's requirement, and are not available for all.

Networking and Web Applications Software –
Networking Software provides the required support necessary for computers to interact with each other and with data storage facilities. The networking software is also used when software is running on a network of computers (such as the World Wide Web). It includes all network management software, server software, security and encryption software, and software to develop web-based applications like HTML, PHP, XML, etc.

Embedded Software –
This type of software is embedded into the hardware normally in the Read-Only Memory (ROM) as a part of a large system and is used to support certain functionality under the control conditions. Examples are software used in instrumentation and control applications like washing machines, satellites, microwaves, etc.

Reservation Software –
A Reservation system is primarily used to store and retrieve information and perform transactions related to air travel, car rental, hotels, or other activities. They also provide access to bus and railway reservations, although these are not always integrated with the main system. These are also used to relay computerized information for users in the hotel industry, making a reservation and ensuring that the hotel is not overbooked.

Business Software –
This category of software is used to support business applications and is the most widely used category of software. Examples are software for inventory management, accounts, banking, hospitals, schools, stock markets, etc.

Entertainment Software –
Education and entertainment software provides a powerful tool for educational agencies, especially those that deal with educating young children. There is a wide range of entertainment software such as computer games, educational games, translation software, mapping software, etc.

Artificial Intelligence Software –
Software like expert systems, decision support systems, pattern recognition software, artificial neural networks, etc. come under this category. They involve complex problems which are not affected by complex computations using non-numerical algorithms.

Scientific Software –
Scientific and engineering software satisfies the needs of a scientific or engineering user to perform enterprise-specific tasks. Such software is written for specific applications using principles, techniques, and formulae specific to that field. Examples are software like MATLAB, AUTOCAD, PSPICE, ORCAD, etc.

Utilities Software –
The programs coming under this category perform specific tasks and are different from other software in terms of size, cost, and complexity. Examples are anti-virus software, voice recognition software, compression programs, etc.

Document Management Software –
Document Management Software is used to track, manage and store documents in order to reduce the paperwork. Such systems are capable of keeping a record of the various versions created and modified by different users (history tracking). They commonly provide storage, versioning, metadata, security, as well as indexing and retrieval capabilities.

A GENERIC VIEW OF PROCESS SOFTWARE ENGINEERING-A LAYERED TECHNOLOGY

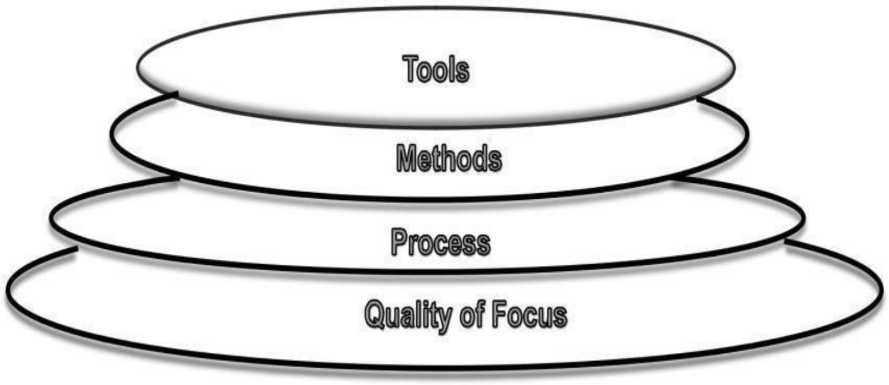


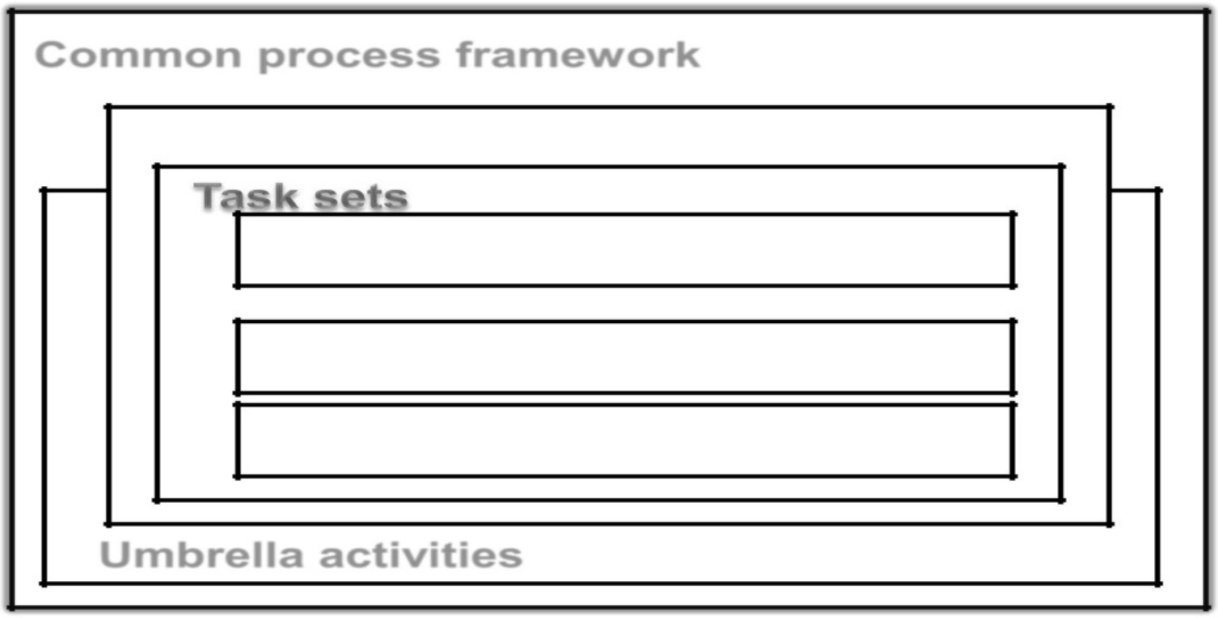
Fig: Software Engineering-A layered technology

SOFTWARE ENGINEERING - A LAYERED TECHNOLOGY

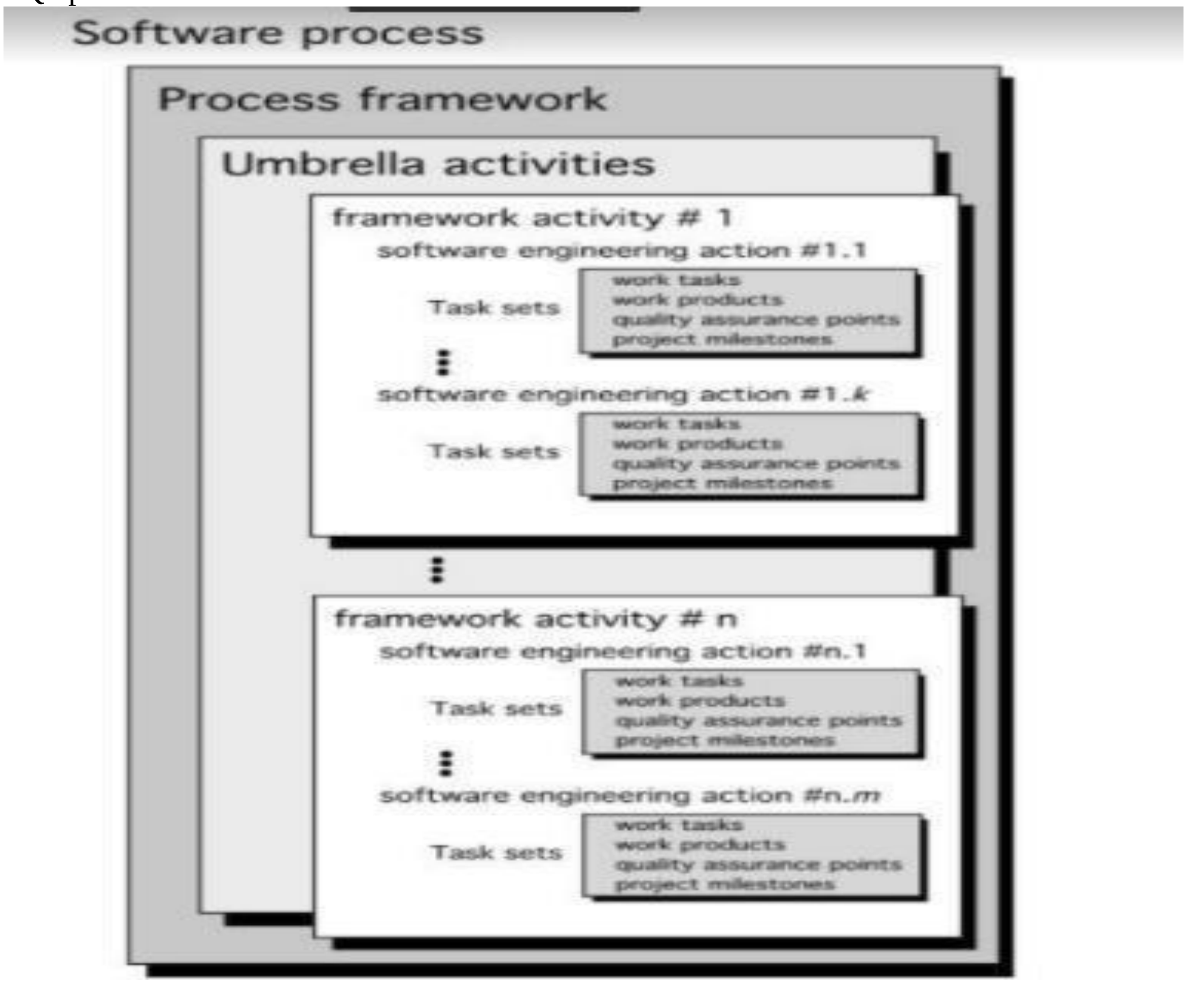
- Quality focus - Bedrock that supports Software Engineering.
- Process - Foundation for software Engineering
- Methods - Provide technical How-to’s for building software
- Tools - Provide semi-automatic and automatic support to methods

A PROCESS FRAMEWORK

- Establishes the foundation for a complete software process
- Identifies a number of framework activities applicable to all software projects
- Also include a set of umbrella activities that are applicable across the entire software process.



A PROCESS FRAMEWORK :
Common process framework Umbrella activities Framework activities Tasks, Milestones, deliverables SQApoints.



A PROCESS FRAMEWORK

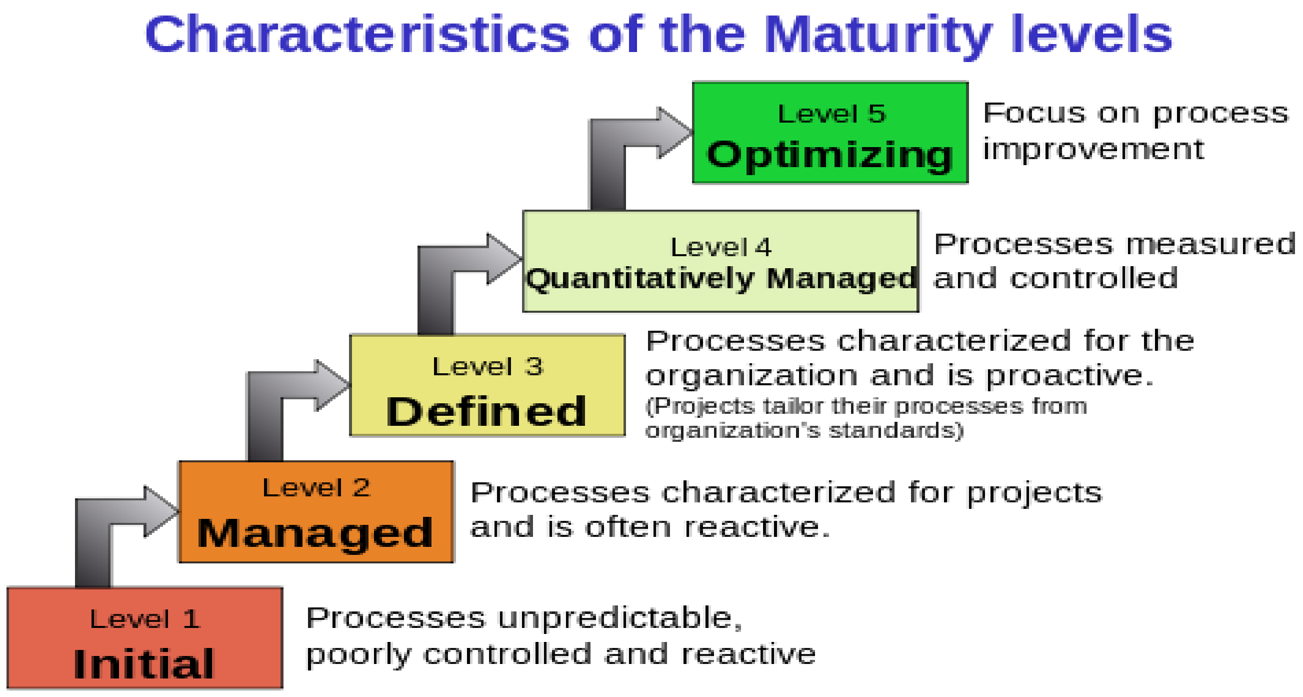
Used as a basis for the description of process models Generic process activities
- Communication
- Planning
- Modeling
- Construction
- Deployment

A PROCESS FRAMEWORK
Generic view of engineering complimented by a number of umbrella activities

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.


Characteristics of the Maturity levels
Level 5
Optimizing
Focus on process improvement
Level 4
Quantitatively Managed
Processes measured and controlled
Level 3
Defined
Processes characterized for the organization and is proactive.
(Projects tailor their processes from organization's standards)
Level 2
Managed
Processes characterized for projects and is often reactive.
Level 1
Initial
Processes unpredictable, poorly controlled and reactive

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

Continuous model:
-Lets organization select specific improvement that best meet its business objectives and minimize risk-Levels are called capability levels.
-Describes a process in 2 dimensions
-Each process area is assessed against specific goals and practices and is rated according to the following capability levels.

Six levels of CMMI

- Level 0:Incomplete
- Level 1:Performed
- Level 2:Managed
- Level 3:Defined
- Level 4:Quantitatively managed
- Level 5:Optimized

- 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 feed back from the user
  -Use of innovative ideas and techniques, statistical quality control 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 PATTERNS
Software Process is defined as collection of Patterns. Process pattern provides a template. It comprises of
• Process Template
-Pattern Name
-Intent
-Types
-Task pattern
- Stage pattern
-Phase Pattern
• Initial Context
• Problem
• Solution
• Resulting Context
• Related Patterns

PROCESS ASSESSMENT
Does not specify the quality of the software or whether the software will be delivered on time or will it stand up to the user requirements. It attempts to keep a check on the current state of the software process with the intention of improving it.
PROCESS ASSESSMENT Software Process Software Process Assessment Software Process Improvement Motivates Capability determination.

APPROACHES TO SOFTWARE ASSESSMENT
• Standard CMMI assessment (SCAMPI)
• CMM based appraisal for internal process improvement
• SPICE(ISO/IEC 15504)

ISO 9001:2000 for software Personal and Team Software Process Personal software processPLANNING
HIGH LEVEL DESIGN
HIGH LEVEL DESIGN REVIEW
DEVELOPMENT
POSTMORTEM

PERSONAL AND TEAM SOFTWARE PROCESS

PERSONAL SOFTWARE PROCESS:
The personal software process is focused on individuals to improve their performance. The PSP is an individual process, and it is a bottom-up approach to software process improvement. The PSP is a prescriptive process, it is a more mature methodology with a well-defined set of tools and techniques.

Key Features of PSP :

- **Process-focused:** PSP is a process-focused methodology that emphasizes the importance of following a disciplined approach to software development.
- **Personalized:** PSP is personalized to an individual’s skill level, experience, and work habits. It recognizes that individuals have different strengths and weaknesses, and tailors the process to meet their specific needs.
- **Metrics-driven:** PSP is metrics-driven, meaning that it emphasizes the collection and analysis of data to measure progress and identify areas for improvement.

