Software engineering

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 OFSOFTWARE

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 serviceother programs
- **Embedded software--** resides in read-only memory and is used to control products and systems for the consumer and industrial markets.
- **Artificial intelligence software.** Artificial intelligence (AI) software makes use of nonnumeric algorithms to solve complex problems that are not amenable to computation or straightforwardanalysis
- **Engineering and scientific software.** Engineering and scientific software have been characterized by "number crunching" algorithms.

LEGACY SOFTWARE

Legacy software are older programs that are developed decades ago. The quality of legacy software is poor because it has inextensible design, convoluted code, poor and nonexistent documentation, test cases and results that are not achieved.

As time passes legacy systems evolve due to following reasons:

- The software must be adapted to meet the needs of new computing environment or technology.
- The software must be enhanced to implement new business requirements.
- The software must be extended to make it interoperable with more modern systems or database
- The software must be rearchitected to make it viable within a network environment.

SOFTWARE MYTHS

Myths are widely held but false beliefs and views which propagate misinformation and confusion.

Three types of myth are associated with software:

- Management myth
- Customer myth
- Practitioner's myth

MANAGEMENT MYTHS

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

CUSTOMER MYTHS

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

PRACTITIONER'S MYTH

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

SOFTWARE ENGINEERING-A LAYERED TECHNOLOGY

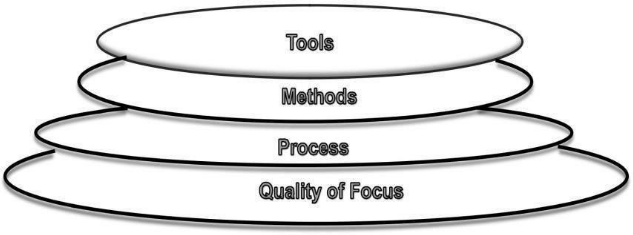


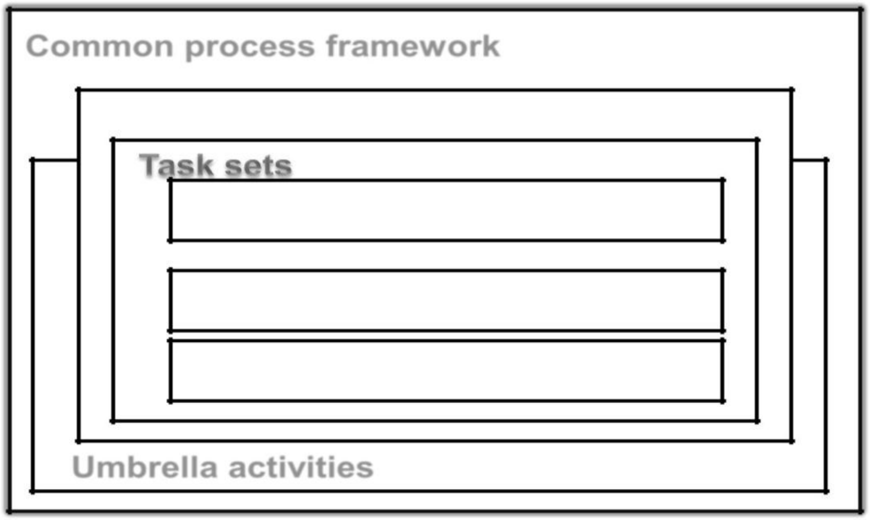
Fig: Software Engineering-A layered technology

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

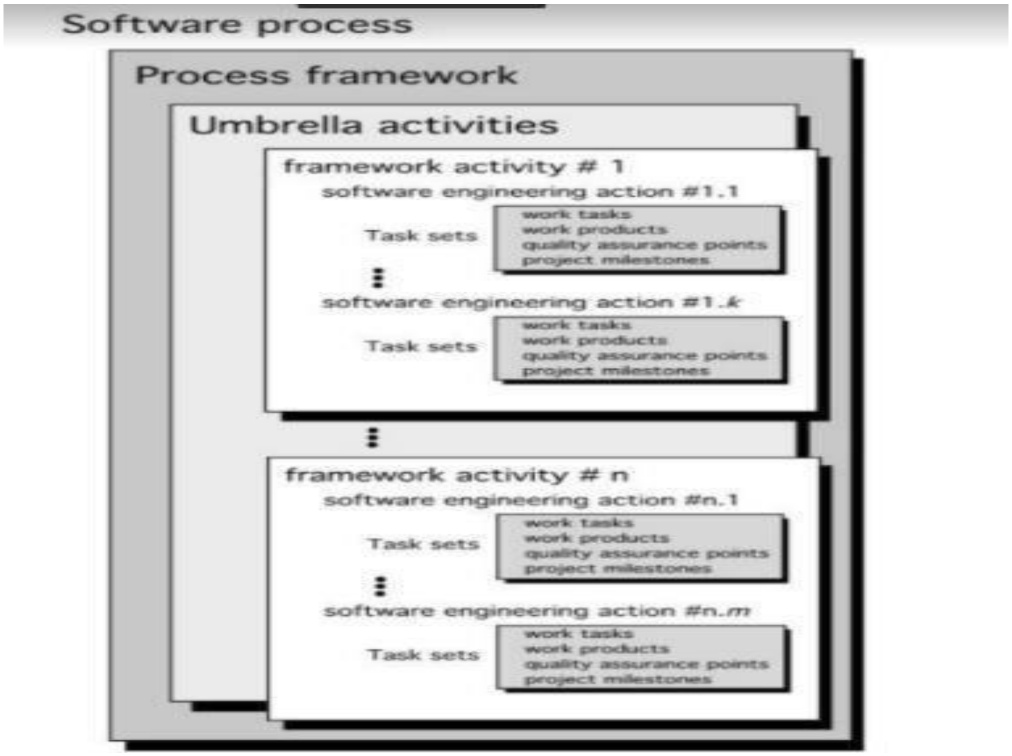
- Establishes the foundation for a complete software process
- Identifies a number of framework activities applicable to all software projects
- Also include a set of umbrella activities that are applicable across the entire software process.



A PROCESS FRAMEWORK comprises of :

Common process framework Umbrella activities Framework activities

Tasks, Milestones, deliverables SQA points



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 qualityassurance
- Software configuration management
- Document preparation and production
- Reusabilitymanagement
- Measurement
- Risk management

CAPABILITY MATURITY MODEL INTEGRATION(CMMI)

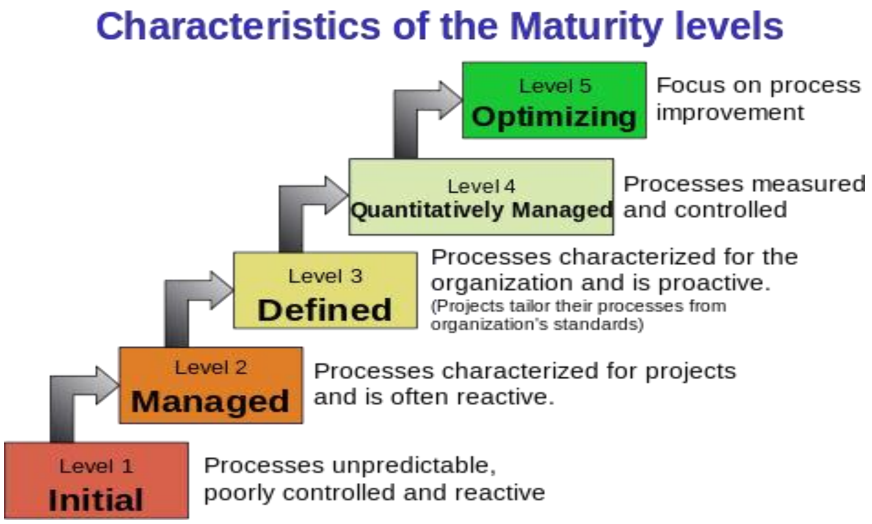
- Developed by SEI(Software Engineering institute)
- Assess the process model followed by an organization and rate the organization with different levels
- A set of software engineering capabilities should be present as organizations reach different levels of process capability and maturity.

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

Continuous model:
-Lets 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.

CMMI

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


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

- 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 and other methods for process improvement.

CMMI - Staged model

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

PROCESS 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 process

- ➢ PLANNING
- ➢ HIGH LEVEL DESIGN
- ➢ HIGH LEVEL DESIGN REVIEW
- ➢ DEVELOPMENT
- ➢ POSTMORTEM

Personal and Team Software Process

Team software process Goal of TSP

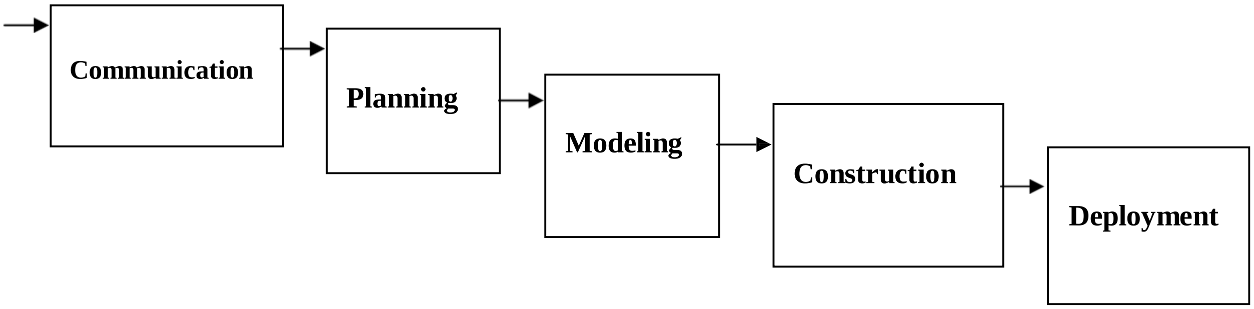
- Build self-directed teams
- Motivate the teams
- Acceptance of CMM level 5 behavior as normal to accelerate software process improvement
- Provide improvement guidance to high maturity organization

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 level and progresses through analysis, design, code and testing

PROBLEMS IN WATERFALLMODEL

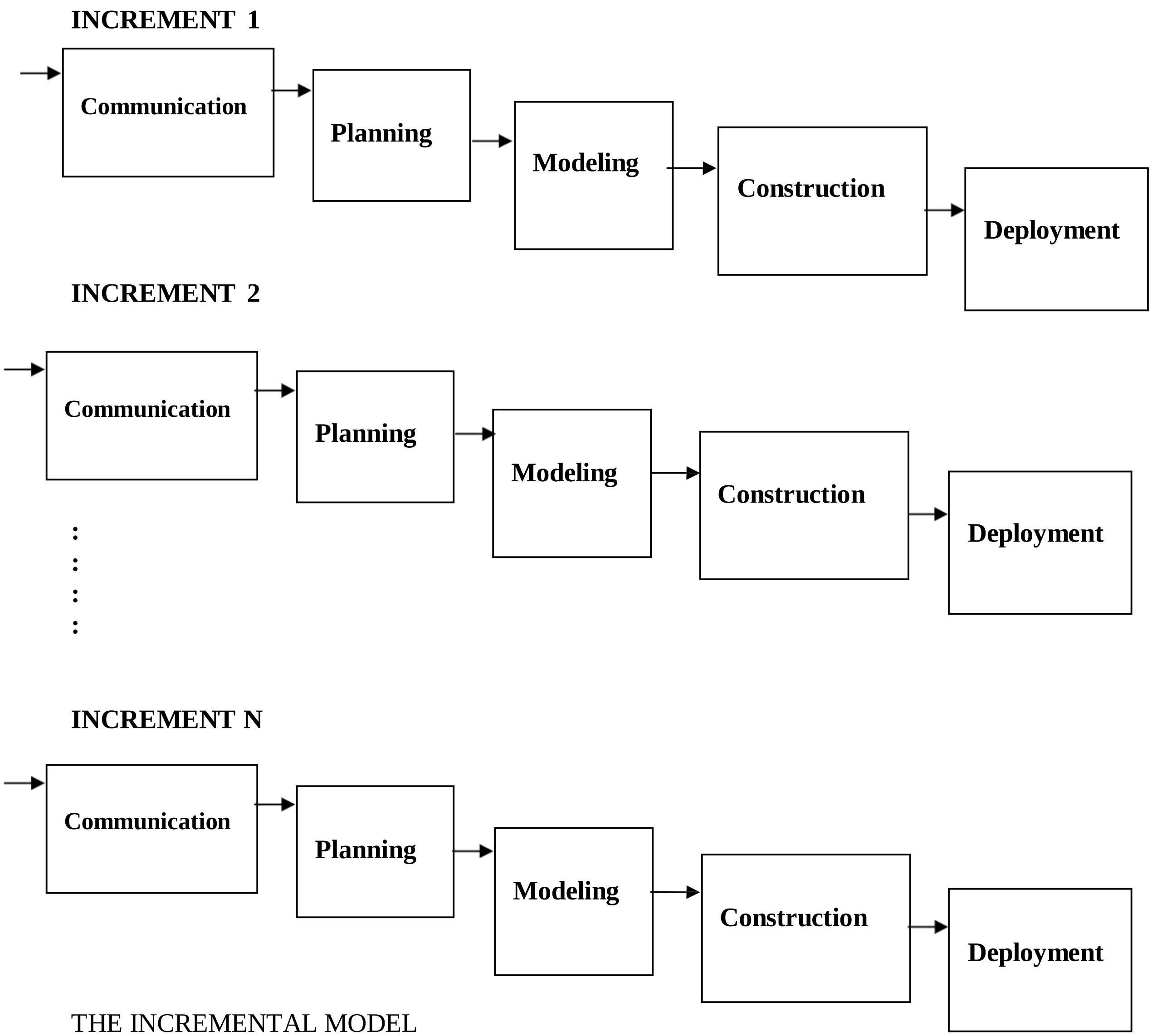
- Real projects rarely follow the sequential flow since they are always iterative
- The model requires requirements to be explicitly spelled out in the beginning, which is often difficult
- A working model is not available until late in the project time plan

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 quickly
- Functionality expanded further in later releases
- Software is developed in increments

The Incremental Model

- ➢ Communication
- ➢ Planning
- ➢ Modeling
- ➢ Construction
- ➢ Deployment



THE INCREMENTAL MODEL

- 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 product to better meet the needs of customer
- Process is repeated until the complete product is produced

THE RAD (Rapid Application Development) MODEL

- An incremental software process model
- Having a short development cycle
- High-speed adoption of the waterfall model using a component based construction approach
- Creates a fully functional system within a very short span time of 60 to 90 days

The RAD Model consists of the following phases:

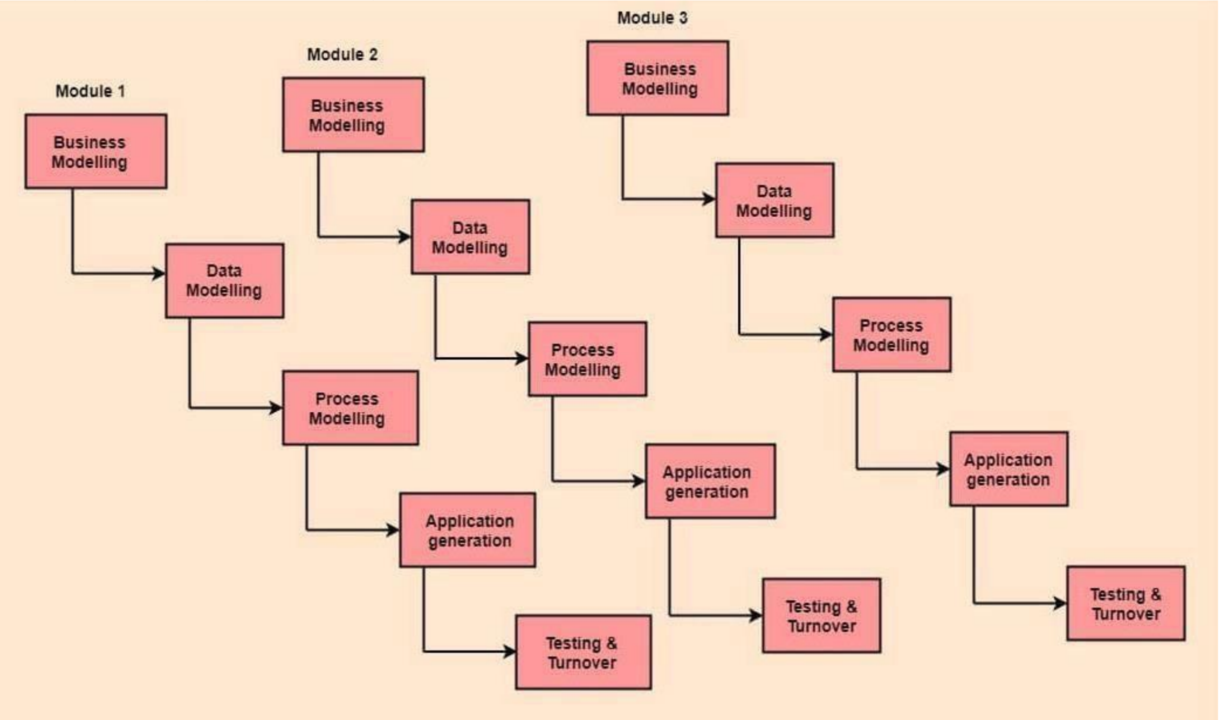
Communication Planning Construction

Component reuses automatic code generation testing

Modeling

Business modeling Data modeling Process modeling

Deployment integration delivery feedback



THE RAD MODEL

- Multiple software teams work in parallel on different functions
- Modeling encompasses three major phases: Business modeling, Data modeling and process modeling
- Construction uses reusable components, automatic code generation and testing

Problems in RAD

- Requires a number of RAD teams
- Requires commitment from both developer and customer for rapid-fire completion of activities
- Requires modularity
- Not suited when technical risks are high

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

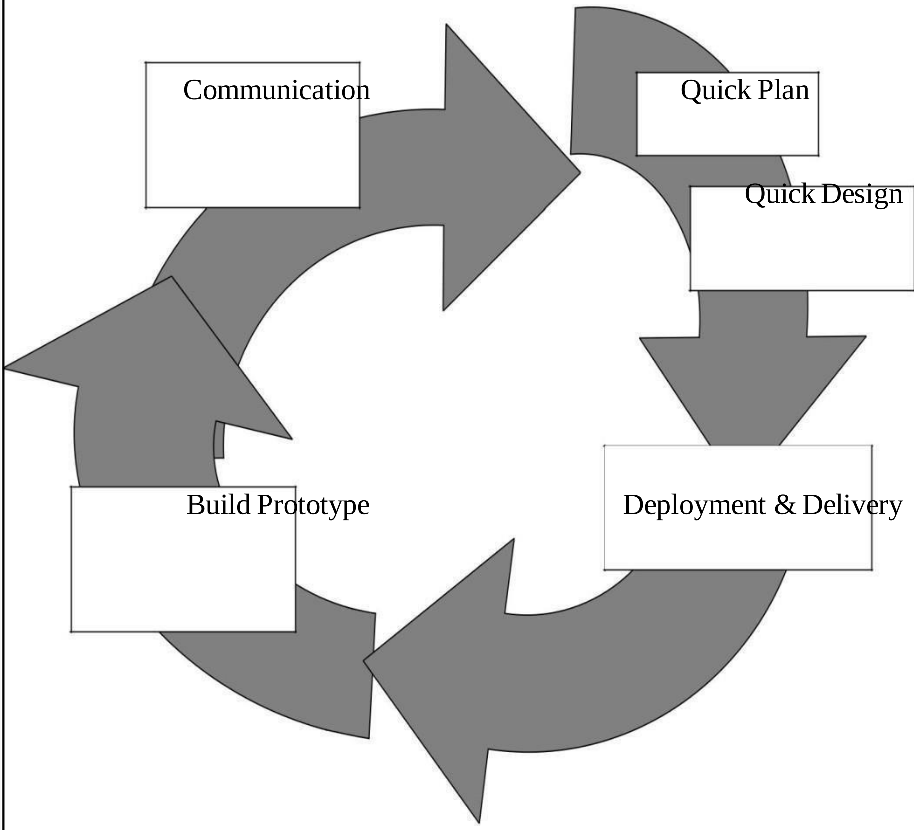
Types of evolutionary models

– Prototyping
– Spiral model
– Concurrent development model

PROTOTYPING

- Mock up or model(throw away version) of a software product
- Used when customer defines a set of objective but does not identify input, output, or processing requirements
- Developer is not sure of:

– efficiency of an algorithm
adaptability of an operating system
– human/machine interaction



STEPS IN PROTOTYPING

- Begins with requirement gathering
- Identify whatever requirements are known
- Outline areas where further definition is mandatory
- A quick design occur
- Quick design leads to the construction of prototype
- Prototype is evaluated by the customer

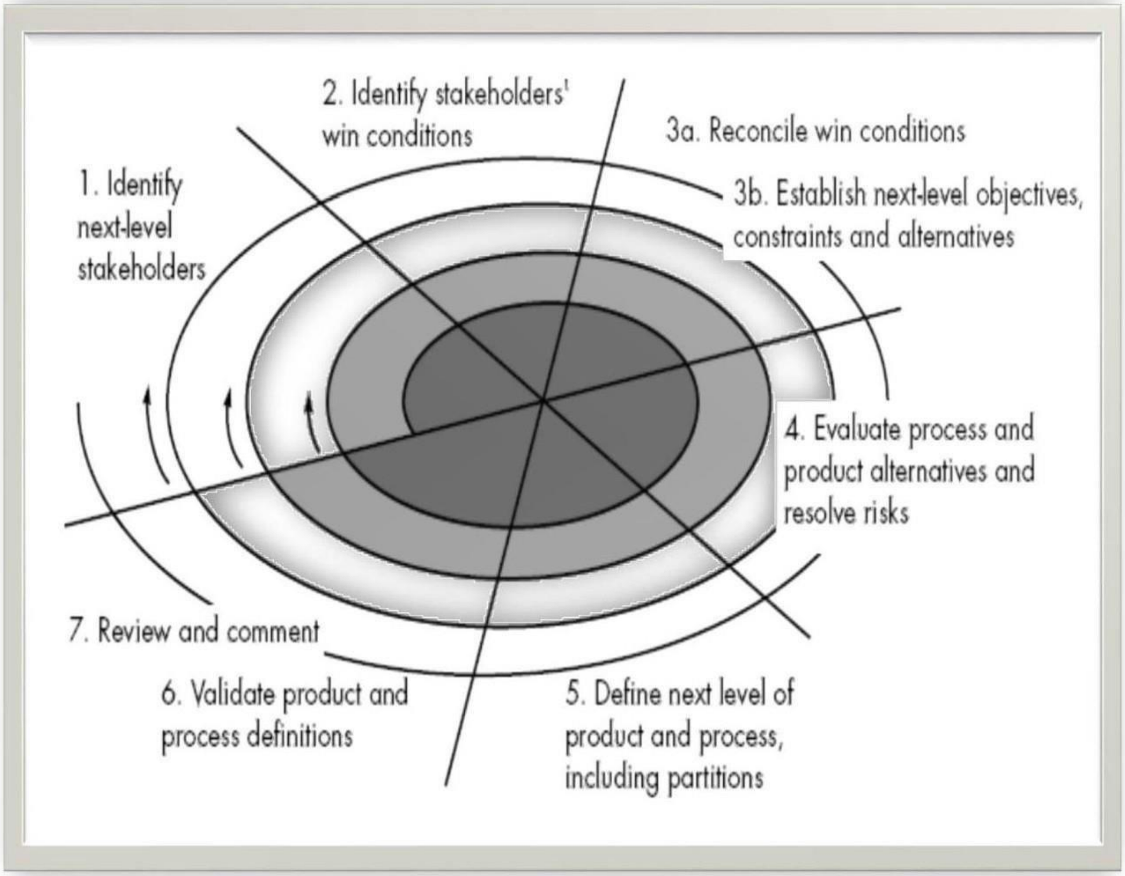
- Requirements are refined
- Prototype is turned to satisfy the needs of customer

LIMITATIONS OF PROTOTYPING

- In a rush to get it working, overall software quality or long term maintainability are generally overlooked
- Use of inappropriate OS or PL
- Use of inefficient algorithm

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
- And sophisticated version of software

THE CONCURRENT DEVELOPMENT MODEL

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

A FINAL COMMENT ON EVOLUTIONARY PROCESS

- Difficult in project planning
- Speed of evolution is not known

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

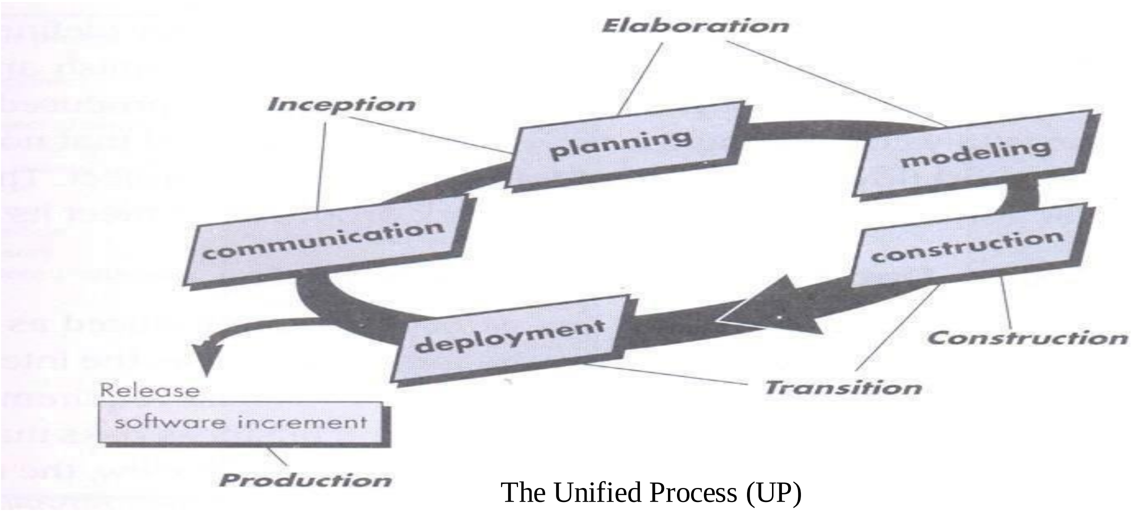
- Requirement is balance between high quality and flexibility and extensibility

THE UNIFIED PROCESS

Evolved by Rumbaugh, Booch, Jacobson. Combines the best features their OO models. Adopts additional features proposed by other experts. Resulted in Unified Modeling Language (UML). Unified process developed Rumbaugh and Booch. A framework for Object-Oriented Software Engineering using UML

PHASES OF UNIFIED PROCESS

- INCEPTION PHASE
- ELABORATION PHASE
- CONSTRUCTION PHASE
- TRANSITION PHASE



The Unified Process (UP)

UNIFIED PROCESS WORK PRODUCTS

Tasks which are required to be completed during different phases

1. **Inception Phase**

*Vision document
*Initial Use-Case model
*Initial Riskassessment
*Project Plan

2. **Elaboration Phase**

*Use-Case model
*Analysis model
*Software Architecture description
*Preliminary design model
*Preliminary model

3. **Construction Phase**

*Design model
*System components
*Test plan and procedure
*Test cases
*Manual

4. **Transition Phase**

*Delivered software increment
*Beta test results
*General user feedback

Software engineering

Unit II

System requirements

- Requirements analysis is very critical process that enables the success of a system or software project to be assessed.

- Requirements are generally split into two types:

- *Functional* and

- *Non-functional requirements*.

Functional Requirements:

- These are the requirements that the end user specifically demands as basic facilities that the system should offer. All these functionalities need to be necessarily incorporated into the system as a part of the contract.

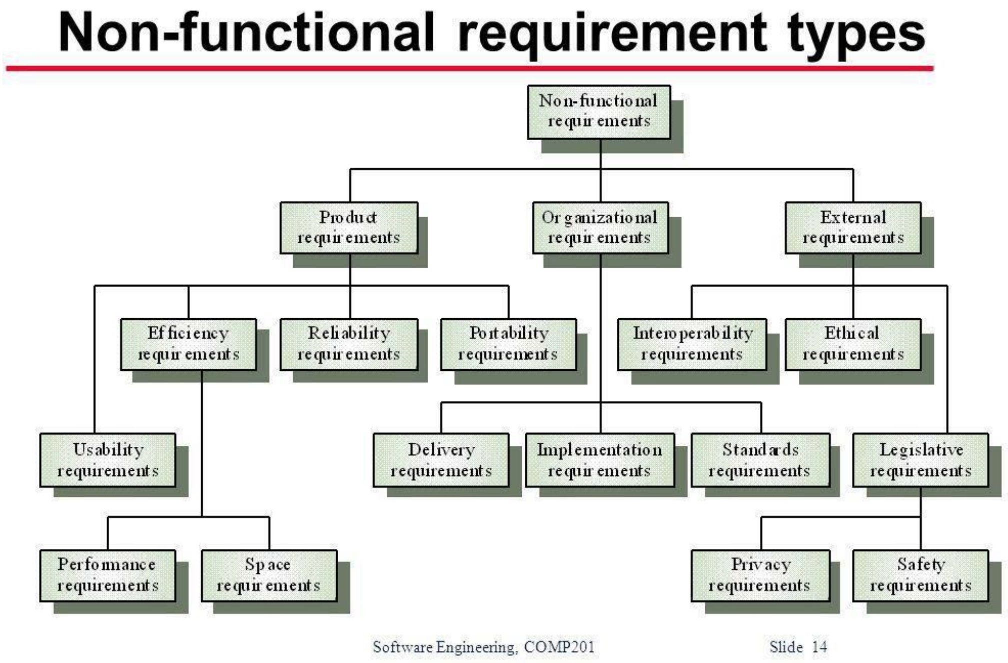
- These are represented or stated in the form of input to be given to the system, the operation performed and the output expected.

- They are basically the requirements stated by the user which one can see directly in the final product, unlike the non-functional requirements.

NON- FUNCTIONAL REQUIREMENTS:

- NON- FUNCTIONAL REQUIREMENTS, As the name suggests, are requirements that are not directly concerned with the specific functions delivered by the system.

- These are basically the quality constraints that the system must satisfy according to the project contract. The priority or extent to which these factors are implemented varies from one project to other. They are also called non-behavioral requirements.


Non-functional requirement types

Software Engineering, COMP201 Slide 14

The type of non-functional requirements are:

- **Product requirements**: These requirements specify product behavior.
- **Organizational requirements**: These requirements are derived from policies and procedures in the customer's and developer's organization.
- **External requirements**: This broad heading covers all requirements that are derived from factors external to the system and its development process.

USER REQUIREMENTS:

- The user requirements for a system should describe the functional and non-functional requirements so that they are understandable by system users without detailed technical knowledge.
- If you are writing user requirements , you should not use software jargons, structured notations or formal notations or describe the requirements by describing the system implementation.

- You should write user requirements in simple language , with simple tables and forms and intuitive diagrams.

Various problems can arise when requirements are written in natural language sentences in a text document:

1. **Lack of clarity**: It is some times difficult to use language in a precise and unambiguous way without making the document wordy and difficult to read.
2. **Requirements confusion**: functional requirements, non- functional requirements, system goals and design information may not be clarity distinguished.
3. **Requirements amalgamation**: several different requirements may be expressed together as a single requirement.

System Requirements:

- System requirements are expanded version of the user requirements that are used by software engineers as the starting point for the system design.
- They add detail and explain how the user requirements should be provided by the system.
- System requirements should simply describe the external behaviour of the system and its operational constraints.

Natural language is often used to write system requirements specifications as well as user requirements. However because system requirements are more detailed than user requirements, natural language specifications can be confusing and hard to understand:

1. Natural language understanding relies on the specification readers and writers using the same words for the same concept. This leads to misunderstanding because of the ambiguity of natural language.
2. A Natural language requirements specification is overflexible. You can say the same thing in completely different ways.
3. There is no easy way to modularise Natural language requirements. It may be difficult to find all related requirements.

Structured language specifications:

Structured natural language is a way of writing system requirements where the freedom of the requirements written is limited and all requirements are written in a standard way.

When a standard form is used for specifying functional requirements, the following information should be included:

- Description of the function or entry being specified.
- Description of its inputs and where these come form
- Description of its outputs and where these go to
- Indication of what other entities are used
- Description of the action to be taken
- If a functional approach is used, a pre-condition setting out what must be true before the function is called and post-condition specifying what is true after the function is called.
- Description of the of the operation.

Interface requirements:

- Almost all software systems must operate with existing systems that have already been implemented and installed in an environment.
- If the new system and the existing system must work together, The interfaces of existing systems have to be precisely specified.
- These specifications should be defined early in the process and included in the requirements document.