- **Incremental:** PSP is incremental, meaning that it breaks down the development process into smaller, more manageable pieces that can be completed in a step-by-step fashion.

TEAM SOFTWARE PROCESS

TSP is a team-based process. It is focused on team productivity. Basically, it is a top-down approach. The TSP is an adaptive process, and process management methodology.

Key Features of TSP :

- **Team-focused:** TSP is team-focused, meaning that it emphasizes the importance of collaboration and communication among team members throughout the software development process.
- **Process-driven:** TSP is process-driven, meaning that it provides a structured approach to software development that emphasizes the importance of following a disciplined process.
- **Metrics-driven:** TSP is metrics-driven, meaning that it emphasizes the collection and analysis of data to measure progress, identify areas for improvement, and make data-driven decisions.
- **Incremental:** TSP is incremental, meaning that it breaks down the development process into smaller, more manageable pieces that can be completed in a step-by-step fashion.
- **Quality-focused:** TSP is quality-focused, meaning that it emphasizes the importance of producing high-quality software that meets user requirements and is free of defects.
- **Feedback-oriented:** TSP is feedback-oriented, meaning that it emphasizes the importance of receiving feedback from peers, mentors, and other stakeholders to identify areas for improvement

PROCESS MODELS

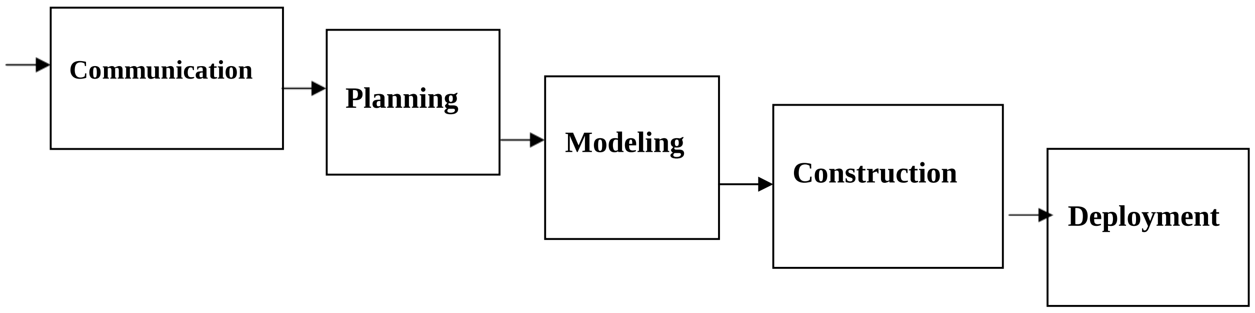
- Help in the software development
- Guide the software team through a set of framework activities
- Process Models may be linear, incremental or evolutionary

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 leveland progresses through analysis, design, code and testing

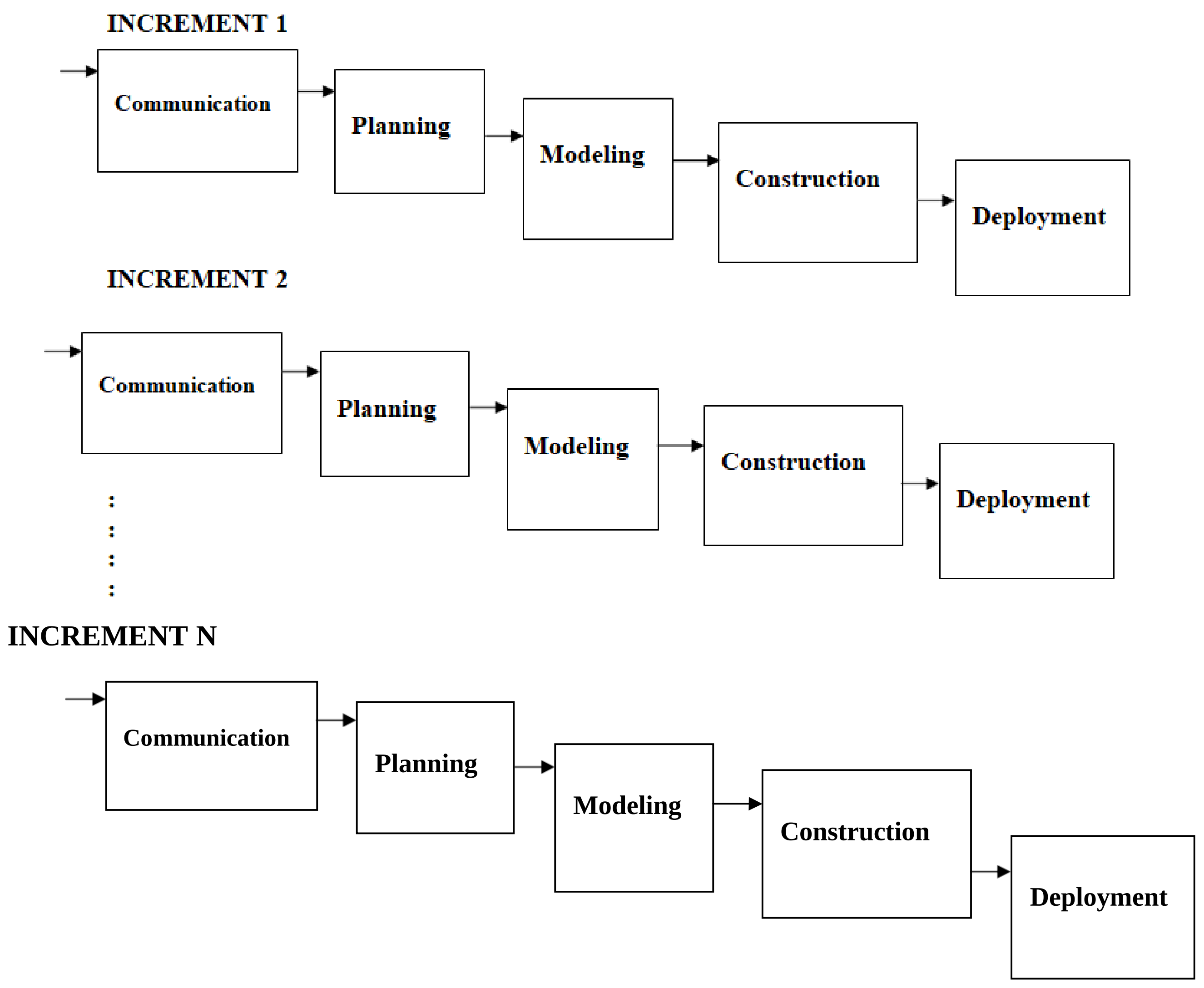


PROBLEMS IN WATER FALLMODEL

- Real projects rarely follow the 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 INCREMENTAL PROCESS MODEL

- Linear sequential model is not suited for projects which are iterative in nature
- Incremental model suits such projects
- Used when initial requirements are reasonably well-defined and compelling need to provide limited functionality.
- Functionality expanded further in later releases
  Communication
  Planning
  Modeling
  Construction
  Deployment



- Software releases in increments
- 1st increment constitutes Core product
- Basic requirements are addressed
- Core product undergoes detailed evaluation by the customer
- As a result, plan is developed for the next increment. Plan addresses the modification of core productto better meet the needs of customer
- Process is repeated until the complete product is produced

EVOLUTIONARY PROCESSMODEL

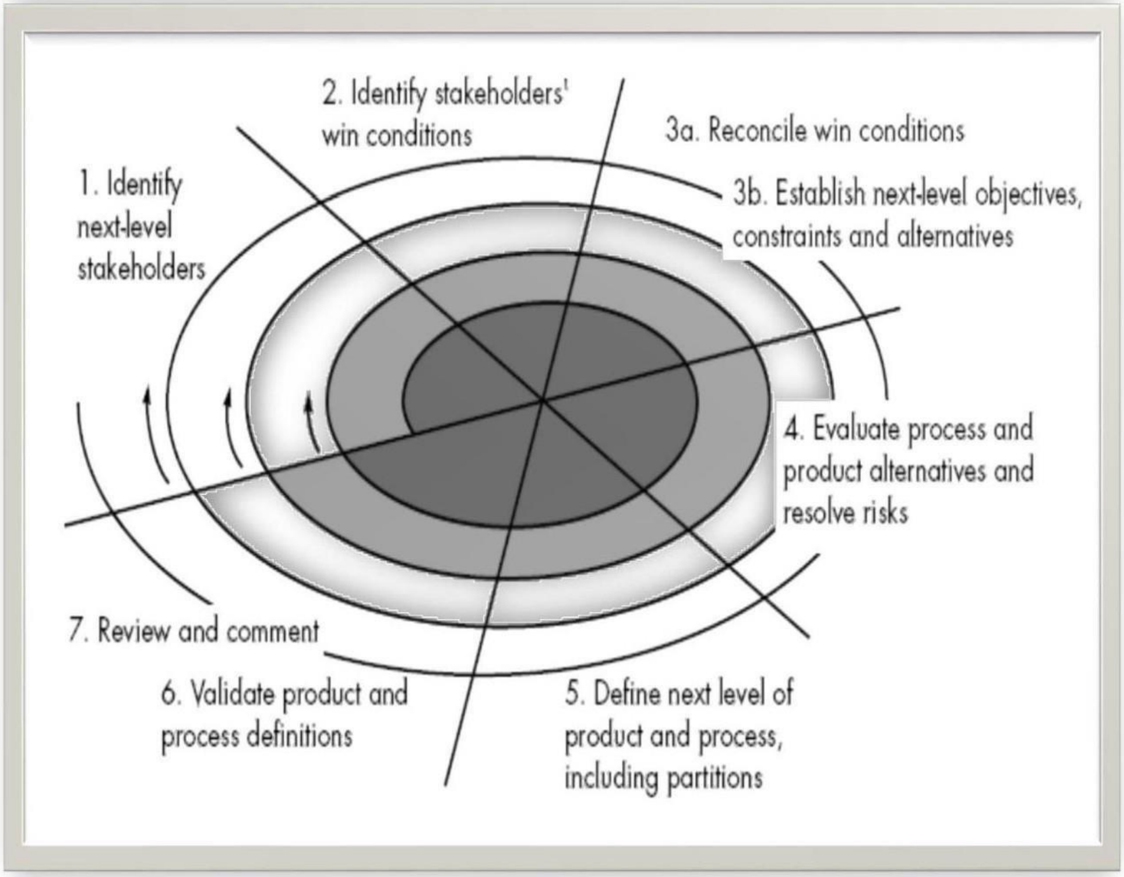
- Software evolves over a period of time
- Business and product requirements often change as development proceeds making a straight-line path to an end product unrealistic
- Evolutionary models are iterative and as such are applicable to modern day applications

Types of evolutionary models
– Prototyping
– Spiral model
– Concurrent development model

– human/machine interaction

THE SPIRAL MODEL

An evolutionary model which combines the best feature of the classical life cycle 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


2. Identify stakeholders' win conditions
3a. Reconcile win conditions
1. Identify next-level stakeholders
3b. Establish next-level objectives, constraints and alternatives
4. Evaluate process and product alternatives and resolve risks
7. Review and comment
6. Validate product and process definitions
5. Define next level of product and process, including partitions

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
- Subsequent circuits develop a prototype
- sophisticated version of software

Does not focus on flexibility and extensibility (more emphasis on high quality)

- Requirement is balance between high quality and flexibility and extensibility

Agile Methodology

What is the Agile methodology? The Agile methodology is a project management approach that involves breaking the project into phases and emphasizes continuous collaboration and improvement. Teams follow a cycle of planning, executing, and evaluating

gile software development methodologies develop software in a small period and allow incorporating changes in the software while developing it. Our software development company uses agile methods as per the user requirements to deliver the best software.

It contains six phases: concept, inception, iteration, release, maintenance, and retirement. The Agile life cycle will vary slightly depending on the project management methodology chosen by a team.

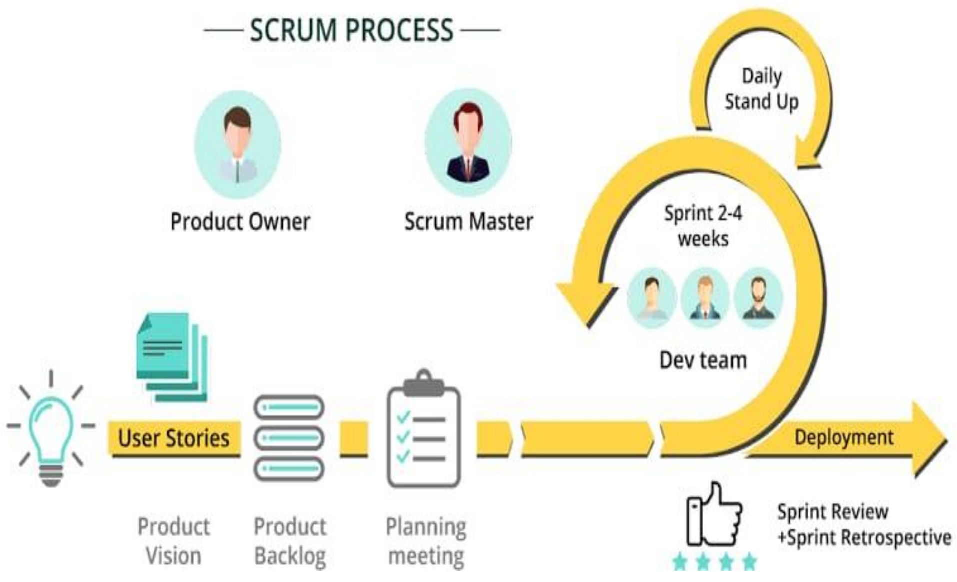
gile development is important because it helps to ensure that development teams complete projects on time and within budget. It also helps to improve communication between the development team and the product owner. Additionally, Agile development methodology can help reduce the risks associated with complex projects.

xamples of Agile Methodology. The most popular and common examples are Scrum, eXtreme Programming (XP), Feature Driven Development (FDD), Dynamic Systems Development Method (DSDM), Adaptive Software Development (ASD), Crystal, and Lean Software Development (LSD).

Main Agile methodologies:

1. Scrum

Scrum is, undoubtedly, the most used of the many frameworks underpinning Agile methodology. Scrum is characterised by cycles or stages of development, known as sprints, and by the maximisation of development time for a software product towards a goal, the Product Goal. This Product Goal is a larger value objective, in which sprints bring the scrum team product a step closer.



It is usually used in the management of the development of software products but can be used successfully in a business-related context.

Every day starts with a small 15-minute meeting, the daily Scrum, which takes the role of synchronising activities and finding the best way to plan out the working day, allowing for a check on sprint "health" and product progress.

Advantages:

Team motivation is good because programmers want to meet the deadline for every sprint;

Transparency allows the project to be followed by all the members in a team or even throughout the organisation;

A simple "definition of done" is used for validating requirements

Focus on quality is a constant with the scrum method, resulting in fewer mistakes;

The dynamics of this method allow developers to reorganise priorities, ensuring that sprints that have not yet been completed get more attention;

Good sprint planning is prioritised, so that the whole scrum team understands the “why, what and how” of allocated tasks.

Disadvantages:

The segmentation of the project and the search for the agility of development can sometimes lead the team to lose track of the project as a whole, focusing on a single part;

Every developer role may not be well defined, resulting in some confusion amongst team members.