There are three types of interface that may have to be:

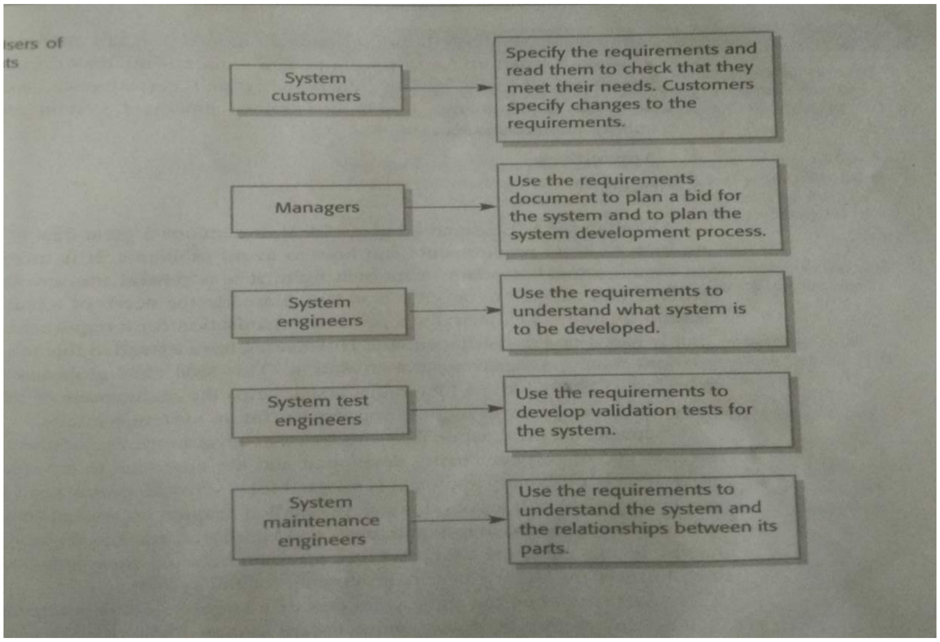
- Procedural interface: Where existing programs or sub-systems offer a range of services that accessed by calling interface procedures. These interfaces are sometimes called Application Programming Interfaces(APIs)
- Data structures: That are passed from one sub-system to another. Graphical data models are the best notations for this type of description.
- Representation of data: That have been established for an existing sub-system. These interfaces are most common in embedded, real-time-system. Some programming languages such as Ada support this level of specification.

SOFTWARE REQUIREMENTS DOCUMENT:

The software requirements document(sometimes called the software requirements specification or SRS) is the official statement of what the system developers should implement.

It should include both the user requirements for a system and a detailed specification of the system requirements.

The users of a requirements document:



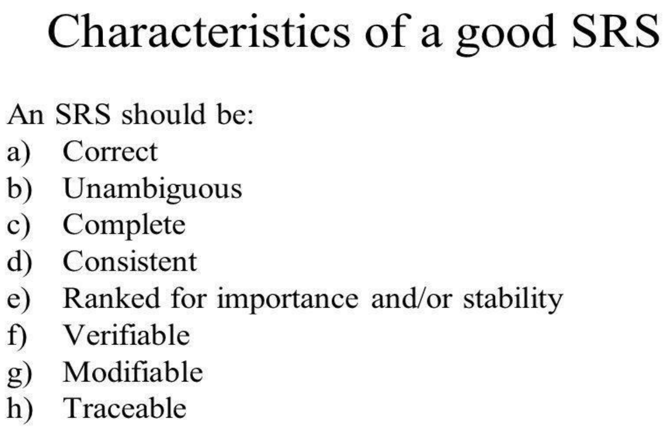
System customers → Specify the requirements and read them to check that they meet their needs. Customers specify changes to the requirements.

Managers → Use the requirements document to plan a bid for the system and to plan the system development process.

System engineers → Use the requirements to understand what system is to be developed.

System test engineers → Use the requirements to develop validation tests for the system.

System maintenance engineers → Use the requirements to understand the system and the relationships between its parts.

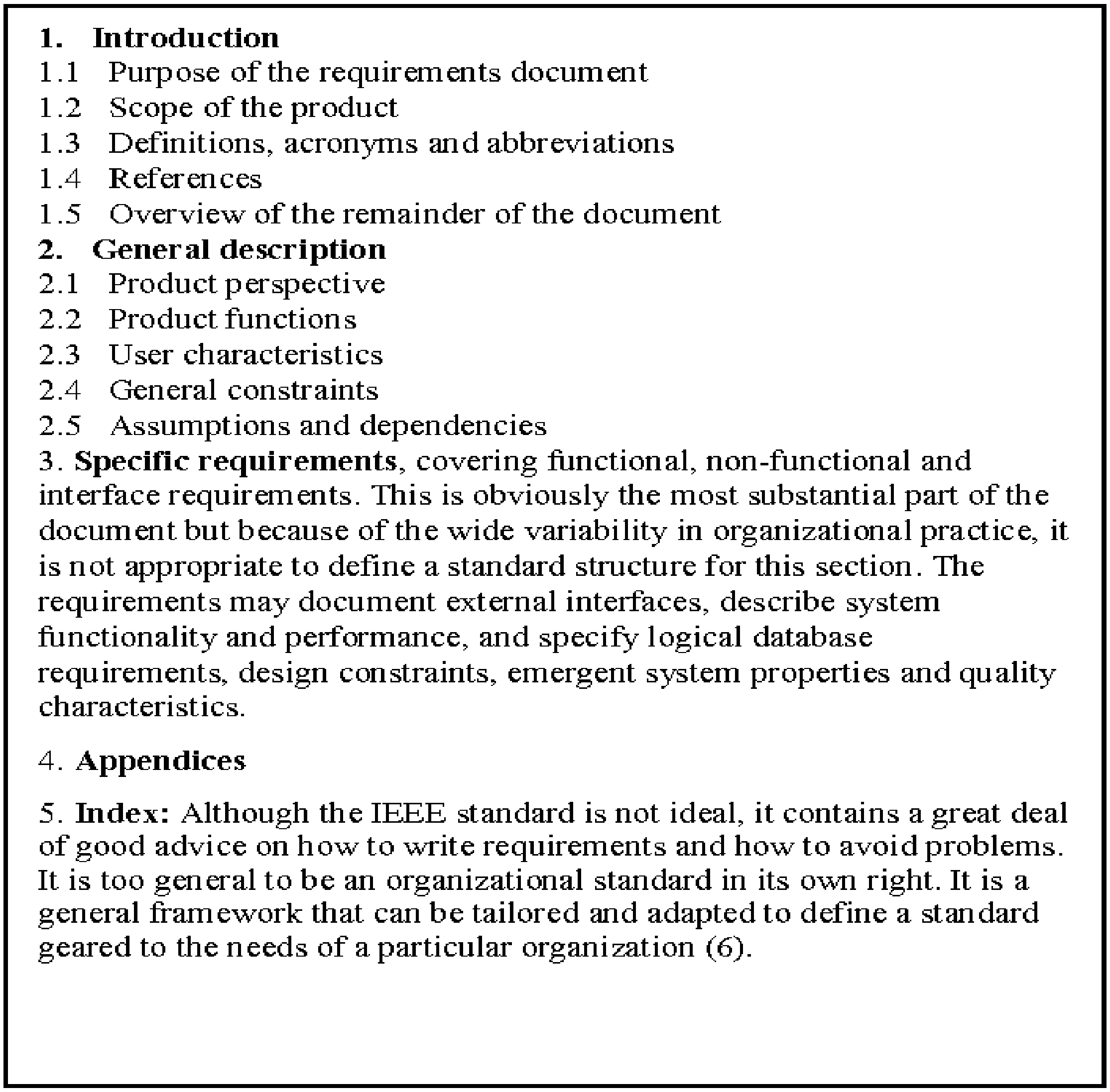


Characteristics of a good SRS

An SRS should be:

a) Correct
b) Unambiguous
c) Complete
d) Consistent
e) Ranked for importance and/or stability
f) Verifiable
g) Modifiable
h) Traceable

IEEE Standards suggests the following structure for requirements documents:



1. Introduction

1.1 Purpose of the requirements document
1.2 Scope of the product
1.3 Definitions, acronyms and abbreviations
1.4 References
1.5 Overview of the remainder of the document

2. General description

2.1 Product perspective
2.2 Product functions
2.3 User characteristics
2.4 General constraints
2.5 Assumptions and dependencies

3. **Specific requirements**, covering functional, non-functional and interface requirements. This is obviously the most substantial part of the document but because of the wide variability in organizational practice, it is not appropriate to define a standard structure for this section. The requirements may document external interfaces, describe system functionality and performance, and specify logical database requirements, design constraints, emergent system properties and quality characteristics.

4. **Appendices**

5. **Index:** Although the IEEE standard is not ideal, it contains a great deal of good advice on how to write requirements and how to avoid problems. It is too general to be an organizational standard in its own right. It is a general framework that can be tailored and adapted to define a standard geared to the needs of a particular organization (6).

Figure 1. The SRS structure according the IEEE 830-1998 standard

REQUIREMENTS ENGINEERING PROCESS

To create and maintain a system requirement document. The overall process includes four high level requirements engineering sub-processes:

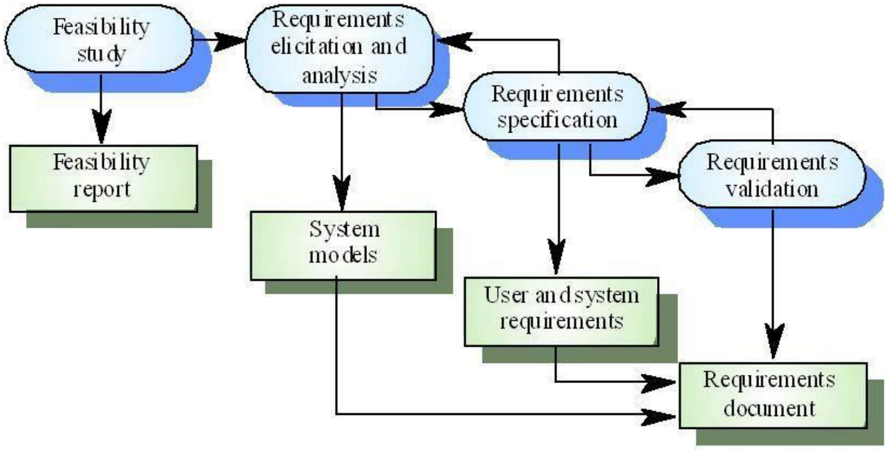
1. Feasibility study--Concerned with assessing whether the system is useful to the business

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

Requirement engineering

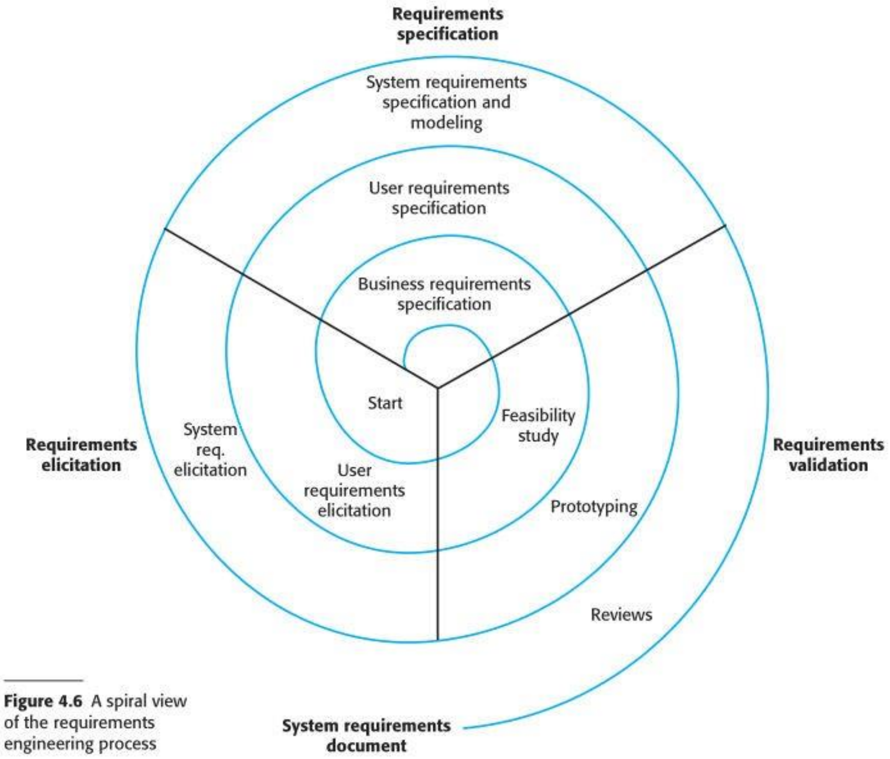


SPIRAL REPRESENTATION OF REQUIREMENTSENGINEERING PROCESS

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
2. Requirements specification
3. 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
Prototyping
Requirements validation
Reviews
System requirements document

Figure 4.6 A spiral view of the requirements engineering process

FEASIBILITY STUDIES

The requirement engineering process should starts with a feasibility study.
Starting point of the requirements engineering process

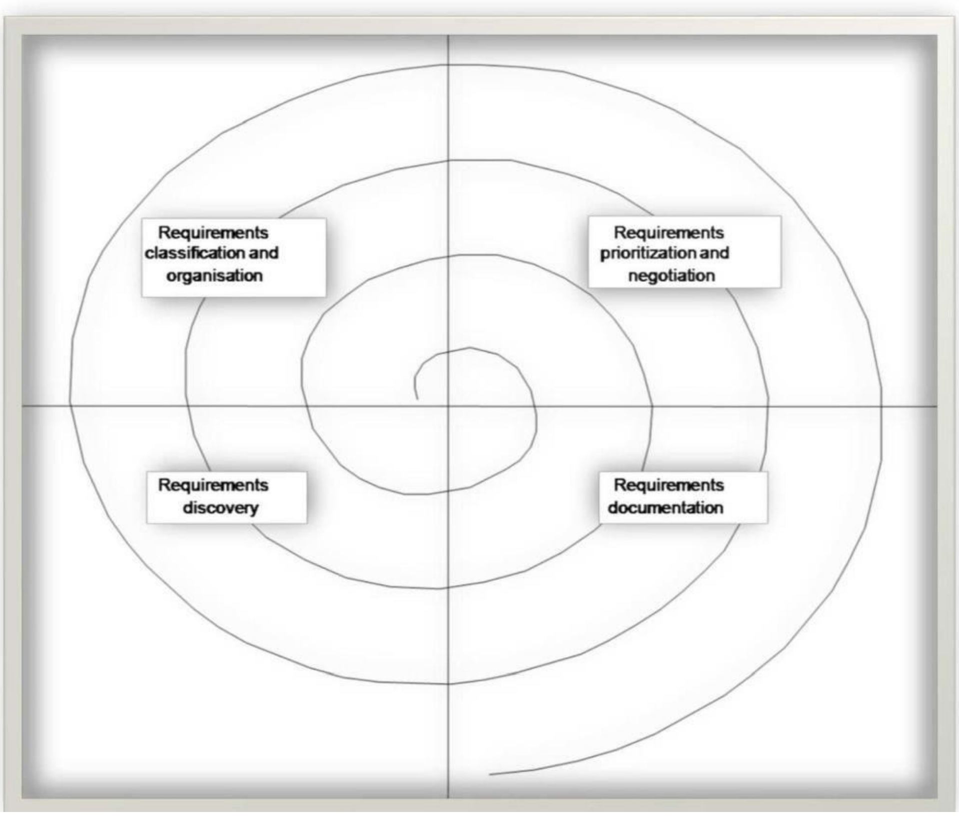
- **Input:** Set of preliminary business requirements, an outline description of the system and howthe system is intended to support business processes
- **Output:** Feasibility report that recommends whether or not it is worth carrying out further Feasibility report answers a number of questions:
  - ➢ Does the system contribute to the overall objective:
  - ➢ Can the system be implemented using the current technology and within given cost and schedule
  - ➢ 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.

5. the economic and business environment in which the analysis takes place is dynamic


Requirements classification and organisation
Requirements prioritization and negotiation
Requirements discovery
Requirements documentation

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 ofspiral

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

- **Interactor viewpoint**--Represents people or other system that interact directly with the system
- **Indirect viewpoint**--Stakeholders who influence the requirements, but don't use the system
- **Domain viewpoint**--Requirements domain characteristics and constraints that influence the requirements.

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

REQUIREMENTSVALIDATION

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

During the requirement validation process, checks should be carried out on the requirements in the requirements document.

These checks include:

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

VALIDATION TECHNIQUES:

- ❖ Requirements reviews – the requirements are analysed systematically by a team of reviewers.

  Reviewers check the following:

  (a) Verifiability: Testable
  (b) Comprehensibility
  (c) Traceability
  (d) Adaptability

- ❖ Prototyping- an executable model of the system is demonstrated to end- users and customers.
- ❖ Test- case generation – Requirements should be testable.

Requirements management

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

Reasons for requirements changes

a) Diverse Users community where users have different requirements and priorities
b) System customers and end users are different
c) Change in the business and technical environment after installation Two classes of requirements
d) Enduring requirements: Relatively stable requirements
e) Volatile requirements: Likely to change during system development process or during operation

Requirements management planning

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

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.

- **Source traceability**--Links the requirements to the stakeholders
- **Requirements traceability**--Links dependent requirements within the requirements document
- **Design traceability**-- Links from the requirements to the design module

CASE tools:

- **Requirement storage** – should maintained in a secure, managed data store.
- **Change management** – simplified if active tool support is available.
- **Traceability management** – allows related requirements to be discovered.

Requirements change management:

There are Three principle stages to a change management process:

1. **Problem analysis and change specification**-- Process starts with a specific change proposaland 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

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. Behavioral 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

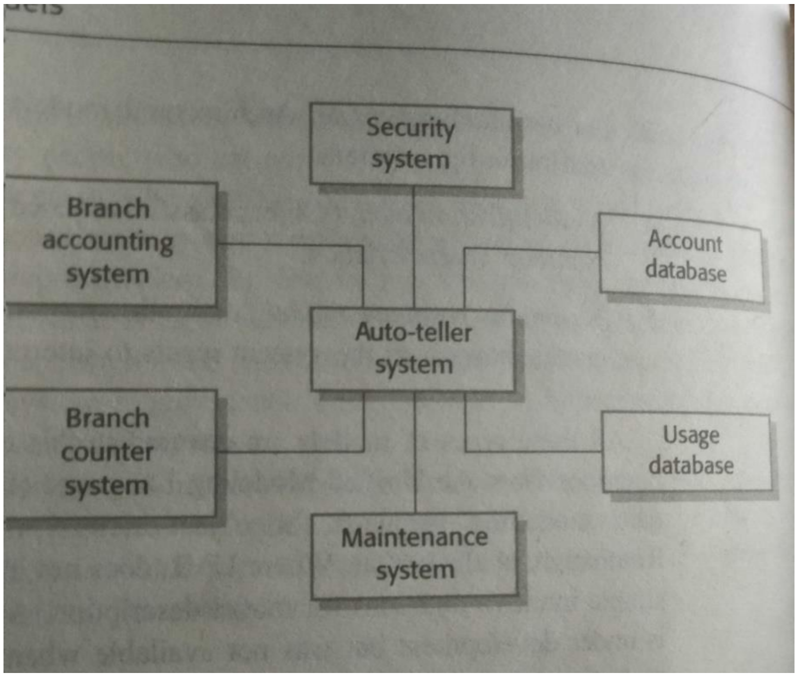
• Lines between rectangles indicate associations between subsystems Disadvantages

--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



Behavioral models

Describes the overall behavior of a system. Two types of behavioral model

1. Data Flow models 2.State machine models

Data flow models --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

transition from on state to another state

Disadvantage

-- Number of possible states increases rapidly for large system models

DATA MODELS

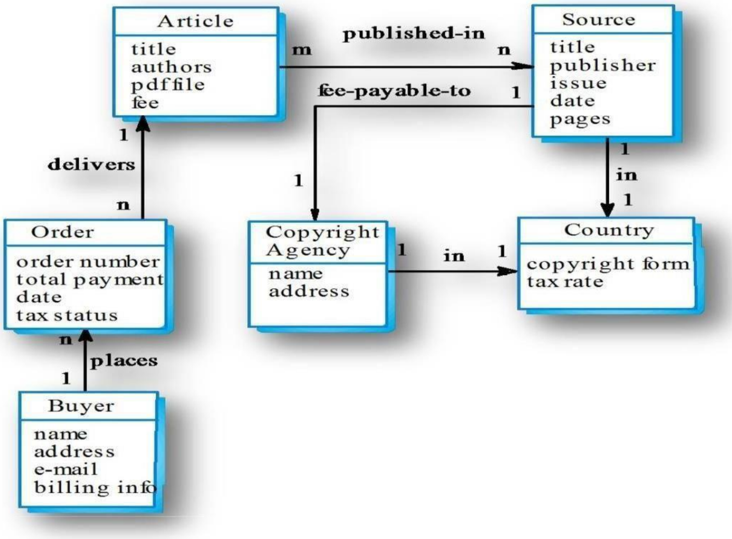
Used to describe the logical structure of data processed by the system. An entity-relation-attribute

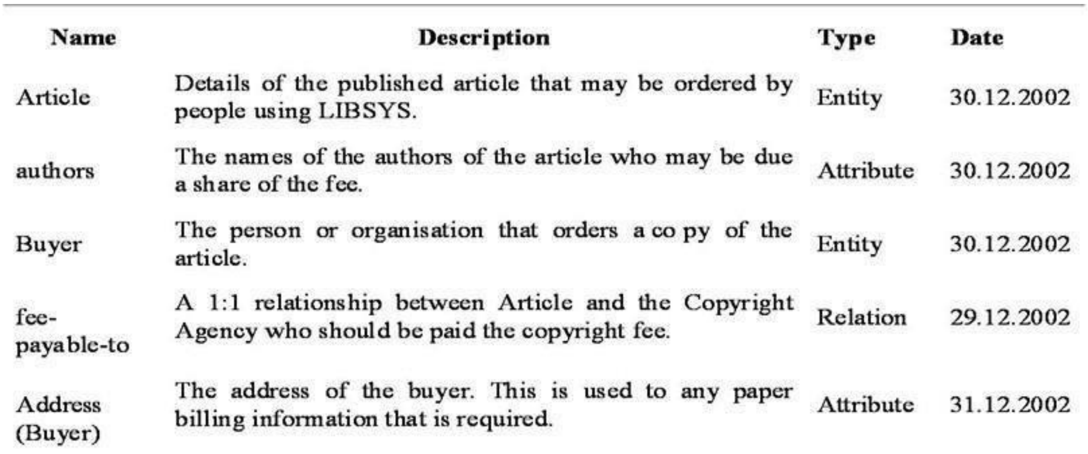
model sets out the entities in the system, the relationships between these entities and the entity attributes.

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.

Data dictionary entries





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 copy 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

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
The name of the object class in the top section

The class attributes in the middle section

The operations associated with the object class are in lower section.

Object name
Class attribute
Operation()

OBJECT MODELS INHERITANCE MODELS

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.

OBJECT MODELS OBJECT AGGREGATION

Some objects are grouping of other objects. An aggregate of a set of other objects. The classes representing these objects may be modeled using an object aggregation model A diamond shape on the source of the link represents the composition.

OBJECT-BEHAVIORAL MODEL
-- Shows the operations provided by the objects
-- Sequence diagram of UML can be used for behavioral modeling

UNIT III

DESIGN ENGINEERING

DESIGN PROCESS

- Software design is an iterative process through which requirements are translated into a “blueprint” for constructing the software.
- The blueprint depicts a holistic view of software. That is, the design represented at a high level of abstraction- a level that can directly traced to the specific system objective and more detailed data, functional, and behavioral requirements.

❑ McGlaughlin suggests three characteristics that serve as a guide for the evaluation of a good design:

1. The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
2. The design must be readable, understandable guide for those who generate code and for those who test and sequentially support the software.
3. The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

DESIGN QUALITY

QUALITY GUIDELINES

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

QUALITY ATTRIBUTES

❖ FURPS quality attributes

1. Functionality
   - Feature set and capabilities of programs
   - Security of the overall system
2. Usability
   - user-friendliness
   - Aesthetics
   - Consistency
   - Documentation
3. Reliability
   - Evaluated by measuring the frequency and severity of failure
   - Mean-time-to-failure(MTTF)
   - Recover from failure
4. Performance
   - Speed, response time, resource consumption, throughput, efficiency.
5. Supportability
   - Extensibility
   - Adaptability
   - Serviceability
   - maintainability

DESIGN CONCEPTS

- A set of fundamental software design concepts has evolved over the history of software engineering.
- Although the degree of interest in each concept has varied over the years, each has stood the test of time.
- Each provides the software designer with a foundation from which more sophisticated design methods can be applied.

Design concepts are:

1. Abstractions

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

1. **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

2. ARCHITECTURE

Structure organization of program components (modules) and their interconnection Architecture Models

- **Structural Models**-- An organized collection of program components
- **Framework Models**-- Represents the design in more abstract way
- **Dynamic Models**-- Represents the behavioral aspects indicating changes as a function of external events
- **Process Models--** Focus on the design of the business or technical process

3. PATTERNS

Provides a description to enables a designer to determine the followings:

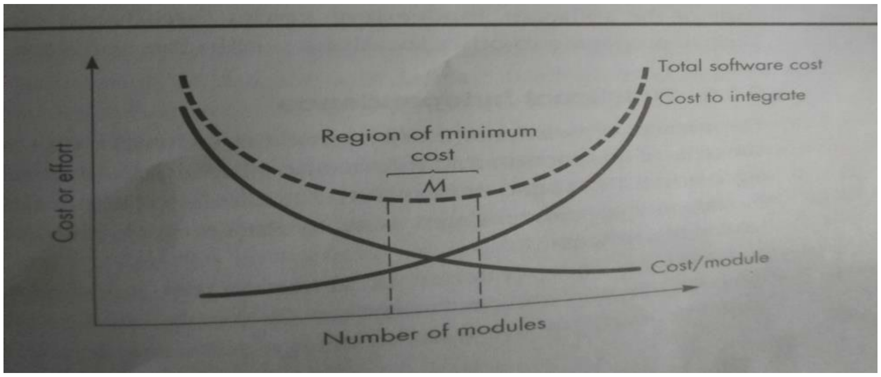
a) whether the pattern is applicable to the current work

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

4. **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 is cp1 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



5. 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.

6. 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:

✓ Cohesion

✓ 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.

7. 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.

8. 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.

9. DESIGN CLASSES

Class represents a different layer of design architecture. Five types of Design Classes

- **User interface class** -- Defines all abstractions that are necessary for human computer interaction
- **Business domain class** -- Refinement of the analysis classes that identity attributes and services to implement some of business domain
- **Process class** -- implements lower level business abstractions required to fully manage the business domain classes
- **Persistent class** -- Represent data stores that will persist beyond the execution of the software

System class -- Implements management and control functions to operate and communicate within the computer environment and with the outside world.

THE DESIGN MODEL

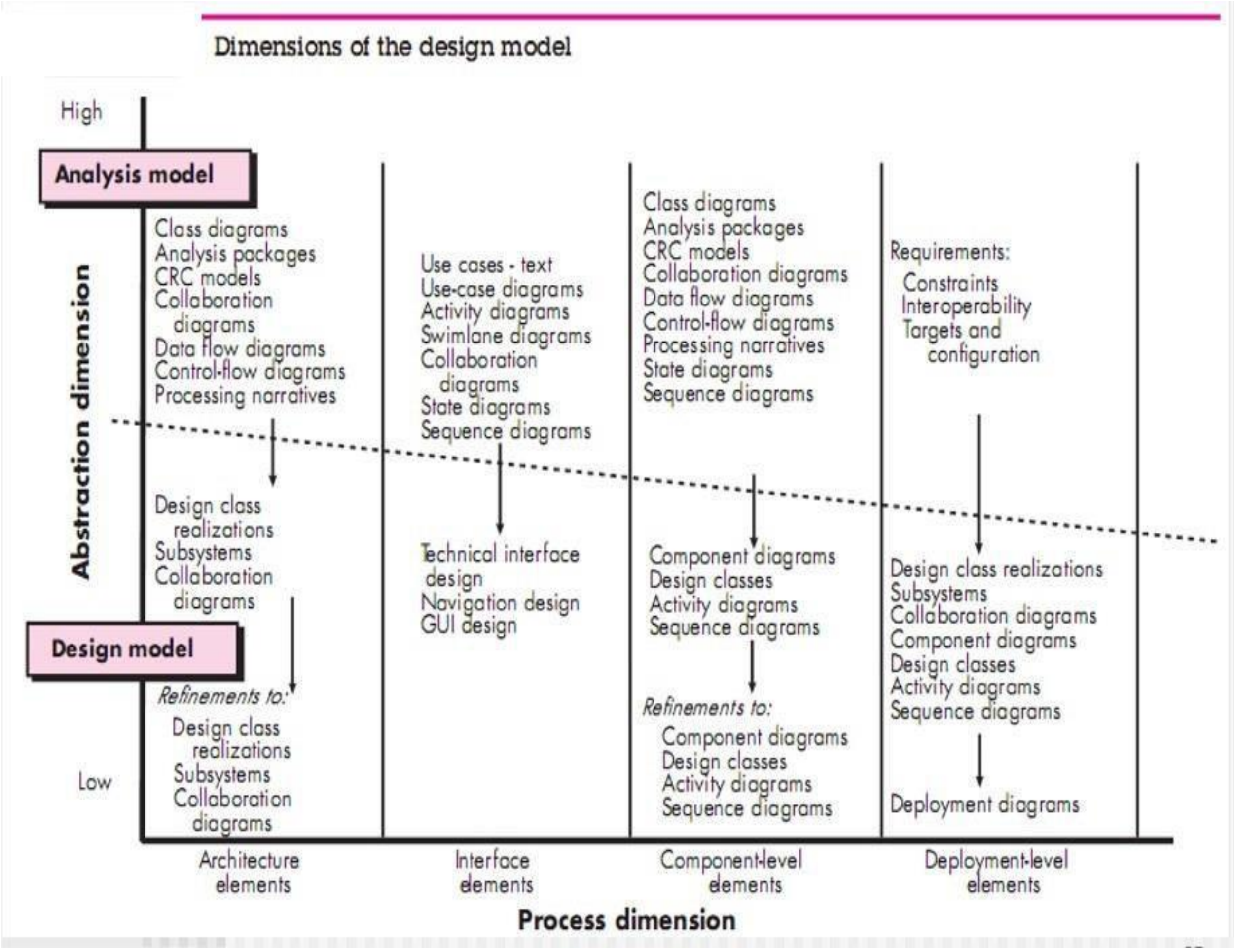
Introduction of Design Model

- The design model can be viewed in two different dimensions.
  - (Horizontally) The process dimension
    - It indicates the evolution of the parts of the design model as each design task is executed.
  - (Vertically) The abstraction dimension

  - It represents the level of detail as each element of the analysis model is transformed into the design model and then iteratively refined.

- The elements of the design model use many of the same UML diagrams that were used in the analysis model.

- The difference is that these diagrams are
  - Refined and elaborated as part of design;
  - More implementation-specific detail is provided,
  - Architectural structure and style, components that reside within the architecture,
  - Interfaces between the components and with the outside world are all emphasized.



1. <u>**Data Design Elements**</u>

- **Customer's/ User's View:**
  - Data Architecting (Creates a model of data that is represented at a high level of abstraction). (Build Architecture of Data)

- **Program Component Level:** The design of Data structure & algorithms.

- **Application Level:** Translate Data Model into a database.
- **Business Level:** Data warehouse(Reporting & Analysis of DB) & Data mining(Analysis).
- At last it means creation of Data Dictionary.