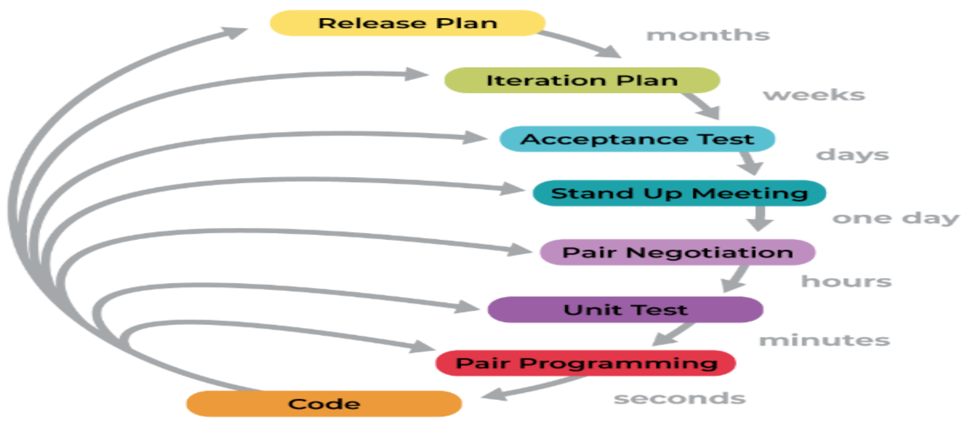
2. Extreme Programming (XP)

This is a typical Agile development framework, developed by Kent Beck, and can be adapted to development companies of various dimensions.

Extreme Programming (“XP”) methodology is based around the idea of discovering “the simplest thing that will work” without putting too much weight on the long-term product view.

It is a methodology that emphasises values such as Communication, Simplicity, Feedback, Courage and Respect, and prioritises customer satisfaction over everything else. This methodology encourages trust by motivating developers to accept changes in customer requirements, even if they arrive during the latter stages of the development cycle.

Teamwork is extremely important in XP, since, when there is a problem, it is solved by the whole team of managers, developers or customers, bringing them together to promote conversation and engagement and break down barriers to communication. They all become essential pieces of the same puzzle, creating a fertile environment for high productivity and efficiency within teams. In Extreme Programming, the software is tested from day one, collecting feedback to improve development. XP promotes activities such as pair programming, and with a strong testing component, it’s an excellent engineering methodology.



Advantages:

The simplicity of the written code is an advantage since it allows for improvement at any given time;

The whole process and the XP development cycle is visible, creating goals for developers along with relatively rapid results;

Software development is more agile than when using other methodologies, due to constant testing;

Promotes a highly energising way of working;

XP also contributes to uplifting and maintaining team talent.

Disadvantages:

The extreme focus on code can lead to less importance being paid to design, meaning that it has to get extra attention later;

This framework may not work at its best if all the team members are not situated in the same geographical area;

In XP projects, a registry of possible errors is not always maintained, and this lack of monitoring can lead to similar bugs in the future.

UNIT-II

SOFTWARE REQUIREMENTS

IEEE defines Requirement as :

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 tosatisfy constract, 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

• Nonfunctional 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

FUNCTIONAL REQUIREMENTS

• 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 Requirements1.

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
-

PROBLEMS FACED USING THE NATURAL LANGUAGE
1. Lack of clarity-- Leads to misunderstanding because of ambiguity of natural language
2. Confusion-- Due to over flexibility, sometime 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. 2.Data structures--Provide data passing from one sub-system to another
3. 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. Itis 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

Suggests that there are 6 requirements 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 forethought about the life cycle 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.IntroductionPurpose
Scope
Definitions, Acronyms and Abbreviations
ReferencesOverview
4. General description Product perspective Product function summary User characteristics General constraints
Assumptions and dependencies
5. Specific Requirements
- Functional requirements

6. External interface requirements
- Performance requirements
- Design constraints
- Attributes eg. security, availability, maintainability, transferability/conversion
- Other requirements

• Appendices
• Index

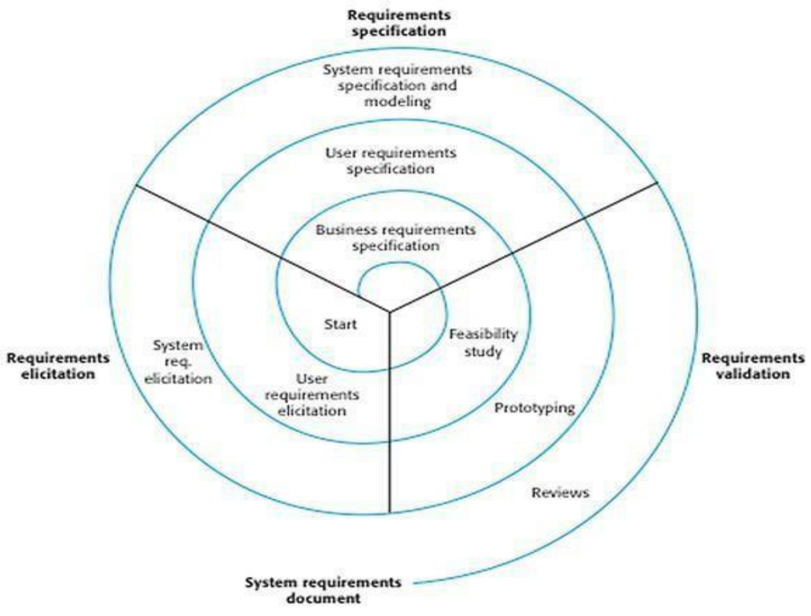
REQUIREMENTS ENGINEERING PROCESS
To create and maintain a system requirement document. The overall process includes four high level requirements engineering sub-processes:
1.Feasibility study
--Concerned with assessing whether the system is useful to the business
2.Elicitation and analysis
--Discovering 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: 1.Requirements elicitation

1. Requirements specification
2. Requirements validation



Requirements specification
System requirements specification and modeling
User requirements specification
Business requirements specification
Start
Feasibility study
Requirements elicitation
System req. elicitation
User requirements elicitation
Requirements validation
Prototyping
Reviews
System requirements document

FEASIBILITY STUDIES

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 within given cost and schedule
3. Can the system be integrated with other system which are 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

1. Stakeholder often don’t know what they want from the computer system.
2. Stakeholder expression of requirements in natural language is sometimes difficult to Understand.
3. Different stakeholders express requirements differently
4. 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
5. The spiral representation of Requirements Engineering

REQUIEMENTS DICOVERY TECHNIQUES

1. View points --Based on the viewpoints expressed by the stake holder
--Recognizes multiple perspectives and provides a framework for discovering conflicts in the requirements proposed by different stakeholders
Three Generic types of viewpoints
1. Interactor viewpoint--Represents people or other system that interact directly with the system
2. Indirect viewpoint--Stakeholders who influence the requirements, but don’t use the system
3. Domain viewpoint--Requirements domain characteristics and constraints that influence the requirements.

2. Interviewing--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 stakeholders answer a 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 interviewee to start discussion with a question or a proposal
3. 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 on to 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 the resubmitted form is still incorrect 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.

4. 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

REQUIREMENTS VALIDATION
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
1. Validity checks --Verification that the system performs the intended function by the user
2.Consistency check --Requirements should not conflict
3.Completeness checks --Includes requirements which define all functions and constraints intended by the system user
4.Realism checks --Ensures that the requirements can be actually implemented
5.Verifiability -- Testable to avoid disputes between customer and developer.

VALIDATION TECHNIQUES
1. REQUIREMENTS REVIEWS
Reviewers check the following:
(a) Verifiability: Testable
(b) Comprehensibility
(c) Traceability
(d) Adaptability
2. PROTOTYPING
3. 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
(a) **Enduring requirements:** Relatively stable requirements
(b) **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
1. Requirements identification -- Each requirement must have unique tag for cross reference and traceability
2. Change management process -- Set of activities that assess the impact and cost of changes
3. Traceability policy -- A matrix showing links between requirements and other elements of software development

4.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 traceability 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 analysed 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

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.

QUALITY GUIDELINES
• Uses recognizable architectural styles or patterns
• Modular; that is logically partitioned into elements or subsystems
• Appropriate data structures for the classes to be implemented
• Independent functional characteristics for components
• Repeatable method
QUALITY ATTRIBUTES
• Functionality
* Feature set and capabilities of programs
* Security of the overall system
• Usability
* user-friendliness
* Documentation
• Reliability
* Evaluated by measuring the frequency and severity of failure
• 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

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

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 and p2. If the complexity of p1 iscp1 and of p2 is cp2 then effort to solve p1=cp1 and effort to 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.

FUNCTIONAL INDEPENDENCE
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 b 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 -- Refinement of the analysis classes that identity attributes and services to implement some of business domain

3. Process class -- implements lower level business abstractions required to fully manage the business domain classes
4. Persistent class -- Represent data stores that will persist beyond the execution of the software
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.

Architectural design elements. Derived from three sources
(1) Information about the application domain of the software
(2) Analysis model such as dataflow diagrams or analysis classes.
(3) Architectural pattern and styles Interface Design elements Set of detailed drawings constituting: (1) User interface
(2) External interfaces to other systems, devices etc
(3) Internal interfaces between various components

CREATING AN ARCHITECTURAL 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.

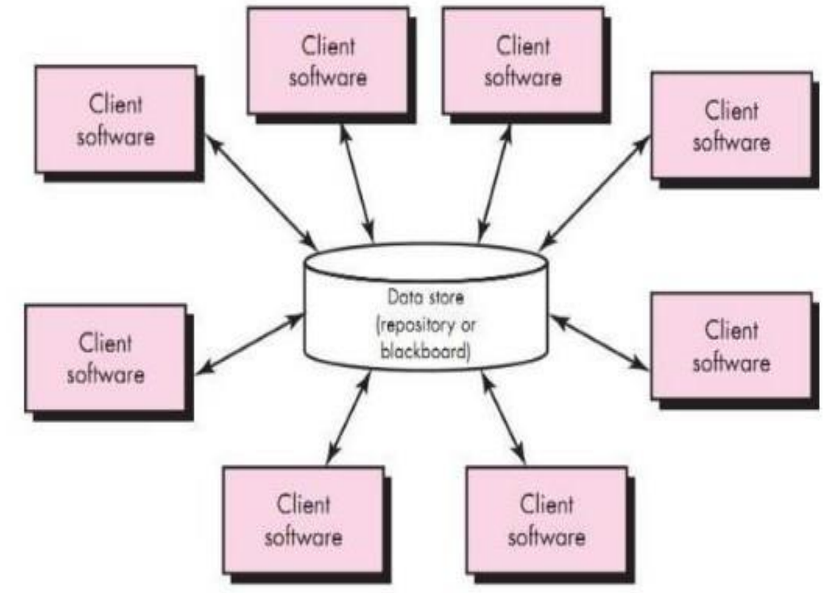
DATA DESIGN AT ARCHITECTURE LEVEL
- Data structure at programming level
- Data base at application level
- Data warehouse at business level.

DATA DESIGN AT COMPONENT LEVEL
Principles for data specification:
1. Proper selection of data objects and data and data models

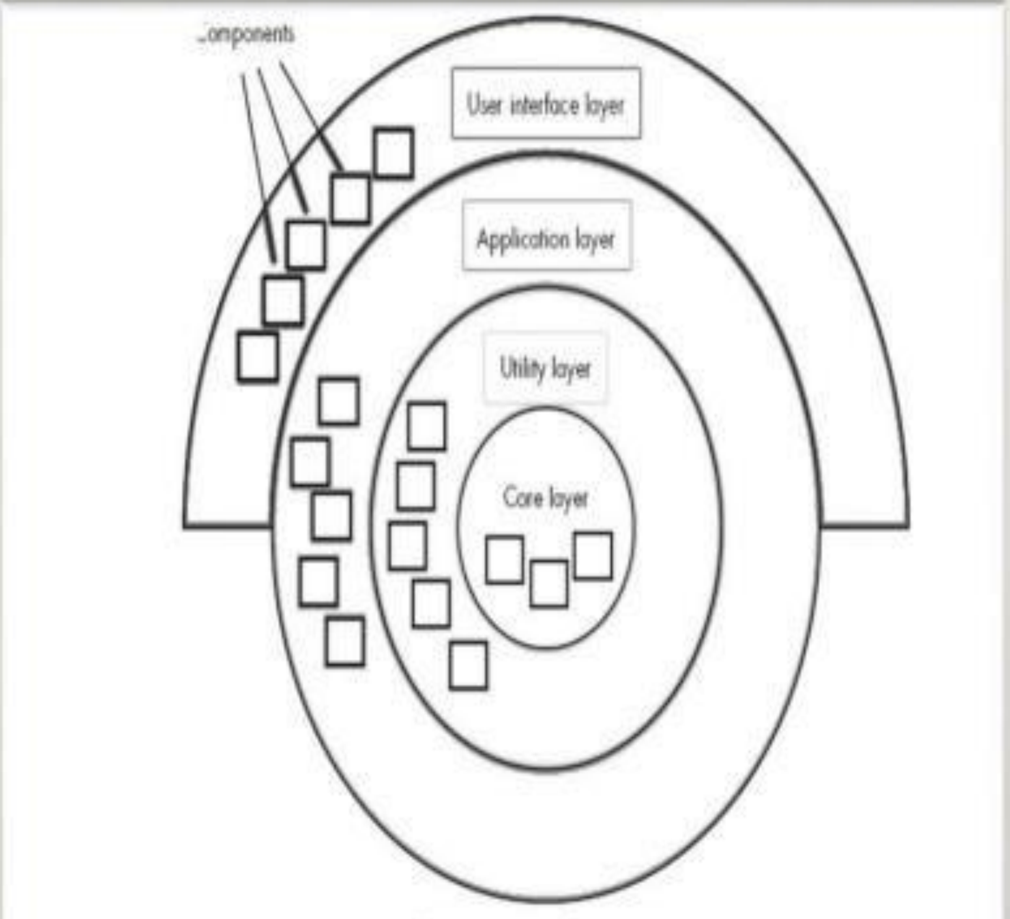
2. Identification of attribute and functions and their encapsulation of these within a class 3.Mechanism for representation of the content of each data object. Class diagrams may be used 4. Refinement of data design elements from requirement analysis to component level design. 5.Information hiding
6. A library of useful data structures and operations be developed.
7. 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) Constraints that define how components can be integrated to form the system
(4) Semantic 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/subprogram 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)



Components
User interface layer
Application layer
Utility layer
Core layer

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
2.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)
3. Distribution
-- Addresses the communication of system in a distributed environment
--Approaches (Patterns)
(a) Broker Pattern
-- Acts as middleman between client and server.

<u>**Performing User interface design:**</u> Golden rules, User interface analysis and design, interface analysis,interface design steps, Design evaluation.

<u>**Golden Rules**</u>
1. Place the user in control
2. Reduce the user's memory load
3. Make the interface consistent

Place the User in Control
• Define interaction modes in a way that does not force a user into unnecessary or undesired actions.

• Provide for flexible interaction.

• Allow user interaction to be interruptible and undoable.

• Streamline interaction as skill levels advance and allow the interaction to be customized.

• Hide technical internals from the casual user.

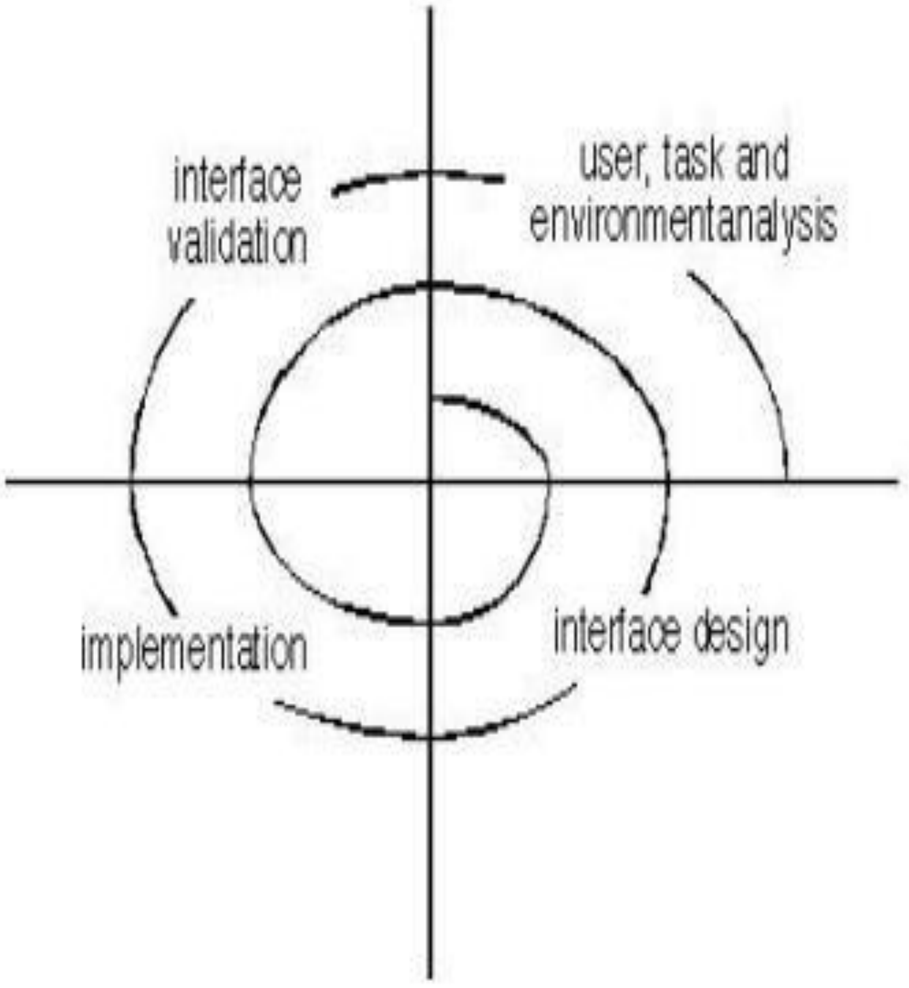
• Design for direct interaction with objects that appear on the screen.