2. **Architectural Design Elements:**

Provides an overall view of the software product(Similar like Floor Plan of house)

- The architectural model [Sha96] is derived from three sources:

  (1) Information about the application domain for the software to be built;

  (2) Specific requirements model elements such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand;

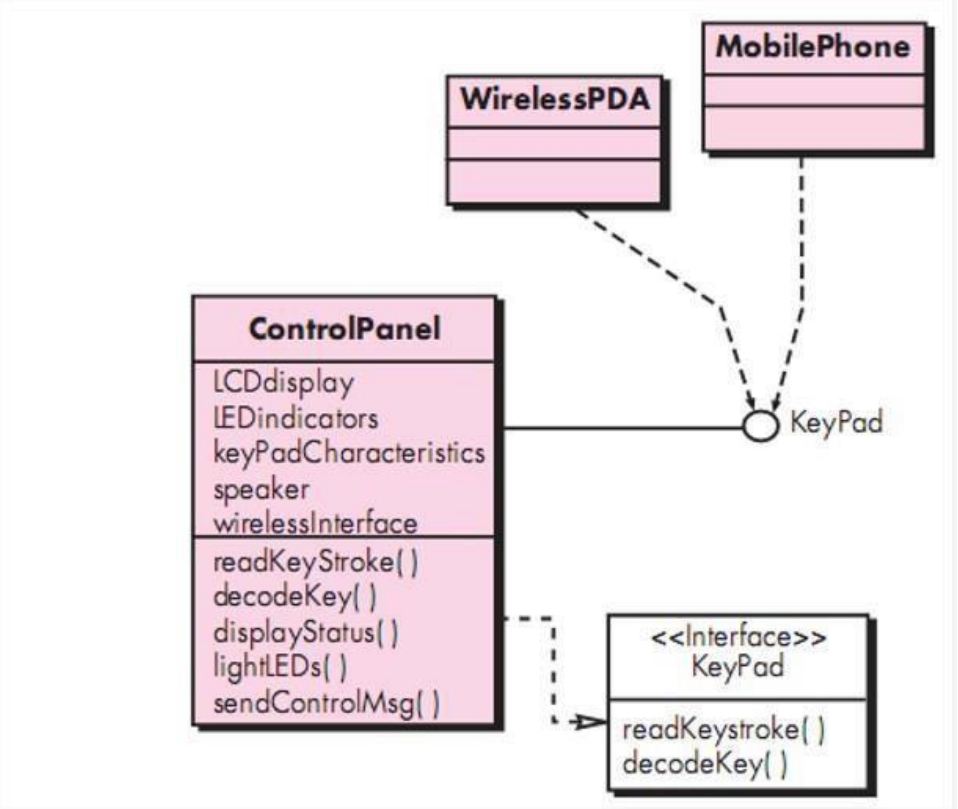
  (3) The availability of architectural styles and patterns

- Difference: An architectural style is a conceptual way of how the system will be created / will work.
- An architectural pattern describes a solution for implementing a style at the level of subsystems or modules and their relationships.

3. **Interface Design Elements**

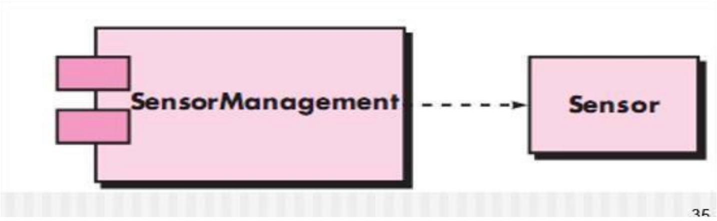
- The interface design elements for software represent information flows into and out of the system and how it is communicated among the components defined as part of the architecture.
- For example : A set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house. These drawings describe the size and shape of doors and windows, the manner in which they operate, the way in which utility connections (e.g., water, electrical, gas, telephone) come into the house and are distributed among the rooms depicted in the floor plan.
- There are three important elements of interface design:
  - (1) The user interface (UI);
  - (2) External interfaces to other systems, devices, networks, or other producers or consumers of information;
  - (3) Internal interfaces between various design components.
- **UI design** (increasingly called usability design) is a major software engineering action
- Usability design incorporates
  - Visual elements (e.g., layout, color, graphics, interaction mechanisms),
  - Ergonomic elements (e.g., information layout and placement, metaphors, UI navigation),

- Technical elements (e.g., UI patterns, reusable components).
- In general, the UI is a unique subsystem within the overall application architecture.



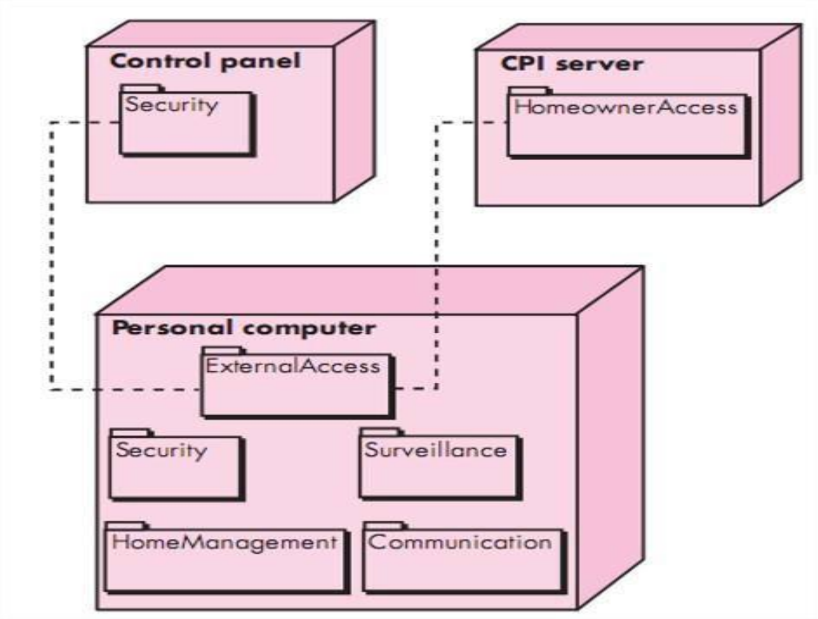
4. **<u>Component-Level Design Elements</u>**

- The component-level design for software fully describes the internal detail of each software component.
- Component elements (detailed drawing of each room, wiring, place of switches…)
  - Internal details of each software component
    - Data structures,
    - algorithmic details,
    - interface to access component operation (behavior).



5. **<u>Deployment-Level Design Elements</u>**

- Deployment-level design elements indicate how software functionality and subsystems will be allocated within the physical computing environment that will support the software.
- For example, the elements of the SafeHome product are configured to operate within three primary computing environments
  - A home-based PC,
  - The SafeHome control panel,
  - Server housed at CPI Corp. (providing Internet-based access to the system).



<u>Software architecture:</u>

The architecture of a system describes its major components, their relationships (structures), and how they interact with each other. Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.

Architecture serves as a **blueprint for a system**. It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.

- It defines a **structured solution** to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.
- Further, it involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product. These decisions comprise of −

➢ Selection of structural elements and their interfaces by which the system is composed.
➢ Behavior as specified in collaborations among those elements.
➢ Composition of these structural and behavioral elements into large subsystem.
➢ Architectural decisions align with business objectives.

- Architectural styles guide the organization.

Data design

Here data design is described at both the architectural and component levels. At the architecture level, data design is the process of creating a model of the information represented at a high level of abstraction (using the customer's view of data)

1.Data Design at the Architectural Level

- The challenge is extract useful information from the data environment, particularly when the information desired is cross-functional.
- To solve this challenge, the business IT community has developed data mining techniques, also called knowledge discovery in databases (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information
- However, the existence of multiple databases, their different structures, and the degree of detail contained with the databases, and many other factors make data mining difficult within an existing database environment
- An alternative solution, called a data warehouse, adds on additional layer to the data architecture
- A data warehouse is a separate data environment that is not directly integrated with day-to-day applications that encompasses all data used by a business

2.Data Design at the Component Level

At the component level, data design focuses on specific data structures required to realize the data objects to be manipulated by a component.

- Refine data objects and develop a set of data abstractions
- Implement data object attributes as one or more data structures
- Review data structures to ensure that appropriate relationships have been established Set of principles for data specification:

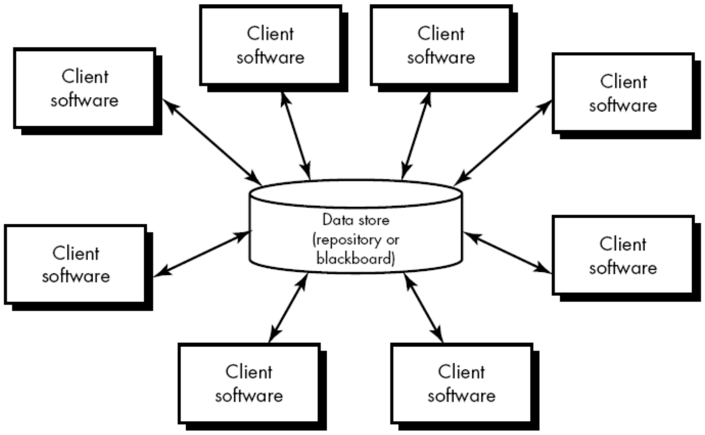
1. The systematic analysis principles applied to function and behavior should also be applied to data
2. All data structures and the operations to be performed on each should be identified
3. A data dictionary should be established and used to define both data and program design
4. Low level data design decisions should be deferred until late in the design process
5. The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure
6. A library of useful data structures and the operations that may be applied to them should be developed
7. A software design and programming language should support the specification and realization of abstract data types

Architectural styles and patterns

Architectural styles:

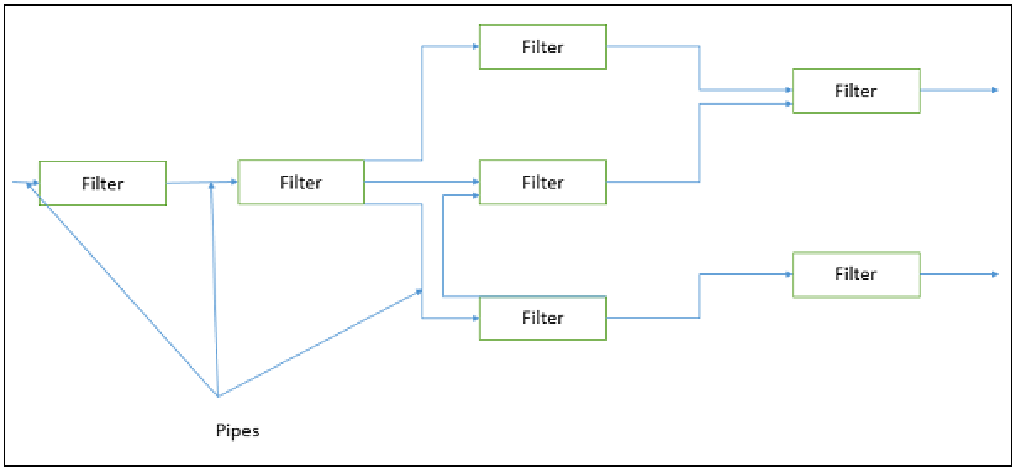
1. Data-centered architecture

- The data store in the file or database is occupying at the center of the architecture.
- Store data is access continuously by the other components like an update, delete, add, modify from the data store.
- Data-centered architecture helps integrity.
- Pass data between clients using the blackboard mechanism.
- The processes are independently executed by the client components.



2. Data-flow architecture

- This architecture is applied when the input data is converted into a series of manipulative components into output data.
- A pipe and filter pattern is a set of components called as filters.
- Filters are connected through pipes and transfer data from one component to the next component.
- The flow of data degenerates into a single line of transform then it is known as batch sequential.



3. Call and return architectures

This architecture style allows to achieve a program structure which is easy to modify.
Following are the sub styles exist in this category:

a)Main program or subprogram architecture

The program is divided into smaller pieces hierarchically.

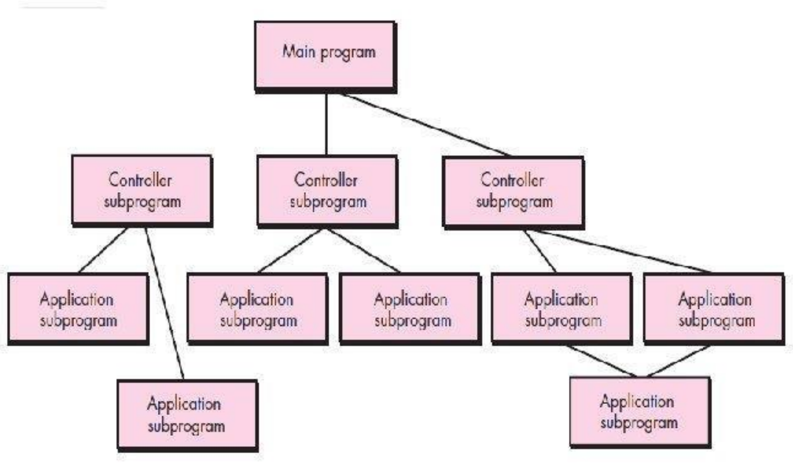
The main program invokes many of program components in the hierarchy that program components are divided into subprogram.

b)Remote procedure call architecture

The main program or subprogram components are distributed in network of multiple computers.

The main aim is to increase the performance.

Call and Return Architecture



63

4. Object-oriented architectures

- This architecture is the latest version of call-and-return architecture.
- It consist of the bundling of data and methods.

5. Layered architectures

- The different layers are defined in the architecture. It consists of outer and inner layer.
- The components of outer layer manage the user interface operations.
- Components execute the operating system interfacing at the inner layer.
- The inner layers are application layer, utility layer and the core layer.
- In many cases, It is possible that more than one pattern is suitable and the alternate architectural style can be designed and evaluated.

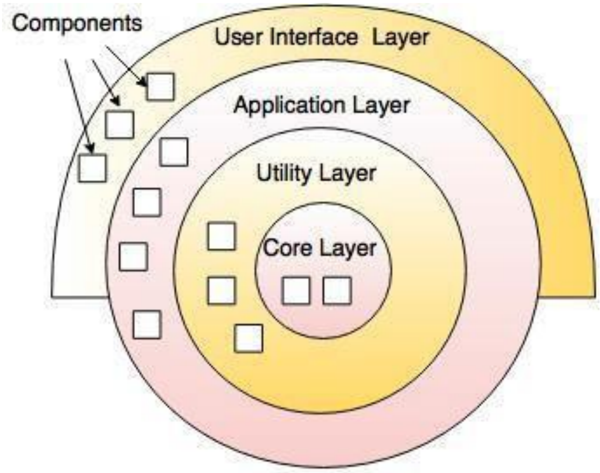


Fig.- Layered Architecture

Architectural patterns:

Different Software Architecture Patterns :

1. Layered Pattern
2. Client-Server Pattern
3. Event-Driven Pattern
4. Microkernel Pattern
5. Microservices Pattern

Let's see one by one in detail.

1.LayeredPattern :

As the name suggests, components(code) in this pattern are separated into layers of subtasks and they are arranged one above another.
Each layer has unique tasks to do and all the layers are independent of one another. Since each layer is independent, one can modify the code inside a layer without affecting others.

It is the most commonly used pattern for designing the majority of software. This layer is also known as 'N-tier architecture'. Basically, this pattern has 4 layers.

1. Presentation layer (The user interface layer where we see and enter data into an application.)
2. Business layer (this layer is responsible for executing business logic as per the request.)
3. Application layer (this layer acts as a medium for communication between the 'presentation layer' and 'data layer'.
4. Data layer (this layer has a database for managing data.)

Ideal for:

E-commerce web applications development like Amazon.

2.Client-ServerPattern :

The client-server pattern has two major entities. They are a server and multiple clients.
Here the server has resources(data, files or services) and a client requests the server for a particular resource. Then the server processes the request and responds back accordingly.

Examples of software developed in this pattern:

- Email.
- WWW.
- File sharing apps.
- Banking, etc…

So this pattern is suitable for developing the kind of software listed in the examples.

3.Event-DrivenPattern :
Event-Driven Architecture is an agile approach in which services (operations) of the software are triggered by events.
Well, what does an event mean?

When a user takes action in the application built using the EDA approach, a state change happens and a reaction is generated that is called an event.

Eg: A new user fills the signup form and clicks the signup button on Facebook and then a FB account is created for him, which is an event.
Ideal for:

Building websites with JavaScript and e-commerce websites in general.

4.MicrokernelPattern :
Microkernel pattern has two major components. They are a core system and plug-in modules.

- The core system handles the fundamental and minimal operations of the application.
- The plug-in modules handle the extended functionalities (like extra features) and customized processing.

5.MicroservicesPattern :
The collection of small services that are combined to form the actual application is the concept of microservices pattern. Instead of building a bigger application, small programs are built for every service (function) of an application independently. And those small programs are bundled together to be a full-fledged application.
So adding new features and modifying existing microservices without affecting other microservices are no longer a challenge when an application is built in a microservices pattern.

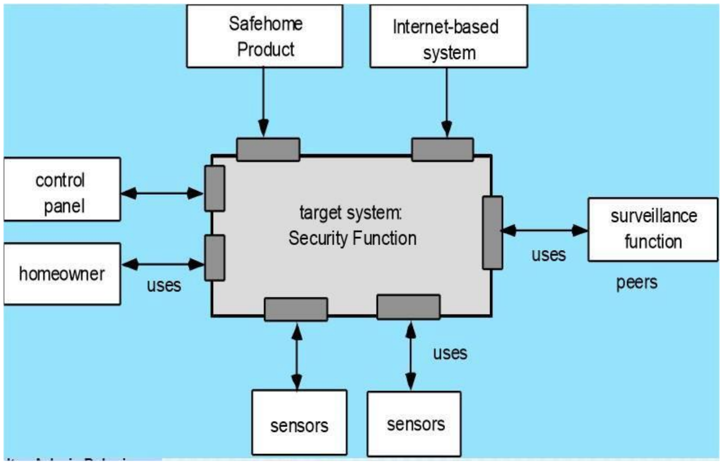
Modules in the application of microservices patterns are loosely coupled. So they are easily understandable, modifiable and scalable.