Make the Interface Consistent. Allow the user to put the current task into a meaningful context. Maintain consistency across a family of applications. If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

<u>**USER INTERFACE ANALYSIS AND DESIGN**</u>

The overall process for analyzing and designing a user interface begins with the creation of different models of system function. There are 4 different models that is to be considered when a user interface is to be analyzed and designed.
User Interface Design Models
User model —Establishes a profile of all end users of the system

Design model — A design model of the entire system incorporates data, architectural, interface and procedural representation of the software.
A design realization of the user model User's Mental model (system perception). the user's mental image of what the interface is Implementation model — the interface "look and feel" coupled with supporting information that describe interface syntax and semantics
Users can be categorized as
1. Novice – No syntactic knowledge of the system and little semantic knowledge of the application or computer usage of the system
2. Knowledgeable, intermittent users- Reasonable semantic knowledge of the application but low recall of syntactic information to use the system
3. Knowledgeable, frequent users- Good semantic and syntactic knowledge User interface analysis and design process
• The user interface analysis and design process is an iterative process and it can be represented as a spiral model
It consists of framework activities 1.User, task and environment analysis 2.Interface design 3.Interface construction 4.Interface validation



User Interface Design Process

Interface analysis
-Understanding the user who interacts with the system based on their skill levels.i.e, requirement gathering
-The task the user performs to accomplish the goals of the system are identified, described and elaborated. Analysis of work environment.
Interface design
In interface design, all interface objects and actions that enable a user to perform all desired task are defined
Implementation
A prototype is initially constructed and then later user interface development tools may be used to complete the construction of the interface.
Validation
The correctness of the system is validated against the user requirement
Interface Analysis
Interface analysis means understanding
– (1) the people (end-users) who will interact with the system through the interface;
– (2) the tasks that end-users must perform to do their work,
– (3) the content that is presented as part of the interface
– (4) the environment in which these tasks will be conducted.

Design Evaluation Cycle: Steps:
Preliminary design Build prototype #1
Interface evaluation is studied by designer Design modifications are made
Build prototype # n
Interface
User evaluate's interface Interface design is complete

SYSTEM MODELS

Used in analysis process to develop understanding of the existing system or new system. Excludes details. An abstraction of the system
Types of system models
1.Context models
2. Behavioural models
3.Data models
4.Object models
5.Structured models

CONTEXT MODELS
A type of architectural model. Consists of sub-systems that make up an entire system
First step: To identify the subsystem.
Represent the high level architectural model as simple block diagram
- Depict each sub system a named rectangle
- Lines between rectangles indicate associations between subsystems Disadvantages

--Concerned with system environment only, doesn't take into account other systems, which may take data or give data to the model

The context of an ATM system consists of the following Auto-teller system Security system Maintenance system Account data base Usage database Branch accounting system Branch counter system

BEHAVIOUR MODELS

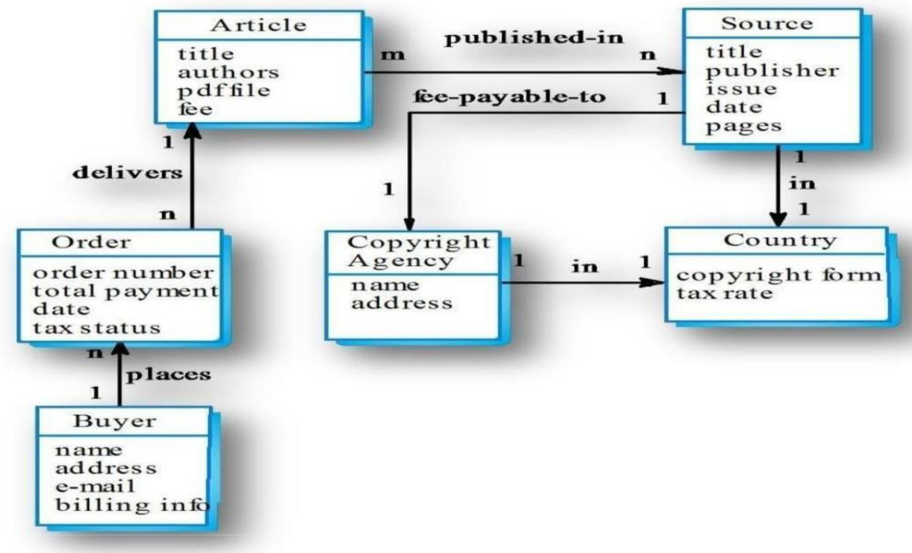
Describes the overall behaviour of a system. Two types of behavioural model
1.Data Flow models
2.State machine models

<u>Data flow models</u> --Concentrate on the flow of data and functional transformation on that data. Show the processing of data and its flow through a sequence of processing steps. Help analyst understand what is going on

Advantages
-- Simple and easily understandable
-- Useful during analysis of requirements
STATE MACHINE MODELS
Describe how a system responds to internal or external events. Shows system states and events that cause transition from one state to another. Does not show the flow of data within the system. Used for modeling of real time systems
Exp: Microwave oven
Assumes that at any time, the system is in one of a number of possible states. Stimulus triggers a transitionfrom on state to another state
Disadvantage
-- Number of possible states increases rapidly for large system models

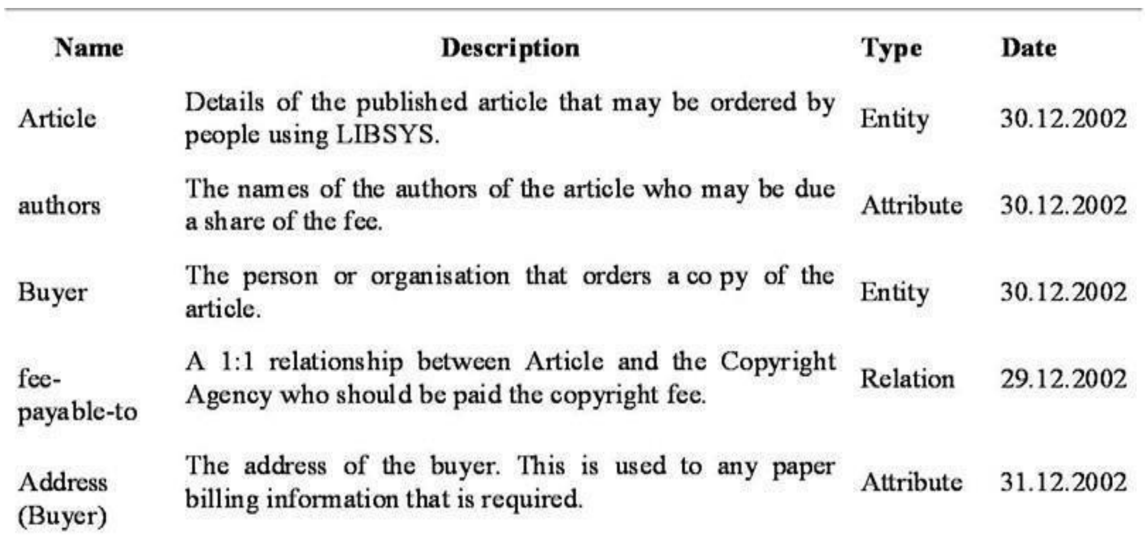
DATA MODELS

Used to describe the logical structure of data processed by the system. An entity-relation- attribute model sets out the entities in the system, the relationships between these entities and the entity attributes. Widely used in database design. Can readily be implemented using relational databases. No specific notation provided in the UML but objects and associations can be used.



OBJECT MODELS
An object oriented approach is commonly used for interactive systems development. Expresses the systems requirements using objects and developing the system in an object oriented PL such as c++ A object class: An abstraction over a set of objects that identifies common attributes. Objects are instances of object class. Many objects may be created from a single class.
Analysis process
-- Identifies objects and object classes Object class in UML
--Represented as a vertically oriented rectangle with three sections
(a) The name of the object class in the top section
(b) The class attributes in the middle section
(c) The operations associated with the object class are in lower section.

<u>OBJECT MODELS INHERITANCE MODELS</u>
A type of object oriented model which involves in object classes attributes. Arranges classes into an inheritance hierarchy with the most general object class at the top of hierarchy Specialized objects inherit their attributes and services
UML notation
-- Inheritance is shown upward rather than downward
--Single Inheritance: Every object class inherits its attributes and operations from a single parent class
--Multiple Inheritance: A class of several of several parents.



Name	Description	Type	Date
Article	Details of the published article that may be ordered by people using LIBSYS.	Entity	30.12.2002
authors	The names of the authors of the article who may be due a share of the fee.	Attribute	30.12.2002
Buyer	The person or organisation that orders a co py of the article.	Entity	30.12.2002
fee-payable-to	A 1:1 relationship between Article and the Copyright Agency who should be paid the copyright fee.	Relation	29.12.2002
Address (Buyer)	The address of the buyer. This is used to any paper billing information that is required.	Attribute	31.12.2002

UML Diagrams

The UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML gives you a standard way to write a system's blueprints, covering conceptual things, such as business processes and system functions, as well as concrete things, such as classes written in a specific programming language, database schemas, and reusable software components.

Model

A model is a simplification of reality. A model provides the blueprints of a system. A model may be structural, emphasizing the organization of the system, or it may be behavioral, emphasizing the dynamics of the system.

Why do we model

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

Through modeling, we achieve four aims.

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

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

Principles of Modeling

There are four basic principles of model

1. The choice of what models to create has a profound influence on how a problem is attacked and how a solution is shaped.
2. Every model may be expressed at different levels of precision.
3. The best models are connected to reality.
4. No single model is sufficient. Every nontrivial system is best approached through a small set of nearly independent models.

An Overview of UML

- The Unified Modeling Language is a standard language for writing software blueprints. The UML may be used to visualize, specify, construct, and document the artifacts of a software-intensive system.
- The UML is appropriate for modeling systems ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems. It is a very expressive language, addressing all the views needed to develop and then deploy such systems.

  The UML is a language for
  - Visualizing
  - Specifying
  - Constructing
  - Documenting

- **Visualizing** The UML is more than just a bunch of graphical symbols. Rather, behind each symbol in the UML notation is a well-defined semantics. In this manner, one developer can write a model in the UML, and another developer, or even another tool, can interpret that model unambiguously
- **Specifying** means building models that are precise, unambiguous, and complete.

- **Constructing** the UML is not a visual programming language, but its models can be directly connected to a variety of programming languages
- **Documenting** a healthy software organization produces all sorts of artifacts in addition to raw executable code. These artifacts include
  - Requirements
  - Architecture
  - Design
  - Source code
  - Project plans
  - Tests
  - Prototypes
  - Releases

To understand the UML, you need to form a **conceptual model of the language**, and this requires learning three major elements:

1. Things
2. Relationships
3. Diagrams

Things in the UML

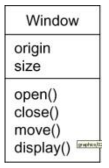
There are four kinds of things in the UML:

Structural things
Behavioral things
Grouping things
Annotational 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.

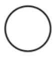
1. Classes
2. Interfaces
3. Collaborations
4. Use cases
5. Active classes
6. Components
7. Nodes

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.



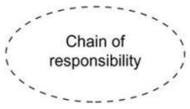
Interface
Interface is a collection of operations that specify a service of a class or component.
An interface therefore describes the externally visible behavior of that element.
An interface might represent the complete behavior of a class or component or only a part of that behavior.

An interface 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



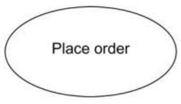
Collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behavior that's bigger than the sum of all the elements. Therefore, collaborations have structural, as well as behavioral, dimensions. A given class might participate in several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

Graphically, a collaboration is rendered as an ellipse with dashed lines, usually including only its name

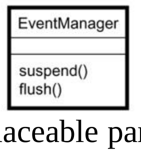


Use case

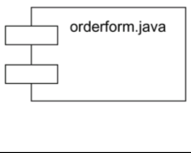
- Use case is a description of set of sequence of actions that a system performs that yields an observable result of value to a particular actor
- Use case is used to structure the behavioral things in a model.
- A use case is realized by a collaboration. Graphically, a use case is rendered as an ellipse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



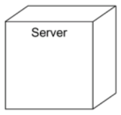
Active class is just like a class except that its objects represent elements whose behavior is concurrent with other elements. Graphically, an active class is rendered just like a class, but with heavy lines, usually including its name, attributes, and operations



Component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Graphically, a component is rendered as a rectangle with tabs



Node is a physical element that exists at run time and represents a computational resource, generally having at least some memory and, often, processing capability. Graphically, a node is rendered as a cube, usually including only its na



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

Interaction

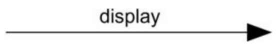
state machine

Interaction

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

An interaction involves a number of other elements, including messages, action sequences and links

Graphically a message is rendered as a directed line, almost always including the name of its operation

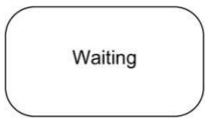


<u>State Machine</u>

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

State machine involves a number of other elements, including states, transitions, events and activities

Graphically, a state is rendered as a rounded rectangle, usually including its name and its substates

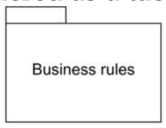


<u>Grouping Things:-</u>

1. are the organizational parts of UML models. These are the boxes into which a model can be decomposed
2. There is one primary kind of grouping thing, namely, packages.

<u>Package:-</u>

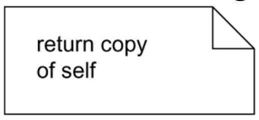
- A package is a general-purpose mechanism for organizing elements into groups. Structural things, behavioral things, and even other grouping things may be placed in a package
- Graphically, a package is rendered as a tabbed folder, usually including only its name and, sometimes, its contents



<u>Annotational things</u> are the explanatory parts of UML models. These are the comments you may apply to describe about any element in a model.