Architectural design

- **As architectural design begins**, the software to be developed must be put into context
- That is, the design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction.

Representing the System in Context

- At the architectural design level, a software architect uses an **architectural context diagram (ACD)** to model the manner in which software interacts with entities external to its boundaries.
- The generic structure of the architectural context diagram is illustrated in Figure.



In figure, systems that interoperate with the target system (the system for which an architectural design is to be developed) are represented as

- **Super ordinate systems : those** systems that use the target system as part of some higher-level processing scheme.
- **Subordinate systems—those** systems that are used by the target system and provide data or processing that are necessary to complete target system functionality.
- **Peer-level systems—those** systems that interact on a peer-to- peer basis (i.e., information is either produced or consumed by the peers and the target system.
- **Actors—entities** *(people, devices)* that interact with the target system by producing or consuming information.
- Each of these external entities communicates with the target system through an interface (the small shaded rectangles).

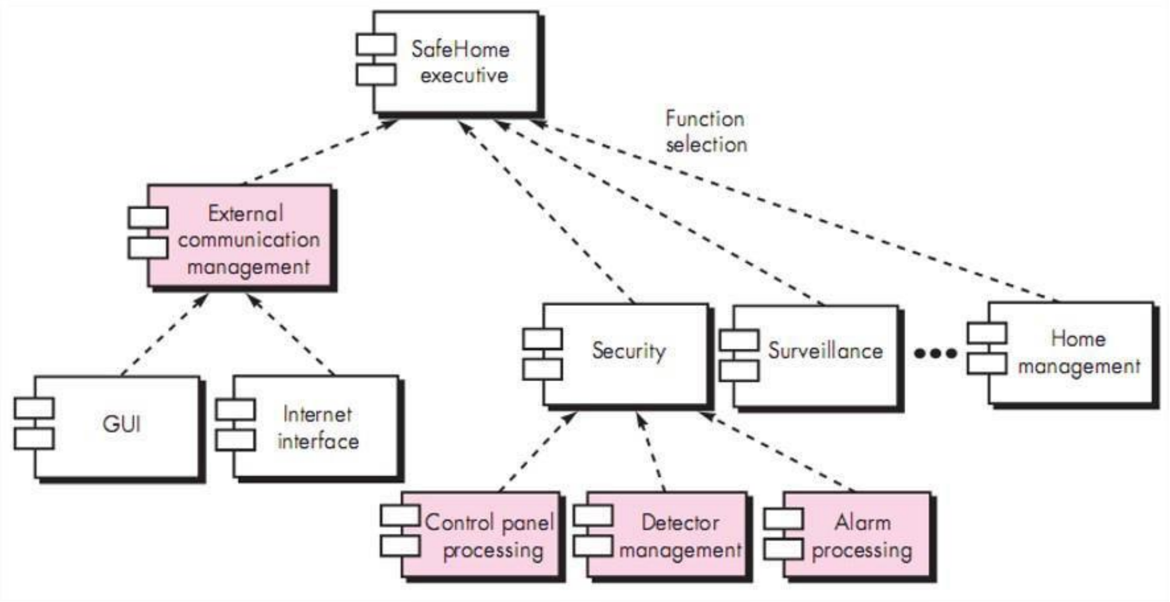
Defining Archetypes

- *An archetype is a class or pattern that represents a core abstraction that is critical to the design of an architecture for the target system.*
- In general, a relatively small set of archetypes is required to design even relatively complex systems.
- *Archetypes are the abstract building blocks of an architectural design.*
- In many cases, archetypes can be derived by examining the analysis classes defined as part of the requirements model.
- An archetype is a generic, idealized model of a person, object, or concept from which similar instances are derived, copied, patterned, or emulated.
- The SafeHome home security function, you might define the following archetypes :
    - **Node** : Represents a cohesive collection of input and output elements of the home security function.
    - For example a node might be included of (1) various sensors and (2) a variety of alarm (output) indicators.
    - **Detector** : An abstraction that covers all sensing equipment that feeds information into the target system.
    - **Indicator**. An abstraction that represents all mechanisms (e.g., alarm siren, flashing lights, bell) for indicating that an alarm condition is occurring.

- **Controller**. An abstraction that describes the mechanism that allows the arming (Supporting) or disarming of a node. If controllers reside on a network, they have the ability to communicate with one another.

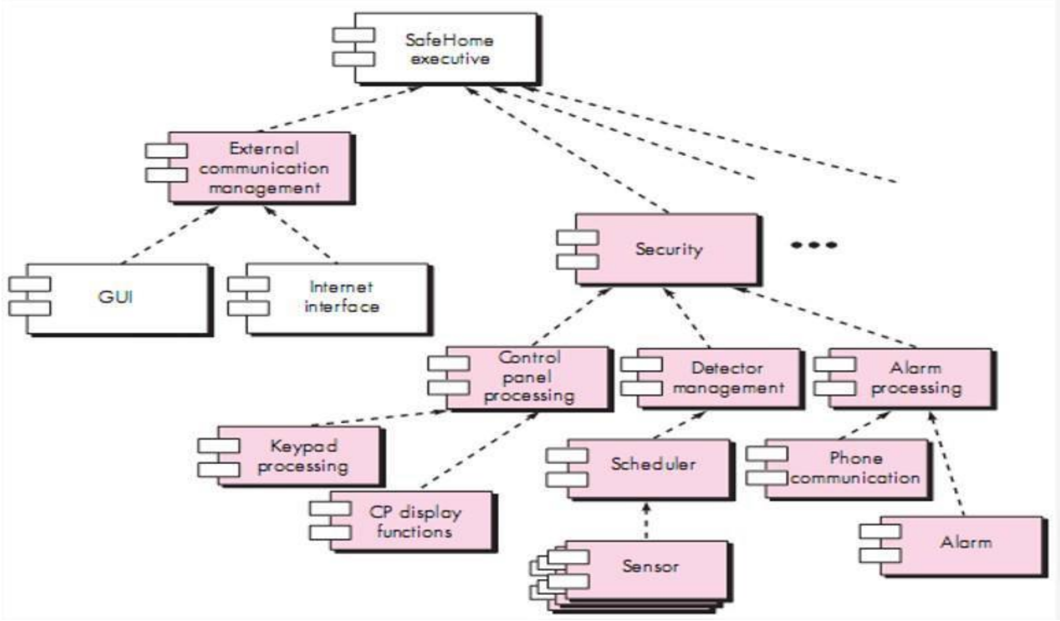
Refining the Architecture into Components

- As the software architecture is refined into components.
- Analysis classes represent entities within the application (business) domain that must be addressed within the software architecture.
- In some cases (e.g., a graphical user interface), a complete subsystem architecture with many components must be designed.
- For Example : The SafeHome home security function example, you might define the set of top-level components that address the following functionality:
- **External communication management — coordinates** communication of the security function with external entities such as other Internet-based systems and external alarm notification.
- **Control panel processing—** manages all control panel functionality.
- **Detector management** — coordinates access to all detectors attached to the system.
- **Alarm processing** — verifies and acts on all alarm conditions
- The overall architectural structure (represented as a **UML component diagram**) is in the following Figure.



Describing Instantiations of the System

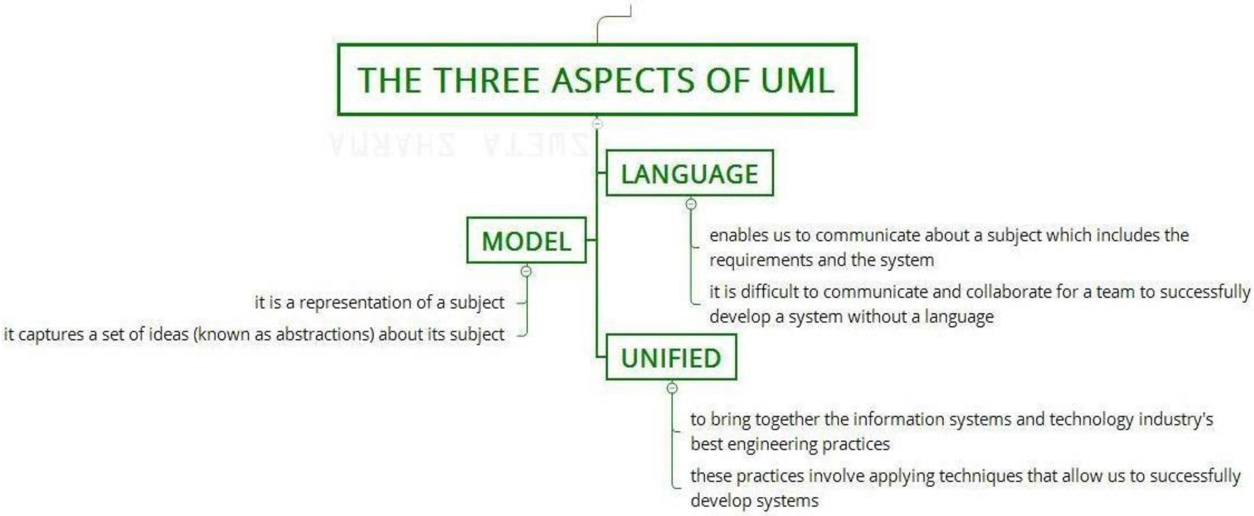
- The architectural design that has been modeled to this point is still relatively high level.
- The context of the system has been represented
- Archetypes that indicate the important abstractions within the problem domain have been defined,
- The overall structure of the system is apparent, and the major software components have been identified.
- However, further refinement is still necessary.
- To accomplish this, an actual instantiation of the architecture is developed.It means, again it simplify by more details.
- The figure demonstrates this concept.



Conceptual model of UML

The Unified Modeling Language (UML) is a standard visual language for describing and modelling software blueprints. The UML is more than just a graphical language. Stated formally, the UML is for: Visualizing, Specifying, Constructing, and Documenting.
The artifacts of a software-intensive system (particularly systems built using the object-oriented style).

Three Aspects of UML:



enables us to communicate about a subject which includes the requirements and the system

it is difficult to communicate and collaborate for a team to successfully develop a system without a language

it is a representation of a subject

it captures a set of ideas (known as abstractions) about its subject

to bring together the information systems and technology industry's best engineering practices

these practices involve applying techniques that allow us to successfully develop systems

- **Figure –** Three Aspects of UML
- **Note –** Language, Model, and Unified are the important aspect of UML as described in the map above.

1. Language:

- It enables us to communicate about a subject which includes the requirements and the system.
- It is difficult to communicate and collaborate for a team to successfully develop a system without a language.

2. Model:

- It is a representation of a subject.
- It captures a set of ideas *(known as abstractions)* about its subject.

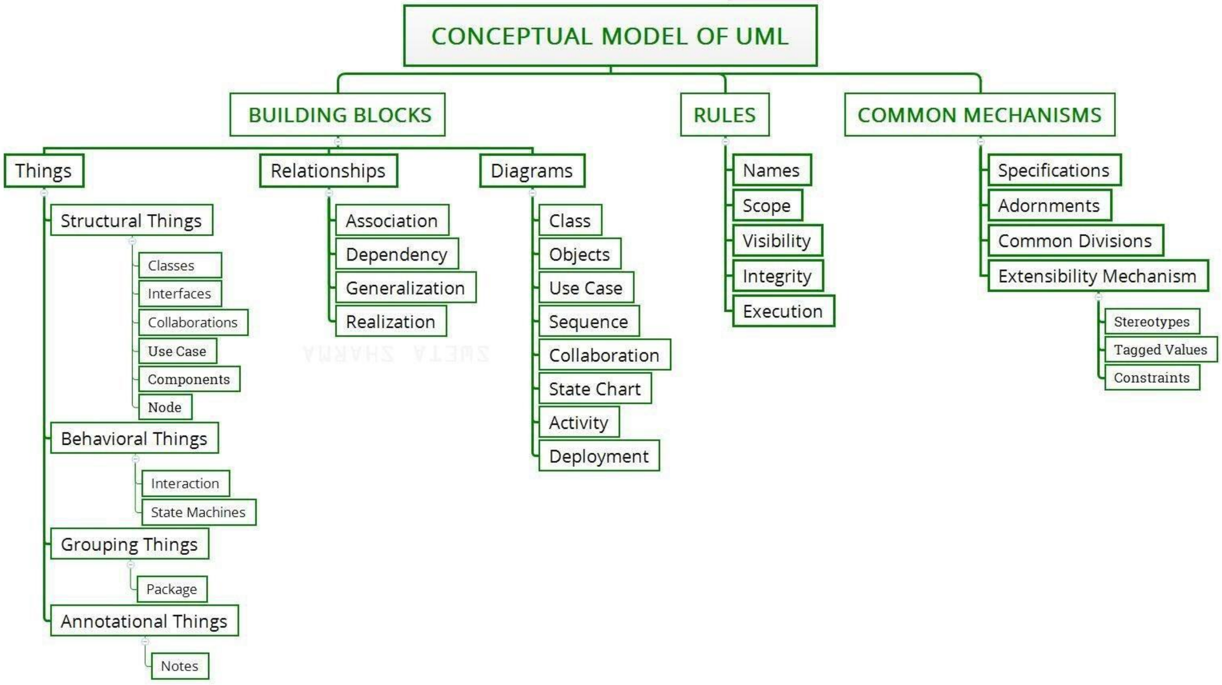
3. Unified:

- It is to bring together the information systems and technology industry's best engineering practices.
- These practices involve applying techniques that allow us to successfully develop systems.

A Conceptual Model:

A conceptual model of the language underlines the three major elements:

- The Building Blocks
- The Rules
- Some Common Mechanisms



<u>BASIC STRUCTURAL MODELING</u>

Contents:

1. Classes
2. Relationships
3. Common Mechanisms
4. Diagrams

1.Classes:

- Names
- Attributes
- Operations

2.Relationships:

- Dependencies
- Generalizations
- Associations
- Aggregation

3.Common Mechanisms:

- Notes
- Other Adornments
- Stereotypes
- Tagged Values
- Constraints

4. Diagrams:

Structural Diagrams

The UML's structural diagrams exist to visualize, specify, construct, and document the static aspects of a system. You can think of the static aspects of a system as representing its relatively stable skeleton and scaffolding. Just as the static aspects of a house encompass the existence and placement of such things as walls, doors, windows, pipes, wires, and vents, so too do the static aspects of a software system encompass the existence and placement of such things as classes, interfaces, collaborations, components, and nodes.

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

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

Behavioral Diagrams

The UML's behavioral diagrams are used to visualize, specify, construct, and document the dynamic aspects of a system. You can think of the dynamic aspects of a system as representing its changing parts. Just as the dynamic aspects of a house encompass airflow and traffic through the rooms of a house, so too do the dynamic aspects of a software system encompass such things as the flow of messages over time and the physical movement of components across a network.

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

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

Class diagram:

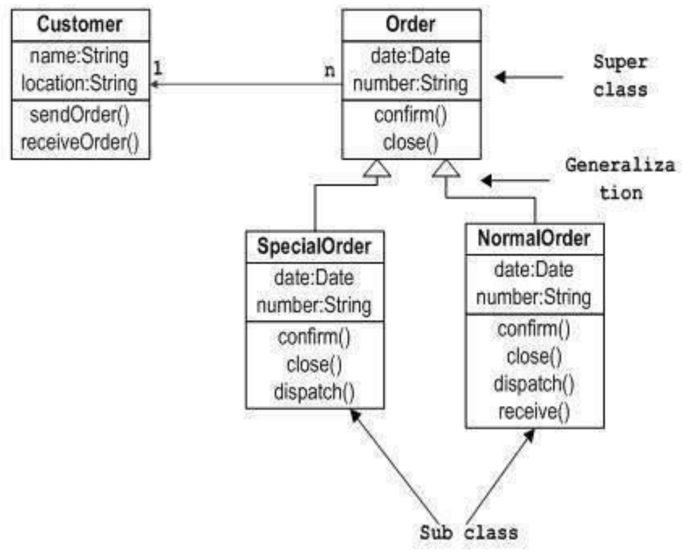
The purpose of class diagram is to model the static view of an application. Class diagrams are the only diagrams which can be directly mapped with object-oriented languages and thus widely used at the time of construction.

UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application, however class diagram is a bit different. It is the most popular UML diagram in the coder community.

The purpose of the class diagram can be summarized as −

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

Sample Class Diagram



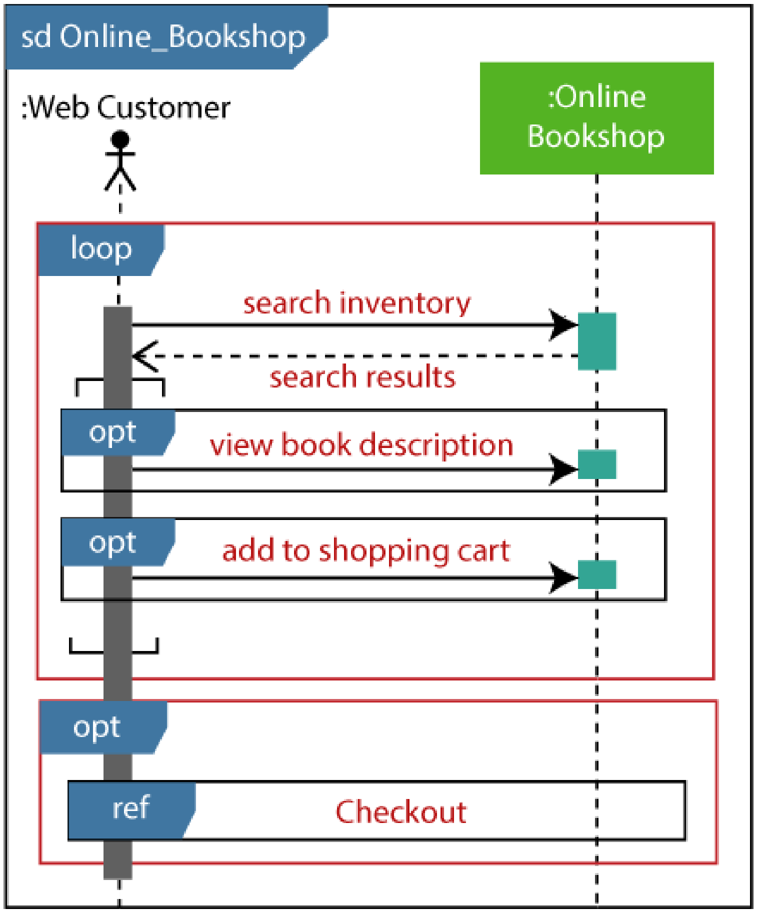
Sequence Diagram:

1. To model high-level interaction among active objects within a system.
2. To model interaction among objects inside a collaboration realizing a use case.
3. It either models generic interactions or some certain instances of interaction.

Example of a Sequence Diagram

An example of a high-level sequence diagram for online bookshop is given below.

Any online customer can search for a book catalog, view a description of a particular book, add a book to its shopping cart, and do checkout.



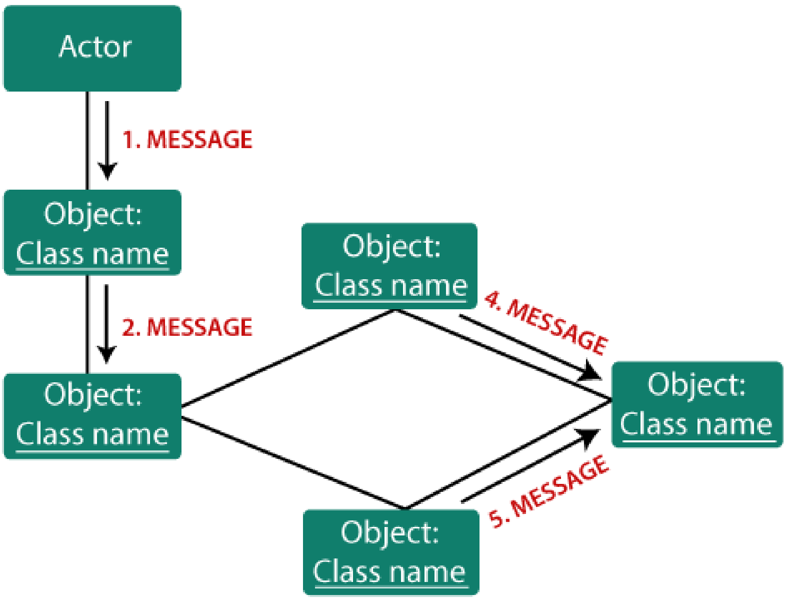
Collaboration diagram:

Notations of a Collaboration Diagram

- Objects: The representation of an object is done by an object symbol with its name and class underlined, separated by a colon.

- Actors: In the collaboration diagram, the actor plays the main role as it invokes the interaction. Each actor has its respective role and name. In this, one actor initiates the use case.
- Links: The link is an instance of association, which associates the objects and actors. It portrays a relationship between the objects through which the messages are sent. It is represented by a solid line. The link helps an object to connect with or navigate to another object, such that the message flows are attached to links.
- Message: It is a communication between objects which carries information and includes a sequence number, so that the activity may take place. It is represented by a labeled arrow, which is placed near a link. The messages are sent from the sender to the receiver, and the direction must be navigable in that particular direction. The receiver must understand the message.

Components of a collaboration diagram



Use Case Diagrams

The purpose of use case diagram is to capture the dynamic aspect of a system. However, this definition is too generic to describe the purpose, as other four diagrams (activity, sequence, collaboration, and Statechart) also have the same purpose. We will look into some specific purpose, which will distinguish it from other four diagrams.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. Hence, when a system is analyzed to gather its functionalities, use cases are prepared and actors are identified.

When the initial task is complete, use case diagrams are modelled to present the outside view.

In brief, the purposes of use case diagrams can be said to be as follows −

- Used to gather the requirements of a system.

- Used to get an outside view of a system.
- Identify the external and internal factors influencing the system.
- Show the interaction among the requirements are actors.

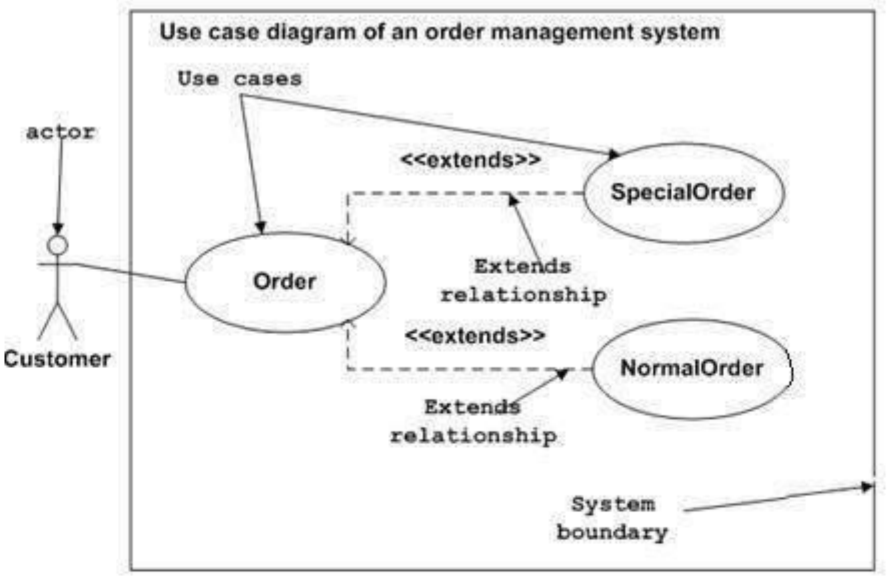


Figure: Sample Use Case diagram

Component Diagrams:

Component diagram is a special kind of diagram in UML. The purpose is also different from all other diagrams discussed so far. It does not describe the functionality of the system but it describes the components used to make those functionalities.

Thus from that point of view, component diagrams are used to visualize the physical components in a system. These components are libraries, packages, files, etc.

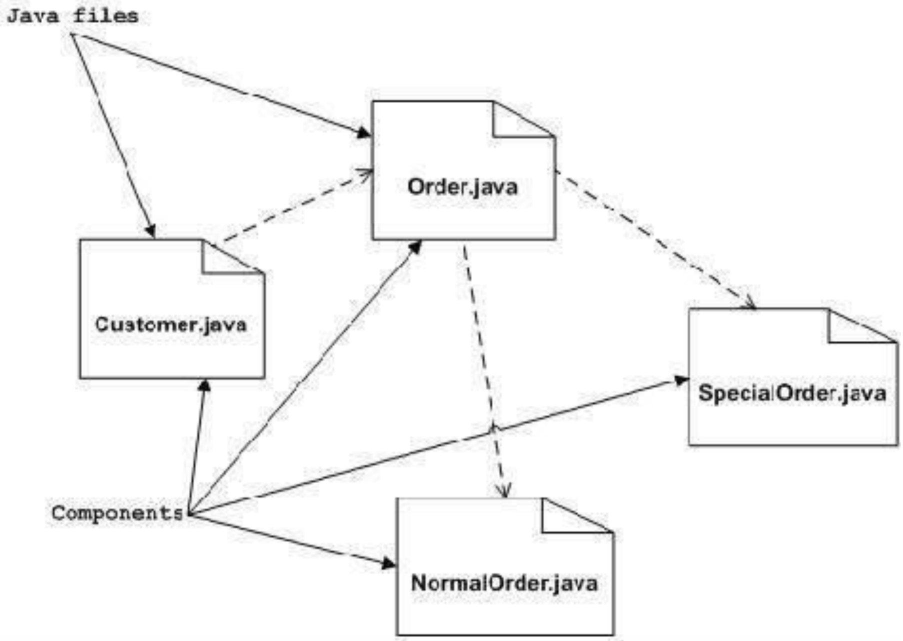
Component diagrams can also be described as a static implementation view of a system. Static implementation represents the organization of the components at a particular moment.

A single component diagram cannot represent the entire system but a collection of diagrams is used to represent the whole.

The purpose of the component diagram can be summarized as −

- Visualize the components of a system.
- Construct executables by using forward and reverse engineering.
- Describe the organization and relationships of the components.

Component diagram of an order management system



UNIT-4

TESTING STRATEGIES

Strategic Approach to software testing:

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which we can place specific test case design techniques and testing methods—should be defined for the software process.

A number of software testing strategies have been proposed in the literature. All provide the software developer with a template for testing and all have the following generic characteristics:
• Testing begins at the component level2 and works "outward" toward the integration of the entire computer-based system.
• Different testing techniques are appropriate at different points in time.
• Testing is conducted by the developer of the software and (for large projects) an independent test group.
• Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

- **Verification and Validation**

  Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). Verification refers to the set of activities that ensure that software correctly implements a specific function. Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Boehm states this another way:

  Verification:"Are we building the product right?"
  Validation: "Are we building the right product?"

- **Organizing for Software Testing**

  For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who have built the software are now asked to test the software. This seems harmless in itself; after all, who knows the program better than its developers? Unfortunately, these same developers have a vested interest in demonstrating that the program is error free, that it works according to customer requirements, and that it will be completed on schedule and within budget. Each of these interests mitigate against thorough testing.

There are often a number of misconceptions that can be erroneously inferred from the preceeding discussion: **(1)** that the developer of software should do no testing at all, **(2)** that the software should be "tossed over the wall" to strangers who will test it mercilessly, **(3**) that testers get involved with the project only when the testing steps are about to begin. Each of these statements is incorrect.

Test strategies for Conventional Software

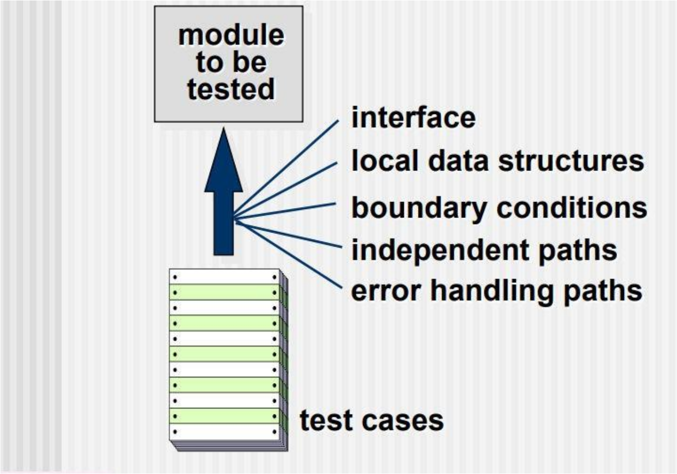
- There are many strategies that can be used to test software.
- At one extreme, you can wait until the system is fully constructed and then conduct tests on the overall system in hopes of finding errors.
- This approach simply does not work. It will result in buggy software.
- At the other extreme, you could conduct tests on a daily basis, whenever any part of the system is constructed.
- This approach, although less appealing to many, can be very effective.
- A testing strategy that is chosen by most software teams falls between the two extremes**.**
- It takes an incremental view of testing,
- Beginning with the testing of individual program units,
- Moving to tests designed to facilitate the integration of the units,
- Culminating with tests that exercise the constructed system.

Unit Test :

Unit testing focuses verification effort on the smallest unit of software design—the software component or module.

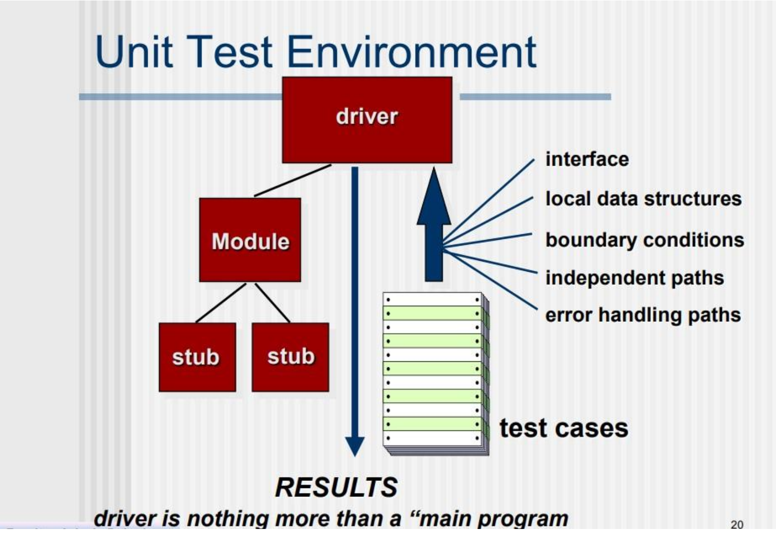
The unit test focuses on the internal processing logic and data structures within the boundaries of a component.

This type of testing can be conducted in parallel for multiple components.



- Unit tests are illustrated schematically in previous Figure.
- **The module interface** is tested to ensure that information properly flows into and out of the program unit under test.
- **Local data structures** are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.

- **Boundary conditions** are tested to ensure that the module operates properly at boundaries established to limit or restrict processing.
- **Finally, all error-handling paths are tested**
- Good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur.



Unit Test Environment

driver

Module

stub

stub

interface

local data structures

boundary conditions

independent paths

error handling paths

test cases

RESULTS

driver is nothing more than a "main program

20

Integration Testing:

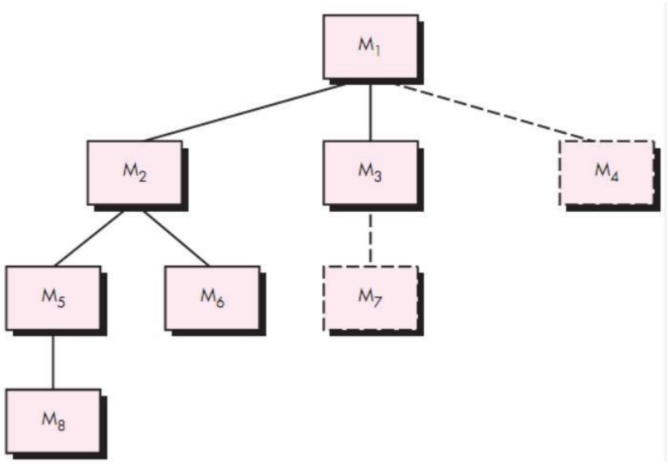
Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.