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



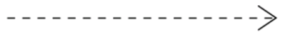
Relationships in the UML: There are four kinds of relationships in the UML:

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

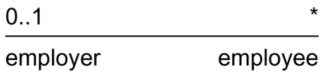
Dependency:-

Dependency is a semantic relationship between two things in which a change to one thing may affect the semantics of the other thing

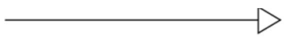
Graphically a dependency is rendered as a dashed line, possibly directed, and occasionally including a label



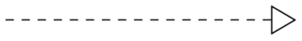
Association is a structural relationship that describes a set of links, a link being a connection among objects.
Graphically an association is rendered as a solid line, possibly directed, occasionally including a label, and often containing other adornments, such as multiplicity and role names



Aggregation is a special kind of association, representing a structural relationship between a whole and its parts. Graphically, a generalization relationship is rendered as a solid line with a hollow arrowhead pointing to the parent



Realization is a semantic relationship between classifiers, wherein one classifier specifies a contract that another classifier guarantees to carry out. Graphically a realization relationship is rendered as a cross between a generalization and a dependency relationship

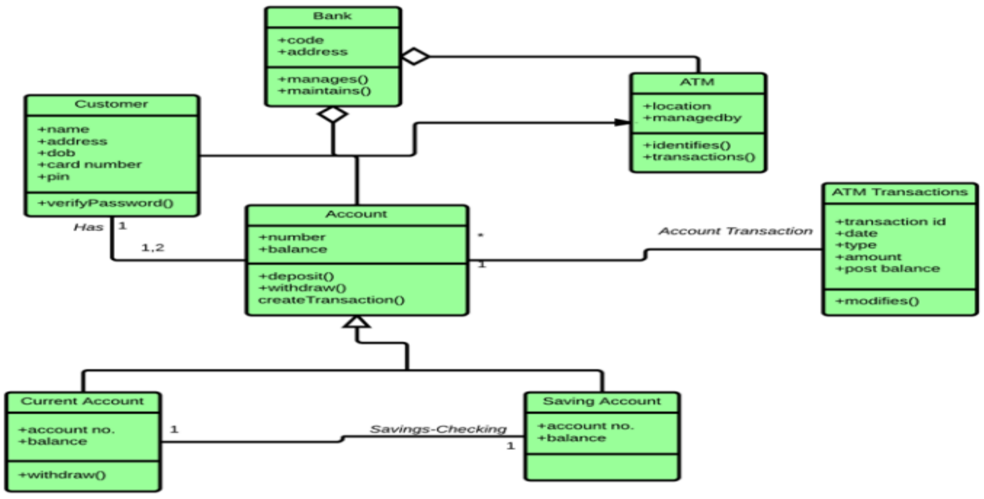


Diagrams in the UML

- **Diagram** is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).
- In theory, a diagram may contain any combination of things and relationships.
- For this reason, the UML includes nine such diagrams:
  - Class diagram
  - Object diagram
  - Use case diagram
  - Sequence diagram
  - Collaboration diagram
  - State chart diagram
  - Activity diagram
  - Component diagram
  - Deployment diagram

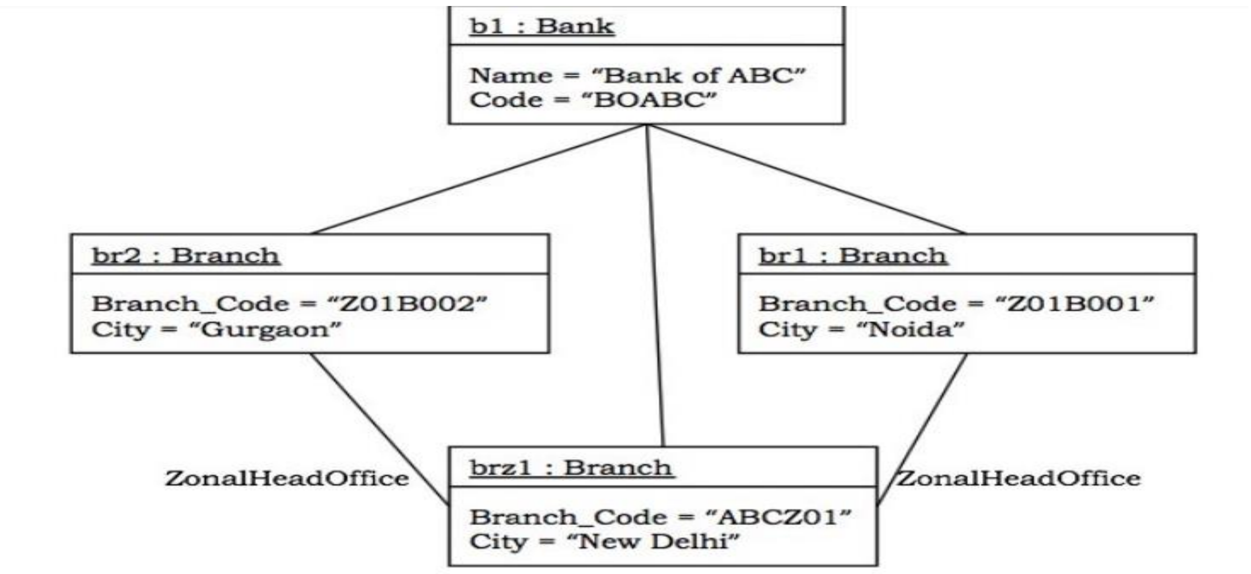
Class diagram

A class diagram shows a set of classes, interfaces, and collaborations and their relationships. Class diagrams that include active classes address the static process view of a system.



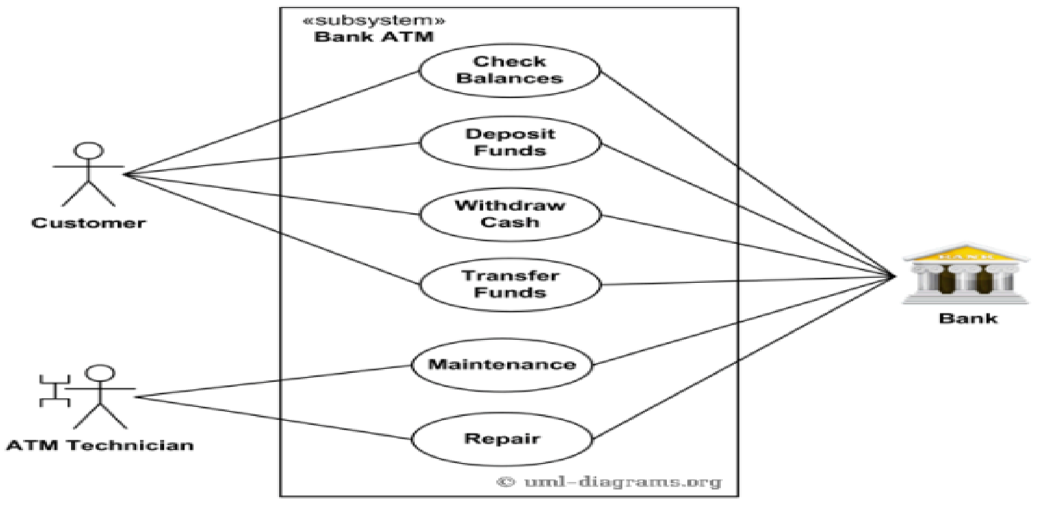
Object diagram

- 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
- An object diagram shows a set of objects and their relationships



Use case diagram

- A use case diagram shows a set of use cases and actors 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.

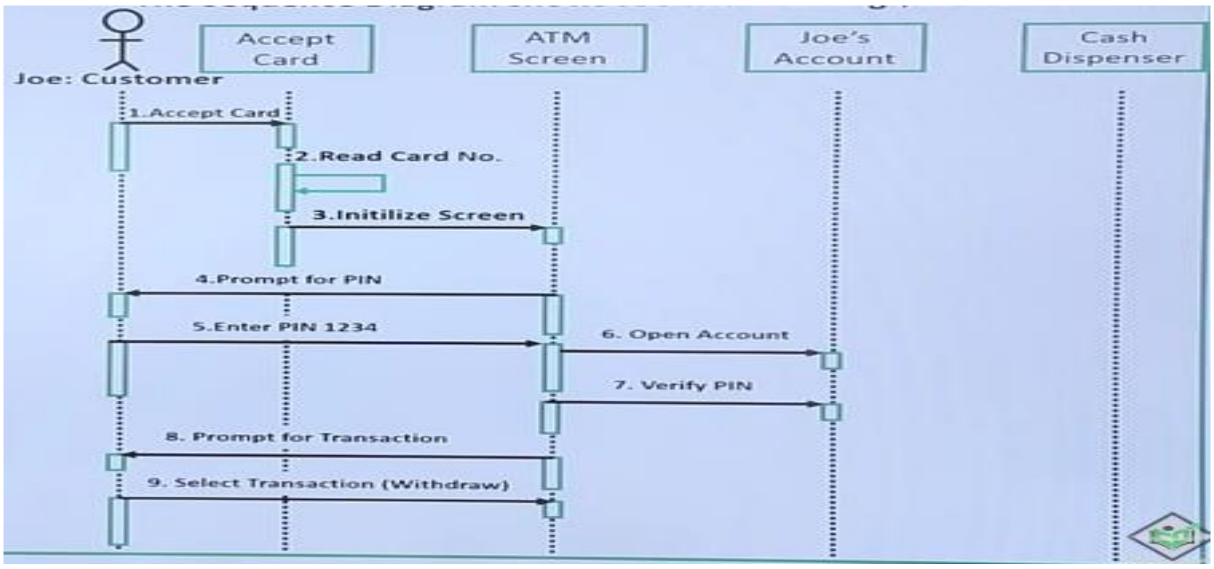


Interaction Diagrams

Both sequence diagrams and collaboration diagrams are kinds of interaction diagrams

Interaction diagrams address the dynamic view of a system

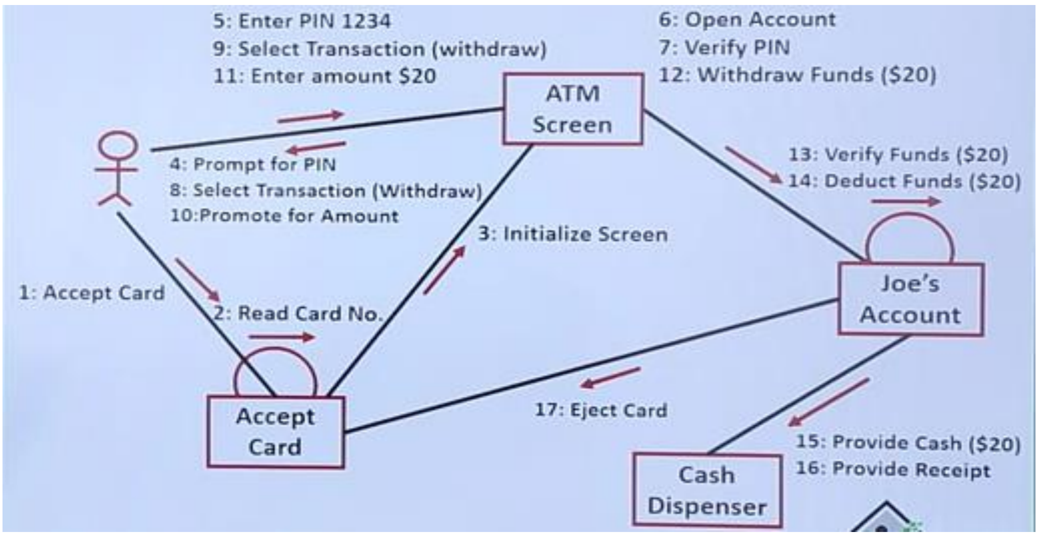
A sequence diagram is an interaction diagram that emphasizes the time-ordering of messages



A collaboration diagram is an interaction diagram that emphasizes the structural organization of the

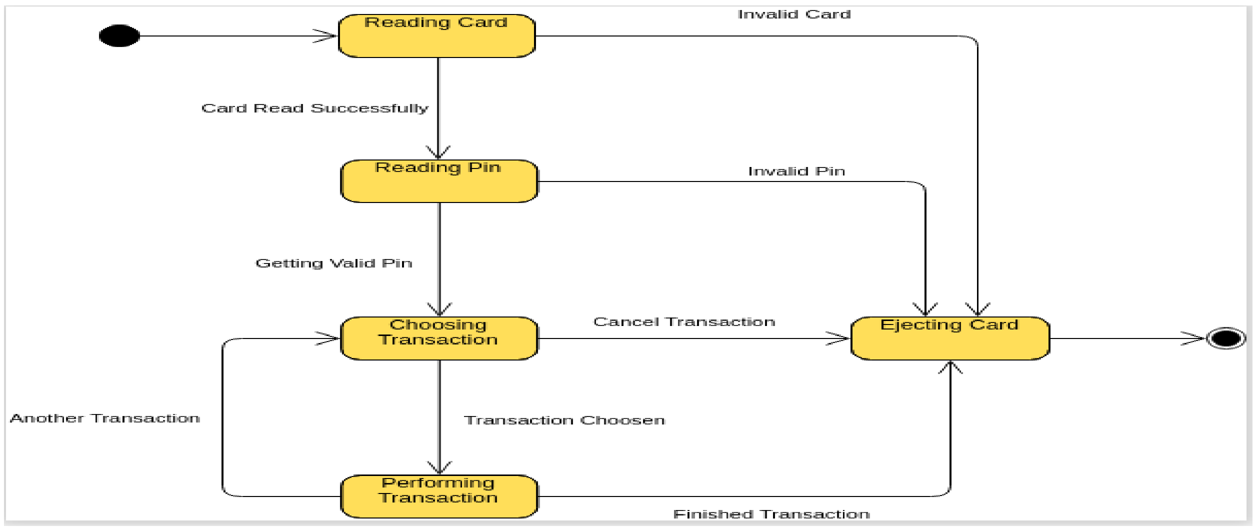
objects that send and receive messages

Sequence diagrams and collaboration diagrams are isomorphic, meaning that you can take one and transform it into the other



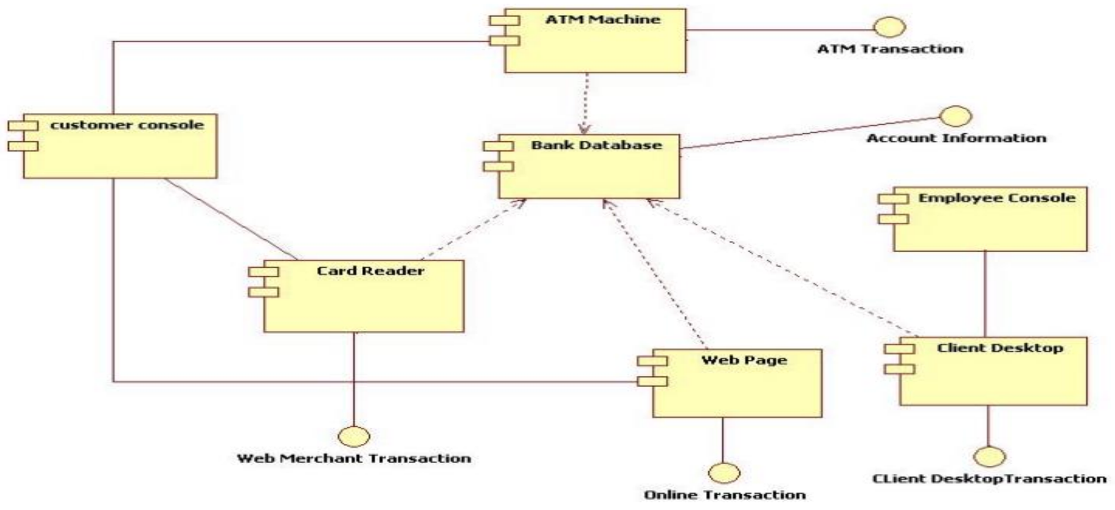
Statechart diagram

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



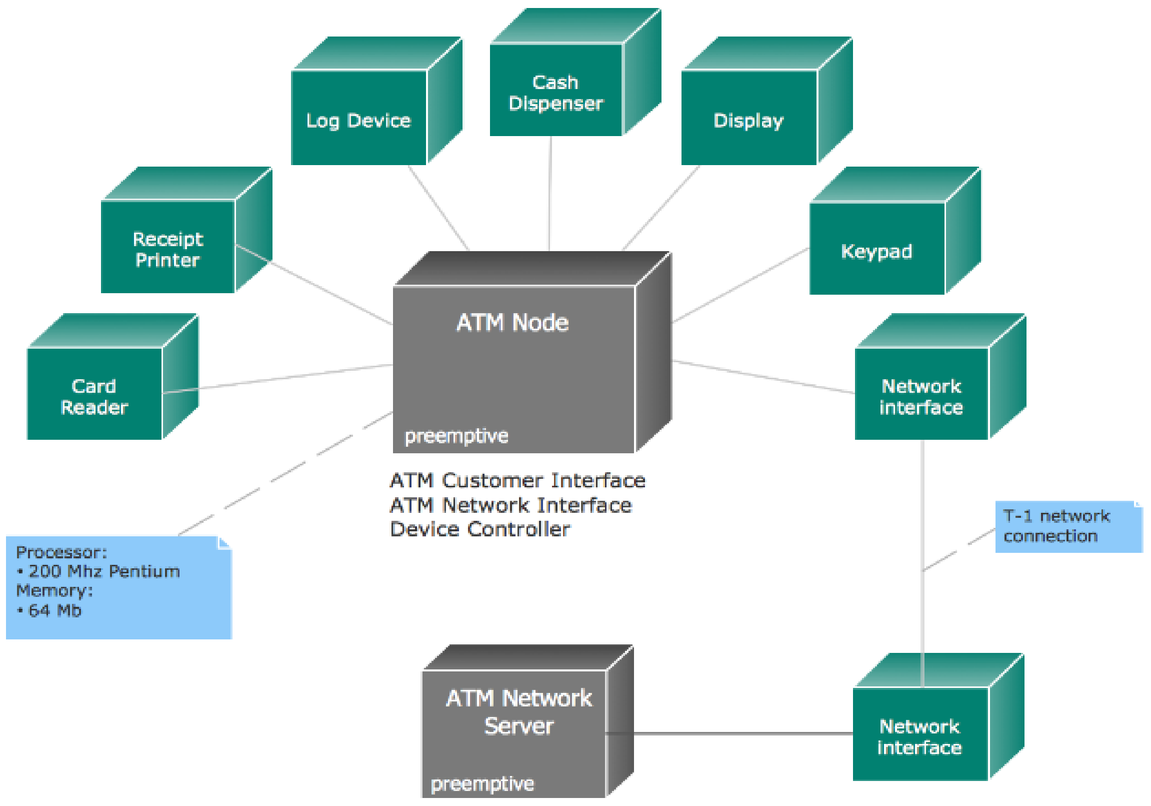
Component diagram

- 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



Deployment diagram

- A deployment diagram shows the configuration of run-time processing nodes and the components that live on them
- Deployment diagrams address the static deployment view of an architecture



UNIT– IV

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

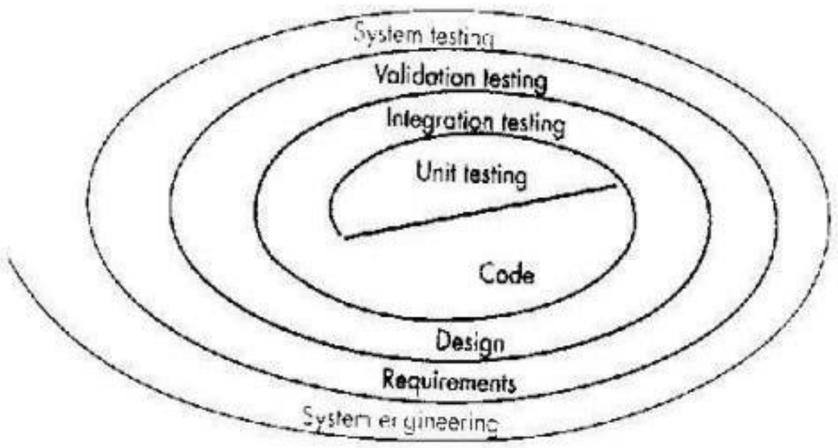
A strategic Approach for Software testing

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 the set of activities that ensure that software correctly implements a specific function. (Ex: Are we building 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

Testing can be done by software developer and independent testing group. Testing and 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 of four levels of testing:
1) Unit Testing
2)Integration Testing
3)Validation Testing and
4) System Testing

Spiral Representation of Testing for Conventional Software



Unit Testing begins at the vortex of the spiral and concentrates on each unit 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.

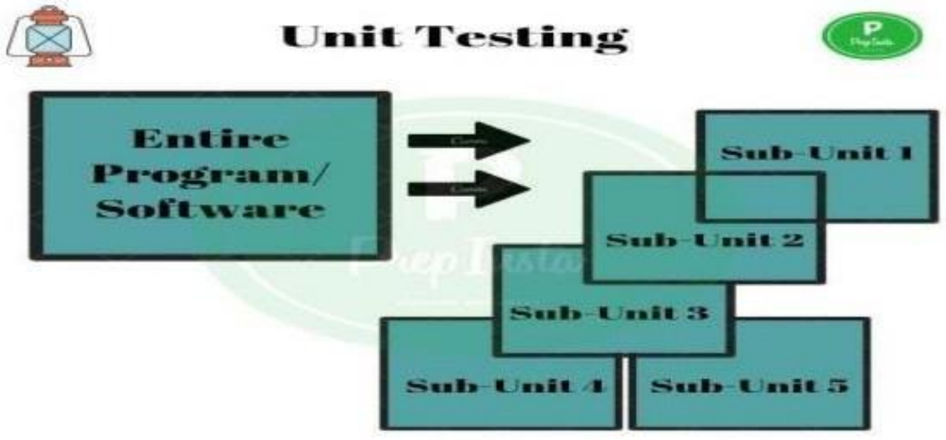


Fig: Unit Testing

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.

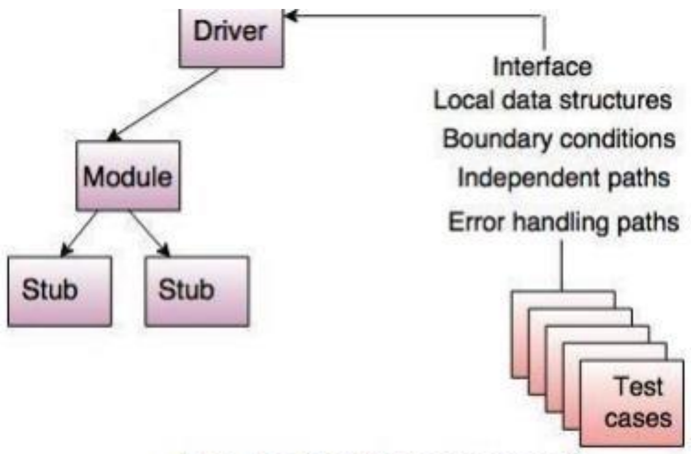


Fig. - Unit test environment

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.

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, behavioural and performance requirements. It succeeds when the software functions 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 by end 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 prespecified 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.

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 operations of a system and examines the procedural detail.

Black box testing
This is also called behavioural 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 behaviour can be determined by studying its input and related output Not concerned with the internal. The various testing methods employed here are:
1) 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.

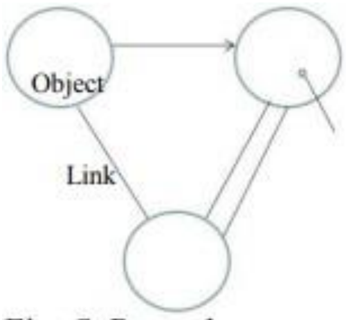


Fig: O-R graph.

2) 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 represents a set of valid or invalid states for input conditions. Reduces the cost of testing
Example

Input consists of 1 to 10 Then classes are n<1,1<=n<=10,n>10 Choose one valid class with value within the allowed range and two invalid classes where values are greater than maximum value and smaller than minimum value.

3) 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 If 0.0<=x<=1.0

Then test cases are (0.0,1.0) for valid input and (-0.1 and 1.1) for invalid input

4)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. Orthogonal testing will reduce the number of test case to 9

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

1. Basis path testing
2.Controlstructure testing

1.Basis path testing

Proposed by Tom McCabe. 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 flow chart 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+2 where E is number of edges, N is number of nodes
2.V(G)=Number of regions

The structured constructs used in the flow graph are:

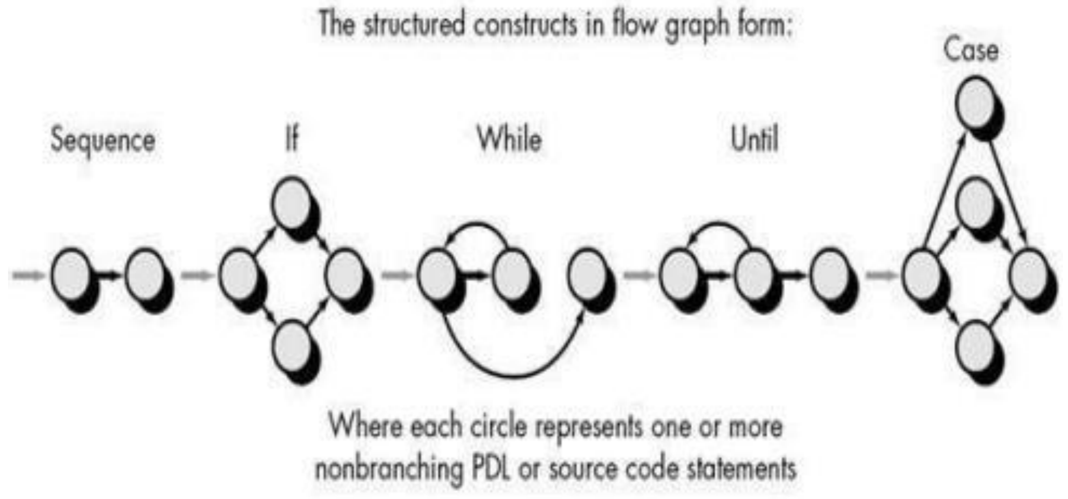

The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

Fig: Basis path testing

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

2.<u>Control Structure testing</u>
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<booleon operator>simple condition
Types of errors include operator errors, variable errors, arithmetic expression errors etc.

b) Data flow 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 subsequent 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: Computational use of a variable like read, write or variable on the right hand of 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+1 passes the loop

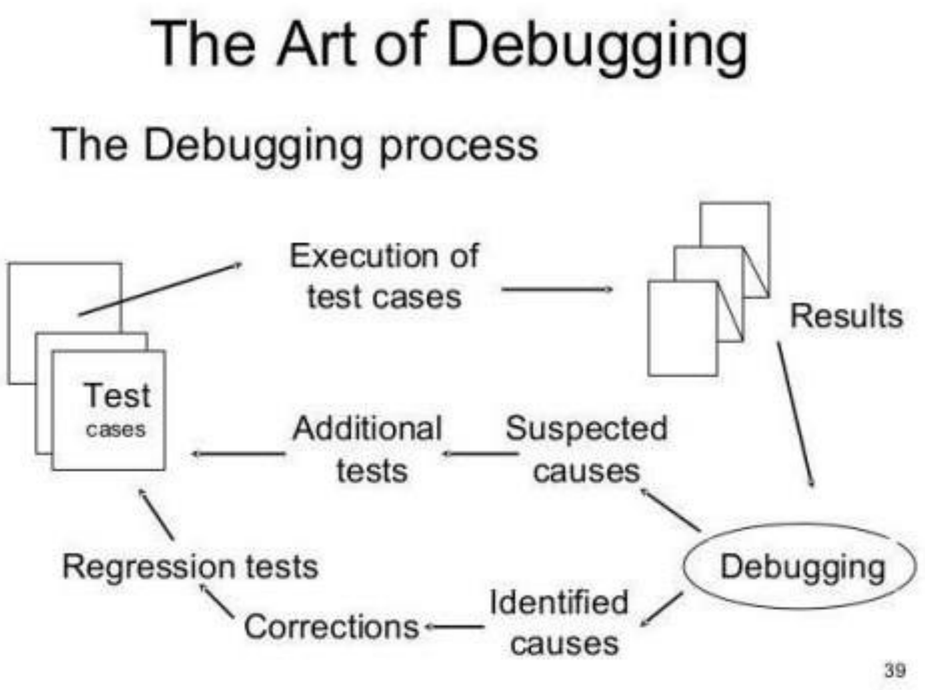


Fig: Debugging process

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.

Back tracking: 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 occurenece 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 behaviour 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

Software process and product metrics

These are quantitative measures that enable software people to gain insight into the efficacy of the software process and the projects that are conducted using the process as a framework. Basic quality and productivity data are collected. These data are then analyzed, compared against past averages, and assessed to determine whether quality and productivity improvements have occurred.Metrics are also used to pinpoint problem areas so that remedies can be developed and the software process can be improved.

Software measurement

Software measurement is a quantified attribute of a characteristic of a software product or the software process.
Compare the similarity with
A software metric is a measure of some property of a piece of software or its specifications.

The purpose of software measurement
Prediction –
To predict complexity
To predict usage
Predictive analytics
Control –
Production hours, cost, security, quality
Assessment –
Usability
Web analytics
Return on Investment

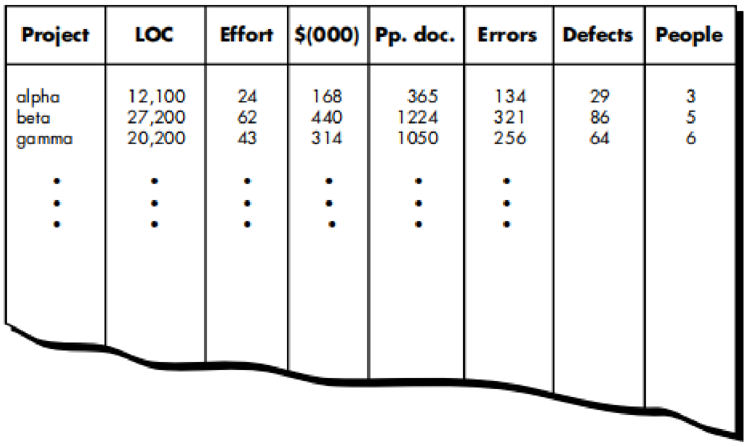
Some generic examples

Halstead
McCall et al.
Boehm – CoCoMo
Albrecht – Function Point Analysis
Bevan – Usability metrics.
Nielsen - Heuristics

Measurements in the physical world can be categorized in two ways: direct measures (e.g., the length of a bolt) and indirect measures (e.g., the "quality" of bolts produced, measured by counting rejects).
Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time. Indirect measures of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many other "–abilities" .
The cost and effort required to build software, the number of lines of code produced, and other direct measures are relatively easy to collect, as long as specific conventions for measurement are established in advance. However, the quality and functionality of software or its efficiency or maintainability are more difficult to assess and can be measured only indirectly.
We have already partitioned the software metrics domain into process, project, and product metrics.

Size-Oriented Metrics

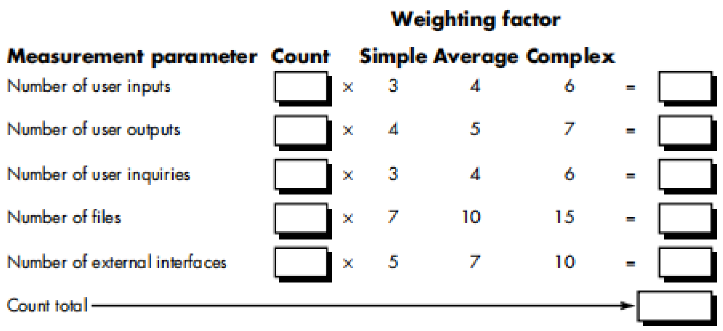
Size-oriented software metrics are derived by normalizing quality and/or productivity measures by considering the size of the software that has been produced. If a software organization maintains simple records, a table of size-oriented measures, such
as the one shown in Figure 4.4, can be created. The table lists each software development project that has been completed over the past few years and corresponding
measures for that project. Referring to the table entry (Figure 4.4) for project alpha: 12,100 lines of code were developed with 24 person-months of effort at a cost of $168,000. It should be noted that the effort and cost recorded in the table represent all software engineering activities (analysis, design, code, and test), not just coding. Further information for project alpha indicates that 365 pages of documentation were developed, 134 errors were recorded before the software was released, and 29 defects



Project	LOC	Effort	$(000)	Pp. doc.	Errors	Defects	People
alpha	12,100	24	168	365	134	29	3
beta	27,200	62	440	1224	321	86	5
gamma	20,200	43	314	1050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

were encountered after release to the customer within the first year of operation. Three people worked on the development of software for project alpha.
In order to develop metrics that can be assimilated with similar metrics from other projects, we choose lines of code as our normalization value. From the rudimentary data contained in the table, a set of simple size-oriented metrics can be developed for each project:
• Errors per KLOC (thousand lines of code).

• Defects4 per KLOC.
• $ per LOC.
• Page of documentation per KLOC.
In addition, other interesting metrics can be computed:
• Errors per person-month.
• LOC per person-month.
• $ per page of documentation.
Size-oriented metrics are not universally accepted as the best way to measure the process of software development [JON86]. Most of the controversy swirls around the use of lines of code as a key measure. Proponents of the LOC measure claim that LOC is an "artifact" of all software development projects that can be easily counted, that many existing software estimation models use LOC or KLOC as a key input, and that a large body of literature and data predicated on LOC already exists. On the other hand, opponents argue that LOC measures are programming language dependent, that they penalize well-designed but shorter programs, that they cannot easily accommodate non procedural languages, and that their use in estimation requires a level of detail that may be difficult to achieve (i.e., the planner must estimate the LOC to be produced long before analysis and design have been completed).
4.3.2 Function-Oriented Metrics
Function-oriented software metrics use a measure of the functionality delivered by the application as a normalization value. Since ‘functionality’ cannot be measured directly, it must be derived indirectly using other direct measures. Function-oriented metrics were first proposed by Albrecht [ALB79], who suggested a measure called the function point. Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and assessments of software complexity.
Function points are computed [IFP94] by completing the table shown in Figure 4.5. Five information domain characteristics are determined and counts are provided in



			Weighting factor				
Measurement parameter	Count		Simple	Average	Complex		
Number of user inputs	☐	×	3	4	6	=	☐
Number of user outputs	☐	×	4	5	7	=	☐
Number of user inquiries	☐	×	3	4	6	=	☐
Number of files	☐	×	7	10	15	=	☐
Number of external interfaces	☐	×	5	7	10	=	☐
Count total							☐

ing manner:5
Number of user inputs. Each user input that provides distinct applicationoriented data to the software is counted. Inputs should be distinguished from inquiries, which are counted separately.
Number of user outputs. Each user output that provides applicationoriented information to the user is counted. In this context output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.
Number of user inquiries. An inquiry is defined as an on-line input that results in the generation of some immediate software response in the form of an on-line output. Each distinct inquiry is counted.
Number of files. Each logical master file (i.e., a logical grouping of data that may be one part of a large database or a separate file) is counted.
Number of external interfaces. All machine readable interfaces (e.g., data files on storage media) that are used to transmit information to another system are counted.
Once these data have been collected, a complexity value is associated with each count. Organizations that use function point methods develop criteria for determining whether a particular entry is simple, average, or complex. Nonetheless, the determination of complexity is somewhat subjective.

To compute function points (FP), the following relationship is used:
FP = count total [0.65 + 0.01 ∑(Fi
)] (4-1)
where count total is the sum of all FP entries obtained from Figure 4.5.
The Fi (i = 1 to 14) are "complexity adjustment values" based on responses to the following questions [ART85]:
1. Does the system require reliable backup and recovery?
2. Are data communications required?
3. Are there distributed processing functions?
4. Is performance critical?
5. Will the system run in an existing, heavily utilized operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the master files updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organizations?
14. Is the application designed to facilitate change and ease of use by the user?
Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential). The constant values in Equation (4-1) and the weighting factors that are applied to information domain counts are determined empirically.
Once function points have been calculated, they are used in a manner analogous to LOC as a way to normalize measures for software productivity, quality, and other attributes:
• Errors per FP.
• Defects per FP.
• $ per FP.
• Pages of documentation per FP.
• FP per person-month

Extended Function Point Metrics

The function point measure was originally designed to be applied to business information systems applications. To accommodate these applications, the data dimension (the information domain values discussed previously) was emphasized to the
exclusion of the functional and behavioral (control) dimensions. For this reason, the
function point measure was inadequate for many engineering and embedded systems (which emphasize function and control). A number of extensions to the basic
function point measure have been proposed to remedy this situation.
A function point extension called feature points [JON91], is a superset of the function
point measure that can be applied to systems and engineering software applications.

Metrics for software quality

A good software engineer uses measurement to assess the quality of the analysis and design models, the source code, and the test cases that have been created as the software is engineered. To accomplish this real-time quality assessment, the engineer must use technical measures to evaluate quality in objective, rather than subjective ways.A good software engineer uses measurement to assess the quality of the analysis and design models, the source code, and the test cases that have been created as the software is engineered. To accomplish this real-time quality assessment, the engineer
must use technical measures (Chapters 19 and 24) to evaluate quality in objective, rather than subjective ways.

An Overview of Factors That Affect Quality

Over 25 years ago, McCall and Cavano [MCC78] defined a set of quality factors that
were a first step toward the development of metrics for software quality. These fac

tors assess software from three distinct points of view: (1) product operation (using
it), (2) product revision (changing it), and (3) product transition (modifying it to work
in a different environment; i.e., "porting" it). In their work, the authors describe the
relationship between these quality factors (what they call a framework) and other
aspects of the software engineering process:
First, the framework provides a mechanism for the project manager to identify what
qualities are important. These qualities are attributes of the software in addition to its func
tional correctness and performance which have life cycle implications. Such factors as main
tainability and portability have been shown in recent years to have significant life cycle cost
impact . . .
Secondly, the framework provides a means for quantitatively assessing how well the
development is progressing relative to the quality goals established . . .
Thirdly, the framework provides for more interaction of QA personnel throughout the
development effort . . .
Lastly, . . . quality assurance personal can use indications of poor quality to help iden
tify [better] standards to be enforced in the future

Measuring Quality

Although there are many measures of software quality, correctness, maintainability,
integrity, and usability provide useful indicators for the project team. Gilb [GIL88] suggests definitions and
measures for each.
Correctness. A program must operate correctly or it provides little value to its users. Correctness is the degree
to which the software performs its required function. The most common measure for correctness is defects
per
KLOC, where a defect is defined as a verified lack of conformance to requirements. When considering the
overall quality of a software product, defects
are those problems reported by a user of the program after the program has
been released for general use. For quality assessment purposes, defects are
counted over a standard period of time, typically one year.
Maintainability. Software maintenance accounts for more effort than any
other software engineering activity. Maintainability is the ease with which a
program can be corrected if an error is encountered, adapted if its environment changes, or enhanced if the
customer desires a change in require
ments. There is no way to measure maintainability directly; therefore, we
must use indirect measures. A simple time-oriented metric is mean-time-tochange (MTTC), the time it takes to
analyze the change request, design an
appropriate modification, implement the change, test it, and distribute the
change to all users. On average, programs that are maintainable will have a
lower MTTC (for equivalent types of changes) than programs that are not
maintainable.

Hitachi [TAJ81] has used a cost-oriented metric for maintainability called
spoilage—the cost to correct defects encountered after the software has been
released to its end-users. When the ratio of spoilage to overall project cost
(for many projects) is plotted as a function of time, a manager can determine
whether the overall maintainability of software produced by a software
development organization is improving. Actions can then be taken in
response to the insight gained from this information.
Integrity. Software integrity has become increasingly important in the age
of hackers and firewalls. This attribute measures a system's ability to withstand attacks (both accidental and
intentional) to its security. Attacks can be

made on all three components of software: programs, data, and documents.
To measure integrity, two additional attributes must be defined: threat and security. Threat is the probability (which can be estimated or derived from empirical evidence) that an attack of a specific type will occur within a given time. Security is the probability (which can be estimated or derived from empirical evidence) that the attack of a specific type will be repelled. The integrity of a system can then be defined as

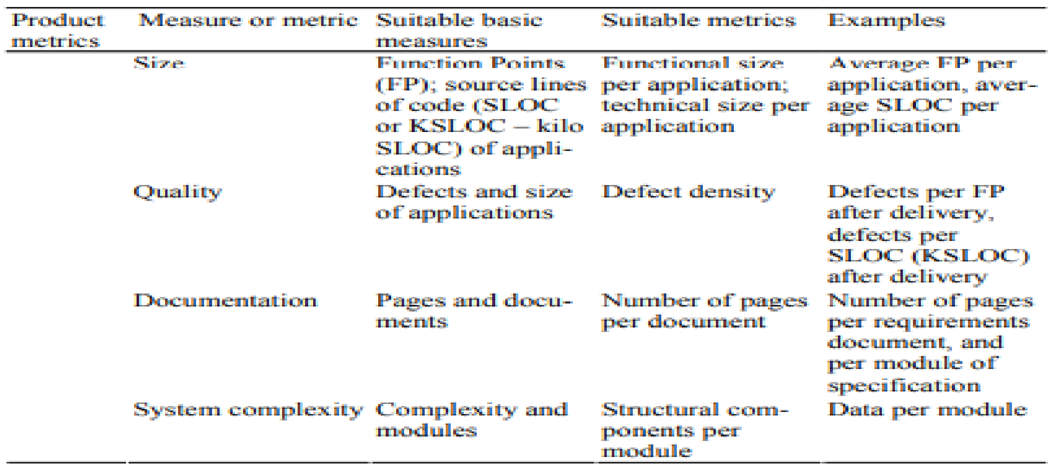
integrity = summation [(1 – threat) (1 – security)]
where threat and security are summed over each type of attack.

Usability. The catch phrase "user-friendliness" has become ubiquitous in discussions of software products. If a program is not user-friendly, it is often doomed to failure, even if the functions that it performs are valuable. Usability is an attempt to quantify user-friendliness and can be measured in terms
of four characteristics: (1) the physical and or intellectual skill required to learn the system, (2) the time required to become moderately efficient in the use of the system, (3) the net increase in productivity (over the approach that the system replaces) measured when the system is used by someone who is moderately efficient, and (4) a subjective assessment (sometimes obtained through a questionnaire) of users attitudes toward the system.
The four factors just described are only a sampling of those that have been proposed as measures for software quality.

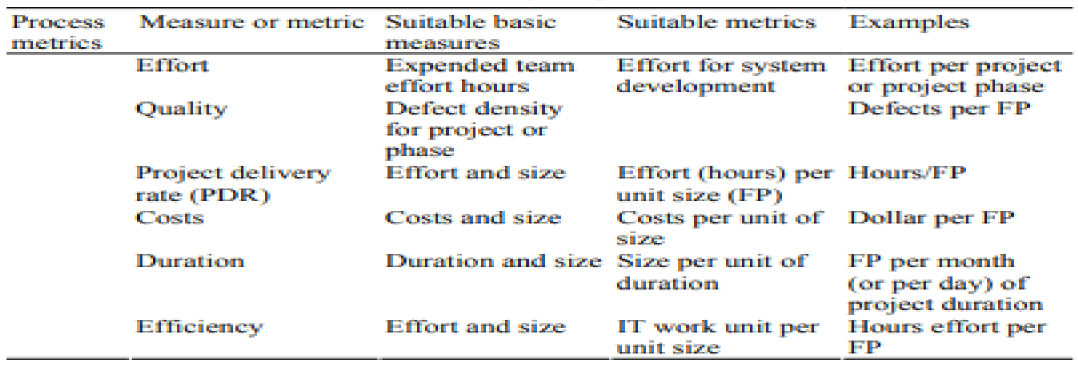
Defect Removal Efficiency

removal efficiency (DRE). In essence, DRE is a measure of the filtering ability of quality assurance and control activities as they are applied throughout all process framework activities.
When considered for a project as a whole, DRE is defined in the following manner:
DRE = E/(E + D)
where E is the number of errors found before delivery of the software to the end-user and D is the number of defects found after delivery.
The ideal value for DRE is 1. That is, no defects are found in the software. Realistically, D will be greater than 0, but the value of DRE can still approach 1. As E increases
(for a given value of D), the overall value of DRE begins to approach 1. In fact, as E increases, it is likely that the final value of D will decrease (errors are filtered out before they become defects). If used as a metric that provides an indicator of the filtering ability of quality control and assurance activities, DRE encourages a software project team
to institute techniques for finding as many errors as possible before delivery.
DRE can also be used within the project to assess a team’s ability to find errors before they are passed to the next framework activity or software engineering task.
For example, the requirements analysis task produces an analysis model that can be reviewed to find and correct errors. Those errors that are not found during the review of the analysis model are passed on to the design task (where they may or may not be found). When used in this context, we redefine DRE as
DREi = Ei
/(Ei + Ei+1) (4-5)
where Ei is the number of errors found during software engineering activity i and Ei+1 is the number of errors found during software engineering activity i+1 that are traceable to errors that were not discovered in software engineering activity i.

A quality objective for a software team (or an individual software engineer) is to achieve DREi that approaches 1. That is, errors should be filtered out before they are passed on to the next activity.



Product metrics	Measure or metric	Suitable basic measures	Suitable metrics	Examples
	Size	Function Points (FP); source lines of code (SLOC or KSLOC – kilo SLOC) of applications	Functional size per application; technical size per application	Average FP per application, average SLOC per application
	Quality	Defects and size of applications	Defect density	Defects per FP after delivery, defects per SLOC (KSLOC) after delivery
	Documentation	Pages and documents	Number of pages per document	Number of pages per requirements document, and per module of specification
	System complexity	Complexity and modules	Structural components per module	Data per module



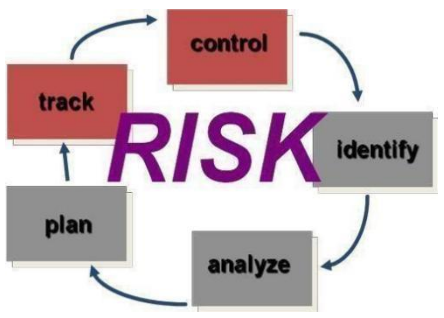
Process metrics	Measure or metric	Suitable basic measures	Suitable metrics	Examples
	Effort	Expended team effort hours	Effort for system development	Effort per project or project phase
	Quality	Defect density for project or phase		Defects per FP
	Project delivery rate (PDR)	Effort and size	Effort (hours) per unit size (FP)	Hours/FP
	Costs	Costs and size	Costs per unit of size	Dollar per FP
	Duration	Duration and size	Size per unit of duration	FP per month (or per day) of project duration
	Efficiency	Effort and size	IT work unit per unit size	Hours effort per FP

Software metrics refers to a broad range of measurements for computer software

UNIT – V

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
1)Project Risk 2)Technical Risk 3)Business Risk 4)Known Risk 5)Unpredictable Risk 6)Predictable risk
Project risk: Threaten the project plan and affect schedule and resultant cost
Technical risk:Threaten the quality and timeliness 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 checklist
Step3:Categorize into risk components-Performance risk, cost risk, support risk and schedule risk
Step4:Divide the risk into one of 4 categories

Risk Identification: The project organizer needs to anticipate the risk in the project as early as possible so that the impact of risk can be reduced by making effective risk management planning.

A project can be of use by a large variety of risk. To identify the significant risk, this might affect a project. It is necessary to categories into the different risk of classes.

There are different types of risks which can affect a software project:

1.Technology risks: Risks that assume from the software or hardware technologies that are usedto develop the system.

2.People risks: Risks that are connected with the person in the development team.

3.Organizational risks: Risks that assume from the organizational environment where thesoftware is being developed.

1.Tools risks: Risks that assume from the software tools and other support software used tocreate the system.

1. **Requirement risks:** Risks that assume from the changes to the customer requirement and the process of managing the requirements change.

2. **Estimation risks:** Risks that assume from the management estimates of the resources required to build the system

Risk Projection (aka Risk Estimation)

Attempts to rate each risk in two ways

· · The probability that the risk is real

· · The consequences of the problems associated with the risk, should it occur.

Project planner, along with other managers and technical staff, performs four risk projection activities:

(1) establish a measure that reflects the perceived likelihood of a risk

(2) delineate the consequences of the risk

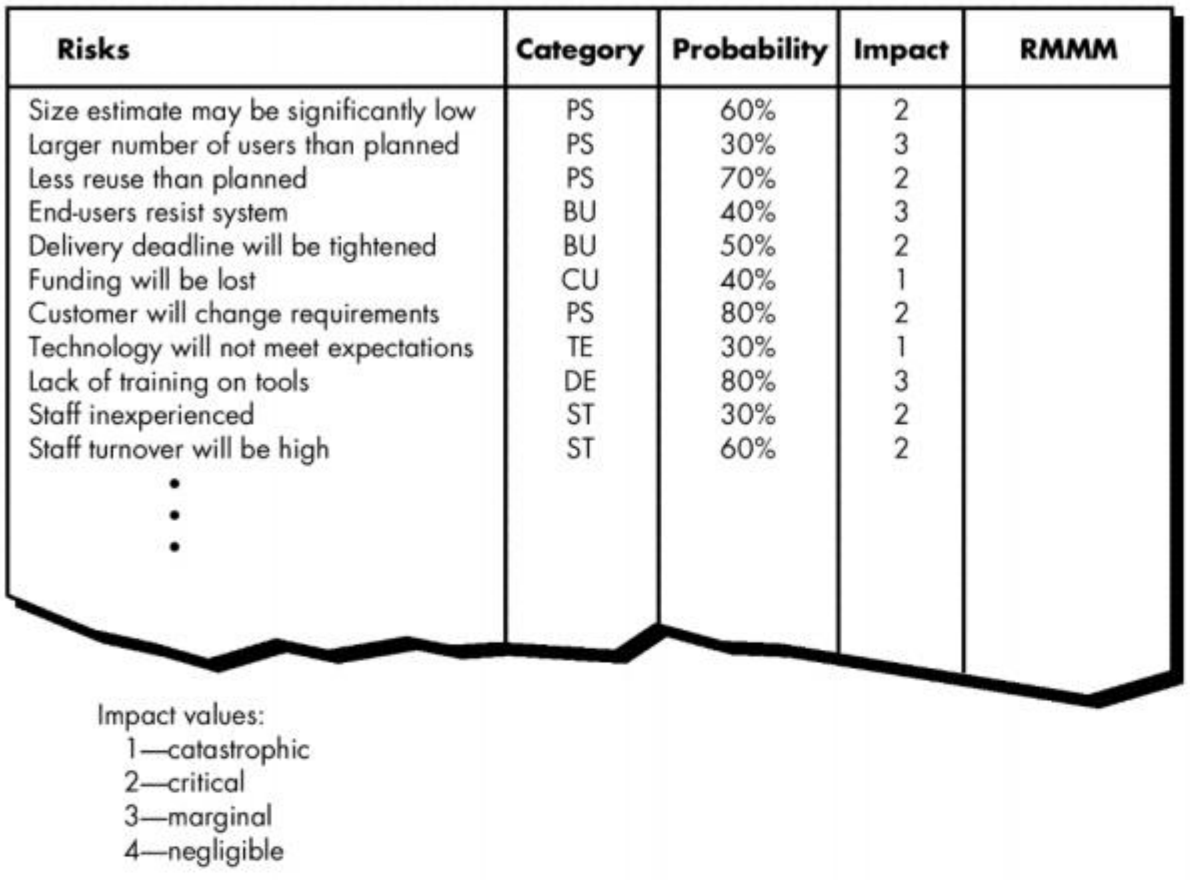
(3) estimate the impact of the risk on the project and the product

(4) note the overall accuracy of the risk projection so that there will be no misunderstandings.

Developing a Risk Table

Risk table provides a project manager with a simple technique for risk projection

igure 2 - Risk Table



Risks	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End-users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet expectations	TE	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	
•				
•				
•				

Impact values:
1—catastrophic
2—critical
3—marginal
4—negligible

Steps in Setting up Risk Table

(1) Project team begins by listing all risks in the first column of the table. Accomplished with the help

of the risk item checklists.

(2) Each risk is categorized in the second column
e.g., PS implies a project size risk, BU implies a business risk).

(3) The probability of occurrence of each risk is entered in the next column of the table. The probability value for each risk can be estimated by team members individually.

(4) Individual team members are polled in round-robin fashion until their assessment of risk probability begins to converge.

Risk Refinement

- · A risk may be stated generally during early stages of project planning.
- · With time, more is learned about the project and the risk
  - o may be possible to refine the risk into a set of more detailed risks
- · Represent risk in ***condition-transition-consequence*** (CTC) format.
  - o Stated in the following form:

Given that <condition> then there is concern that (possibly) <consequence>

- · Using CTC format for the reuse we can write:

Given that all reusable software components must conform to specific design standards and that some do not conform, then there is concern that (possibly) only 70 percent of the planned reusable modules may actually be integrated into the as-built system, resulting in the need to custom engineer the remaining 30 percent of components.

· This general condition can be refined in the following manner:

Subcondition 1. Certain reusable components were developed by a third party with no knowledge of internal design standards.

Subcondition 2. The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

Subcondition 3. Certain reusable components have been implemented in a language that is not supported on the target environment.

Risk Mitigation, Monitoring, and Management

A risk management technique is usually seen in the software Project plan. This can be divided into Risk Mitigation, Monitoring, and Management Plan (RMMM). In this plan, all works are done as part of risk analysis. As part of the overall project plan project manager generally uses this RMMM plan.

In some software teams, risk is documented with the help of a Risk Information Sheet (RIS). This RIS is controlled by using a database system for easier management of information i.e creation, priority ordering, searching, and other analysis. After documentation of RMMM and start of a project, risk mitigation and monitoring steps will start.

Risk Mitigation :

It is an activity used to avoid problems (Risk Avoidance).

Steps for mitigating the risks as follows.

1. Finding out the risk.
2. Removing causes that are the reason for risk creation.
3. Controlling the corresponding documents from time to time.
4. Conducting timely reviews to speed up the work.

Risk Monitoring :

It is an activity used for project tracking.

It has the following primary objectives as follows.

1. To check if predicted risks occur or not.
2. To ensure proper application of risk aversion steps defined for risk.
3. To collect data for future risk analysis.
4. To allocate what problems are caused by which risks throughout the project.

Risk Management and planning :

It assumes that the mitigation activity failed and the risk is a reality. This task is done by Project manager when risk becomes reality and causes severe problems. If the project manager effectively uses project mitigation to remove risks successfully then it is easier to manage the risks. This shows that the response that will be taken for each risk by a manager. The main objective of the risk management plan is the risk register. This risk register describes and focuses on the predicted threats to a software project.

The RMMM Plan

· ***Risk Mitigation, Monitoring and Management Plan*** (RMMM) - documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan.

· Alternative to RMMM - ***risk information sheet*** (RIS)

RIS is maintained using a database system, so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily.

1. Risk monitoring is a project tracking activity

2. Three primary objectives:

   1. assess whether predicted risks do, in fact, occur
   2. ensure that risk aversion steps defined for the risk are being properly applied
   3. collect information that can be used for future risk analysis.

· Problems that occur during a project can be traced to more than one risk.

- Another job of risk monitoring is to attempt to allocate *origin* (what risk(s) caused which problems throughout the project).

QUALITY MANAGEMENT

Software Quality

Software quality product is defined in term of its fitness of purpose. That is, a quality product does

precisely what the users want it to do.

The modern view of a quality associated with a software product several quality methods

such as the following:

Portability: A software device is said to be portable, if it can be freely made to work in various

operating system environments, in multiple machines, with other software products, etc.

Usability: A software product has better usability if various categories of users can easily invoke the

functions of the product.

Reusability: A software product has excellent reusability if different modules of the product can quickly be

reused to develop new products.

Correctness: A software product is correct if various requirements as specified in the SRS document have

been correctly implemented.

Maintainability: A software product is maintainable if bugs can be easily corrected as and when they show

up, new tasks can be easily added to the product, and the functionalities of the product can be easily

modified, etc.

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 tocomplete 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
Degreetowhichdesignspecificationsarefollowedinmanufacturingtheproduct
Quality control Series of inspections, reviews, and tests used to ensure

conformance of a work product toits specifications
Quality assurance
Consists of a set of auditing and reporting functions that assess the effectivenessand completeness of quality control activities

Software Quality Assurance

Software Quality Assurance (SQA) **is simply a way to assure quality in the software.**

Software quality assurance focuses on:
- software’s portability
- software’s usability
- software’s reusability
- software’s correctness
- software’s maintainability
- software’s error control

Software Quality Assurance has:
1. A quality management approach
2. Formal technical reviews
3. Multi testing strategy
4. Effective software engineering technology
5. Measurement and reporting mechanism

SQA Activities

Software quality assurance is composed of a variety of functions associated with two different constituencies?

the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, record keeping, analysis, and reporting.

Following activities are performed by an independent SQA group:

1. **Prepares an SQA plan for a project:** The program is developed during project planning and is
2. reviewed by all stakeholders. The plan governs quality assurance activities performed by the
3. software engineering team and the SQA group. The plan identifies calculation to be performed,
4. audits and reviews to be performed, standards that apply to the project, techniques for error
5. reporting and tracking, documents to be produced by the SQA team, and amount of feedback provided
6. to the software project team.
7. **Participates in the development of the project's software process description:** The software team

selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g. ISO-9001), and other parts of the software project plan.

8. **Reviews software engineering activities to verify compliance with the defined software process:** The SQA group identifies, reports, and tracks deviations from the process and verifies that corrections have been made.
9. **Audits designated software work products to verify compliance with those defined as a part of the software process:** The SQA group reviews selected work products, identifies, documents and tracks deviations, verify that corrections have been made, and periodically reports the results of its work to the project manager.
10. **Ensures that deviations in software work and work products are documented and handled according to a documented procedure:** Deviations may be encountered in the project method, process description, applicable standards, or technical work products.
11. **Records any noncompliance and reports to senior management:** Non- compliance items are tracked until they are resolved.

Software Review is systematic inspection of a software by one or more individuals who work together to
find and resolve errors and defects in the software during the early stages of Software Development Life Cycle (SDLC).

Formal Technical Review (FTR) is a software quality control activity performed by software engineers.

Objectives of formal technical review (FTR): Some of these are:

- Useful to uncover error in logic, function and implementation for any representation of the
- software.

- The purpose of FTR is to verify that the software meets specified requirements.
- To ensure that software is represented according to predefined standards.

The review meeting: Each review meeting should be held considering the following constraints-
Involvement of people:

1. Between 3, 4 and 5 people should be involve in the review.
2. Advance preparation should occur but it should be very short that is at the most 2 hours of work for every person.
3. The short duration of the review meeting should be less than two hour. Gives these constraints, it should be clear that an FTR focuses on specific (and small) part of the overall software.

Review reporting and record keeping :-

1. During the FTR, the reviewer actively records all issues that have been raised.
2. At the end of the meeting all these issues raised are consolidated and a review list is prepared.
3. Finally, a formal technical review summary report is prepared.

Review guidelines :- Guidelines for the conducting of formal technical reviews should be established in advance. These guidelines must be distributed to all reviewers, agreed upon, and then followed. A review that is unregistered can often be worse than a review that does not minimum set of guidelines for FTR.

1. Review the product, not the manufacture (producer).
2. Take written notes (record purpose)
3. Limit the number of participants and insists upon advance preparation.
4. Develop a checklist for each product that is likely to be reviewed.
5. Allocate resources and time schedule for FTRs in order to maintain time schedule.
6. Conduct meaningful training for all reviewers in order to make reviews effective.
7. Reviews earlier reviews which serve as the base for the current review being conducted.
8. Set an agenda and maintain it.
9. Separate the problem areas, but do not attempt to solve every problem notes.
10. Limit debate and rebuttal.

Statistical Quality Assurance

Information about software defects is collected and categorized. Each defect is traced back to its cause Using the Pareto principle (80% of the defects can be traced to 20% of the causes) 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 Threecore steps:
Define customer requirements, deliverables, and project goals via well-defined methods of customer communication.
Measure each existing process and its output to determine current quality performance (e.g., computedefect metrics)
Analyze defect metrics and determine vital few causes.

For an existing process that needs improvement
Improve process by eliminating the root causes for defects
Control future work to ensure that future work does not reintroduce causes of
defectsIf new processes are being developed
Design each new process to avoid root causes of defects and to meet customer
requirementsVerify 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 failureMTTR = mean time to repair
Availability = [MTTF / (MTTF + MTTR)] x 100%

Software Reliability

Software Reliability means **Operational reliability**. It is described as the ability of a system or component to perform its required functions under static conditions for a specific period.

Software reliability is also defined as the probability that a software system fulfills its assigned task in a given environment for a predefined number of input cases, assuming that the hardware and the input are free of error.

Software Reliability is an essential connect of software quality, composed with functionality, usability, performance, serviceability, capability, installability, maintainability, and documentation. Software Reliability is hard to achieve because the complexity of software turn to be high. While any system with a high degree of complexity, containing software, will be hard to reach a certain level of reliability, system developers tend to push complexity into the software layer, with the speedy growth of system size and ease of doing so by upgrading the software.

ISO 9000 Certification

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

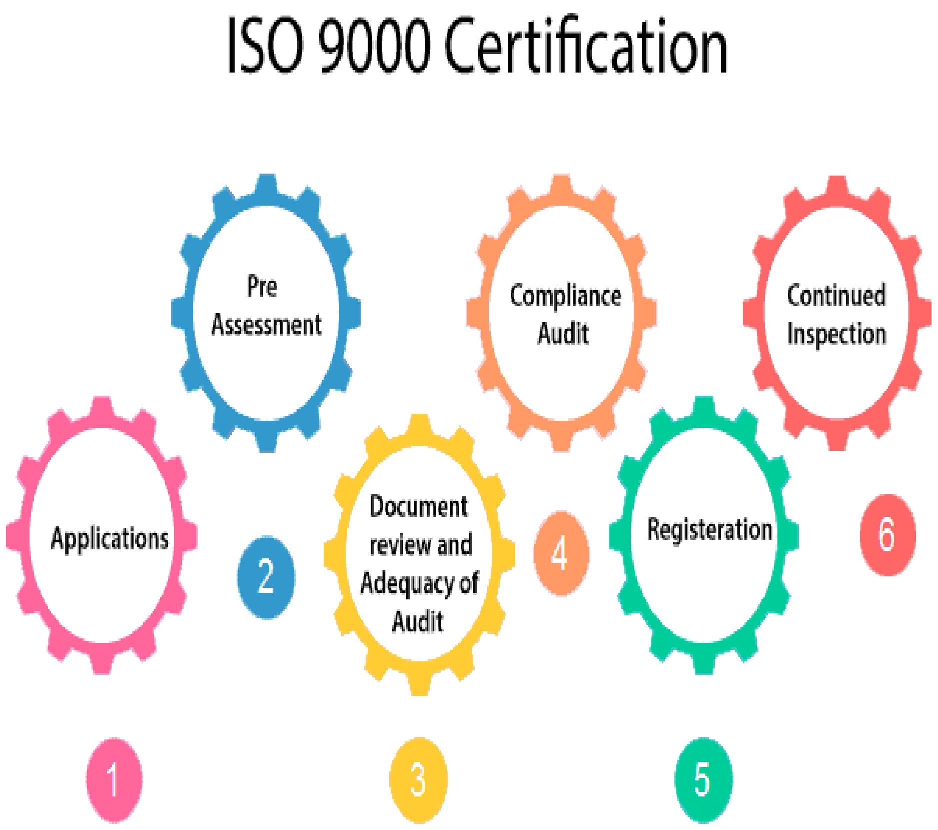
contract between independent parties. The ISO 9000 standard determines the guidelines for

maintaining a quality system.

Types of ISO 9000 Quality Standards

1. **ISO 9001:** This standard applies to the organizations engaged in design, 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 those products. Therefore, ISO 9002 does not apply to software development organizations.
3. **ISO 9003:** This standard applies to organizations that are involved only in the installation and

   testing of the products. For example, Gas companies.



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 by time.