Different Integration Testing Strategies :

- Top-down testing
- Bottom-up testing
- Regression Testing
- Smoke Testing

Top-down testing

- Top-down integration testing is an incremental approach to construction of the software architecture.
- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).
- Modules subordinate (and ultimately subordinate) to **the main control module are incorporated into the structure in either a *depth-first or breadth-first manner.***



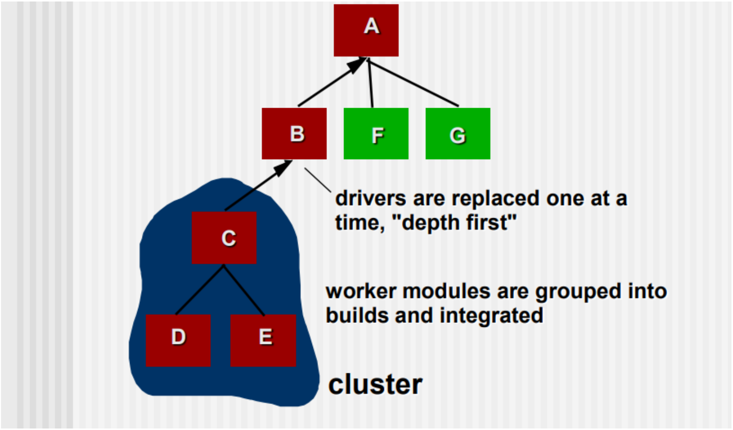
BOTTOM-UP INTEGRATION TESTING:

Bottom-up integration testing, It begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).

Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps…

1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.


A
B
F
G
C
D
E
drivers are replaced one at a time, "depth first"
worker modules are grouped into builds and integrated
cluster

REGRESSION TESTING:

Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.

Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.

Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

It is impractical and inefficient to reexecute every test for every program function once a change has occurred….

Regression testing is a type of software testing that seeks to uncover new software bugs, OR

Regression testing is the process of testing, changes to computer programs to make sure that the older programming still works with the new changes. Here changes such as enhancements, patches or configuration changes, have been made to them.

SMOKE TESTING:

Smoke testing is an integration testing approach that is commonly used when product software is developed

Smoke testing is performed by developers before releasing the build to the testing team and after releasing the build to the testing team it is performed by testers whether to accept the build for further testing or not.

It is designed as a pacing (Speedy) mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.

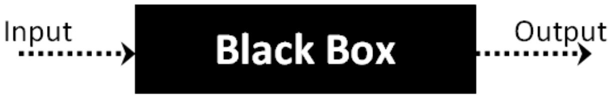
Smoke testing provides a number of benefits when it is applied on complex, time critical software projects.

- I**ntegration risk is minimized.** Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early.
- **The quality of the end product is improved**. Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as architectural and component-level design errors. If these errors are corrected early, better product quality will result.
- **Error diagnosis and correction are simplified.**
- **Progress is easier to assess**

Black – box and white – box testing

Black – box testing:

Black Box Testing is a software testing method in which the functionalities of software applications are tested without having knowledge of internal code structure, implementation details and internal paths. Black Box Testing mainly focuses on input and output of software applications and it is entirely based on software requirements and specifications. It is also known as Behavioral Testing.



- **Types of Black Box Testing**

There are many types of Black Box Testing but the following are the prominent ones –

- **Functional testing** – This black box testing type is related to the functional requirements of a system; it is done by software testers.
- **Non-functional testing** – This type of black box testing is not related to testing of specific functionality, but non-functional requirements such as performance, scalability, usability.
- **Regression testing** – Regression Testing is done after code fixes, upgrades or any other system maintenance to check the new code has not affected the existing code.

Black Box Testing Techniques

Following are the prominent Test Strategy amongst the many used in Black box Testing

- **Equivalence Class Testing:** It is used to minimize the number of possible test cases to an optimum level while maintains reasonable test coverage.
- **Boundary Value Testing:** Boundary value testing is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not. It is very useful in reducing the number of test cases. It is most suitable for the systems where an input is within certain ranges.
- **Decision Table Testing**: A decision table puts causes and their effects in a matrix. There is a unique combination in each column.

White – box testing:

The box testing approach of software testing consists of black box testing and white box testing. We are discussing here white box testing which also known as glass box is **testing, structural testing, clear box testing, open box testing and transparent box testing**.

It tests internal coding and infrastructure of a software focus on checking of predefined inputs against expected and desired outputs. It is based on inner workings of an application and revolves around internal structure testing.

In this type of testing programming skills are required to design test cases. The primary goal of white box testing is to focus on the flow of inputs and outputs through the software and strengthening the security of the software.

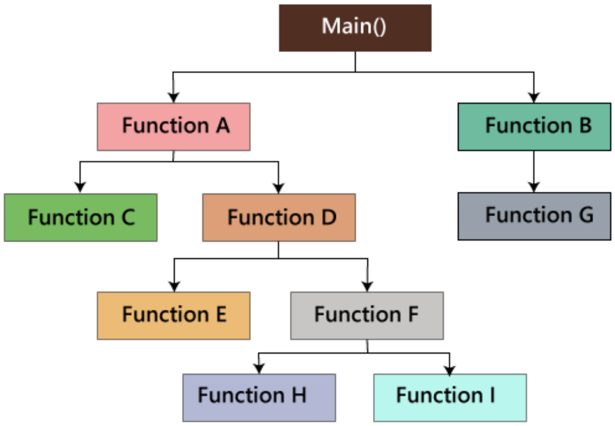
The white box testing contains various tests, which are as follows:

- Path testing

- Loop testing
- Condition testing
- Testing based on the memory perspective
- Test performance of the program

Path testing---

In the path testing, we will write the flow graphs and test all independent paths. Here writing the flow graph implies that flow graphs are representing the flow of the program and also show how every program is added with one another as we can see in the below image:



Loop testing---

In the loop testing, we will test the loops such as while, for, and do-while, etc. and also check for ending condition if working correctly and if the size of the conditions is enough.

Condition testing---

In this, we will test all logical conditions for both **true** and **false** values; that is, we will verify for both **if** and **else** condition.

Testing based on the memory (size) perspective---

The size of the code is increasing for the following reasons:

- **The reuse of code is not there**: let us take one example, where we have four programs of the same application, and the first ten lines of the program are similar. We can write these ten lines as a discrete function, and it should be accessible by the above four programs as well. And also, if any bug is there, we can modify the line of code in the function rather than the entire code.

- The **developers use the logic** that might be modified. If one programmer writes code and the file size is up to 250kb, then another programmer could write a similar code using the different logic, and the file size is up to 100kb.
- The **developer declares so many functions and variables** that might never be used in any portion of the code. Therefore, the size of the program will increase.

Test the performance (Speed, response time) of the program---

The application could be slow for the following reasons:

- When logic is used.
- For the conditional cases, we will use **or** & **and** adequately.
- Switch case, which means we cannot use **nested if**, instead of using a switch case.

Differences between Black Box Testing vs White Box Testing:

Black Box Testing	White Box Testing
➢ It is a way of software testing in which the internal structure or the program or the code is hidden and nothing is known about it.	➢ It is a way of testing the software in which the tester has knowledge about the internal structure or the code or the program of the software.
➢ Implementation of code is not needed for black box testing.	➢ Code implementation is necessary for white box testing.
➢ It is mostly done by software testers.	➢ It is mostly done by software developers.
➢ No knowledge of implementation is needed.	➢ Knowledge of implementation is required.
➢ It can be referred as outer or external software testing.	➢ It is the inner or the internal software testing.
➢ It is functional test of the software.	➢ It is structural test of the software.
➢ This testing can be initiated on the basis of requirement specifications document.	➢ This type of testing of software is started after detail design document.
➢ No knowledge of programming is required.	➢ It is mandatory to have knowledge of programming.
➢ It is the behavior testing of the	➢ It is the logic testing of the software.

Black Box Testing	**White Box Testing**
software.	
It is applicable to the higher levels of testing of software.	It is generally applicable to the lower levels of software testing.
It is also called closed testing.	It is also called as clear box testing.
It is least time consuming.	It is most time consuming.
It is not suitable or preferred for algorithm testing.	It is suitable for algorithm testing.
Can be done by trial and error ways and methods.	Data domains along with inner or internal boundaries can be better tested.
Example: search something on google by using keywords	**Example:** by input to check and verify loops
Types of Black Box Testing:	**Types of White Box Testing:**
A. Functional Testing	A. Path Testing
B. Non-functional testing	B. Loop Testing
C. Regression Testing	C. Condition testing

Difference between Alpha and Beta Testing:

Alpha Testing	**Beta Testing**
Alpha testing involves both the white box and black box testing.	Beta testing commonly uses black-box testing.
Alpha testing is performed by testers who are usually internal employees of the organization.	Beta testing is performed by clients who are not part of the organization.
Alpha testing is performed at the developer's site.	Beta testing is performed at the end-user of the product.
Reliability and security testing	Reliability, security and robustness are

Alpha Testing	Beta Testing
are not checked in alpha testing.	checked during beta testing.
➢ Alpha testing ensures the quality of the product before forwarding to beta testing.	➢ Beta testing also concentrates on the quality of the product but collects users input on the product and ensures that the product is ready for real time users.
➢ Alpha testing requires a testing environment or a lab.	➢ Beta testing doesn't require a testing environment or lab.
➢ Alpha testing may require a long execution cycle.	➢ Beta testing requires only a few weeks of execution.
➢ Developers can immediately address the critical issues or fixes in alpha testing.	➢ Most of the issues or feedback collected from the beta testing will be implemented in future versions of the product.
➢ Multiple test cycles are organized in alpha testing.	➢ Only one or two test cycles are there in beta testing.

Validation Testing

The definition of validation testing in software engineering is in place to determine if the existing system complies with the system requirements and performs the dedicated functions for which it is designed along with meeting the goals and needs of the organization.

This mode of testing is extremely important especially if you want to be ***one of the best software testers***. The software verification and validation testing is the process after the validation testing stage is secondary to verification testing.

The Advantages of Validation Testing :

- *To ensure customer satisfaction*
- *To be confident about the product*
- *To fulfill the client's requirement until the optimum capacity*
- *Software acceptance from the end-user*

Types of Validation Testing

Validation testing types a V-shaped testing pattern, which includes its variations and all the activities that it consists of are:

Unit Testing – It is an important type of validation testing. The point of the unit testing is to search for bugs in the product segment. Simultaneously, it additionally confirms crafted modules and articles which can be tried independently.

Integration testing -This is a significant piece of the validation model wherein the interaction between, where the association between the various interfaces of the pertaining component is tried. Alongside the communication between the various pieces of the framework, the connection of the framework with the PC working framework, document framework, equipment, and some other programming framework it may cooperate with, is likewise tried.

System testing – System testing is done when the whole programming framework is prepared. The principal worry of framework testing is to confirm the framework against the predefined necessities. While doing the tests, the tester isn't worried about the internals of the framework however checks if the framework acts according to desires.

User acceptance testing – During this testing, the tester actually needs to think like the customer and test the product concerning client needs, prerequisites, and business forms and decide if the product can be given over to the customer or not.

System Testing

System Testing includes testing of a fully integrated software system. Generally, a computer system is made with the integration of software (any software is only a single element of a computer system).

The software is developed in units and then interfaced with other software and hardware to create a complete computer system. In other words, a computer system consists of a group of software to perform the various tasks, but only software cannot perform the task; for that software must be interfaced with compatible hardware.

System testing is a series of different type of tests with the purpose to exercise and examine the full working of an integrated software computer system against requirements.

Types of system test:

- Recovery testing
- Security testing
- Stress testing
- Performance testing

Recovery testing:

It is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.

If recovery is automatic, reinitialization, check pointing, mechanisms, data recovery, and restart and evaluated for correctness.

Security testing:

Verifies the protection mechanisms built into a system will.

Stress testing:

It executes a system in a manner that demands resources in abnormal quality, frequency, or volume.

Performance testing:

It designed to test the run – time performance of software with in the context of an integrated system.

The art of debugging

In the context of software engineering, debugging is the process of fixing a bug in the software. In other words, it refers to identifying, analyzing, and removing errors. This activity begins after the software fails to execute properly and concludes by solving the problem and successfully testing the software. It is considered to be an extremely complex and tedious task because errors need to be resolved at all stages of debugging.

The debugging process will always have one of two outcomes :

1. The cause will be found and corrected.

2. The cause will not be found.

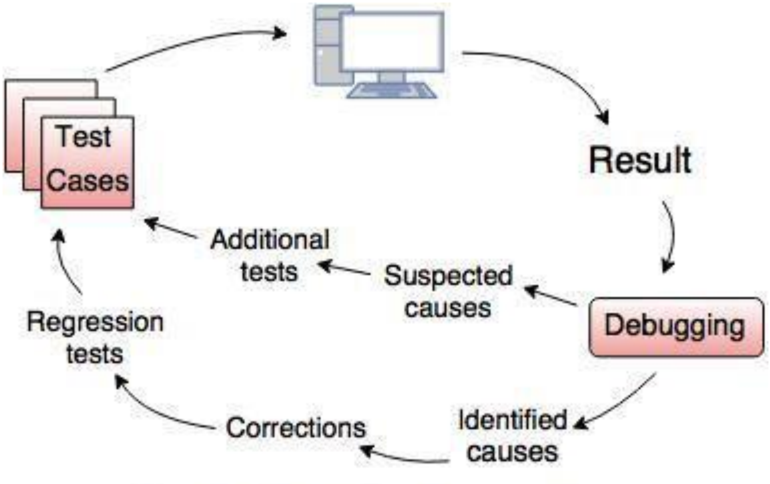


Fig. - Debugging process

Debugging Approaches/Strategies:

1. **Brute Force:** Study the system for a larger duration in order to understand the system. It helps the debugger to construct different representations of systems to be debugging depending on the need. A study of the system is also done actively to find recent changes made to the software.
2. **Backtracking:** Backward analysis of the problem which involves tracing the program backward from the location of the failure message in order to identify the region of faulty code. A detailed study of the region is conducted to find the cause of defects.
3. **Forward analysis** of the program involves tracing the program forwards using breakpoints or print statements at different points in the program and studying the results. The region where the wrong outputs are obtained is the region that needs to be focused on to find the defect.
4. **Using the past experience** of the software debug the software with similar problems in nature. The success of this approach depends on the expertise of the debugger.
5. **Cause elimination:** it introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes.

4.2. Product metrics

Software quality

Software quality is conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

McCall's quality Factors

According to McCall's model, product operation category includes five software quality factors, which deal with the requirements that directly affect the daily operation of the software. They are as follows –

Correctness:

These requirements deal with the correctness of the output of the software system. They include –

- Output mission
- The required accuracy of output that can be negatively affected by inaccurate data or inaccurate calculations.
- The completeness of the output information, which can be affected by incomplete data.

Reliability:

Reliability requirements deal with service failure. They determine the maximum allowed failure rate of the software system, and can refer to the entire system or to one or more of its separate functions.

Efficiency: It deals with the hardware resources needed to perform the different functions of the software system.

Integrity: This factor deals with the software system security, that is, to prevent access to unauthorized persons, also to distinguish between the group of people to be given read as well as write permit.

Usability: Usability requirements deal with the staff resources needed to train a new employee and to operate the software system.

Maintainability: This factor considers the efforts that will be needed by users and maintenance personnel to identify the reasons for software failures, to correct the failures, and to verify the success of the corrections.

Flexibility: This factor deals with the capabilities and efforts required to support adaptive maintenance activities of the software.

Testability: Testability requirements deal with the testing of the software system as well as with its operation.

Portability: Portability requirements tend to the adaptation of a software system to other environments consisting of different hardware, different operating systems, and so forth.

Reusability: This factor deals with the use of software modules originally designed for one project in a new software project currently being developed.

Interoperability: Interoperability requirements focus on creating interfaces with other software systems or with other equipment firmware.

ISO 9126 QUALITY FACTORS:

1. **Functionality:** The functions are those that will satisfy implied needs.
   - Suitability
   - Accuracy
   - Interoperability
   - Security
   - Functionality Compliance
2. **Reliability:** A set of attributes that will bear on the capability of software to maintain the level of performance.
   - Maturity
   - Fault Tolerance
   - Recoverability
   - Reliability Compliance
3. **Usability:** A set of attributes that bear on the effort needed for use by a implied set of users.
   - Understandability
   - Learn ability
   - Operability
   - Attractiveness
   - Usability Compliance
4. **Efficiency:** A set of attributes that bear on the relationship between the level of performance of the software under stated conditions.
   - Time Behavior
   - Resource Utilization
   - Efficiency Compliance
5. **Maintainability:** A set of attributes that bear on the effort needed to make specified modifications.
   - Analyzability
   - Changeability
   - Stability
   - Testability
   - Maintainability Compliance
6. **Portability:** A set of attributes that bear on the ability of software to be transferred from one environment to another.
   - Adaptability
   - Installability

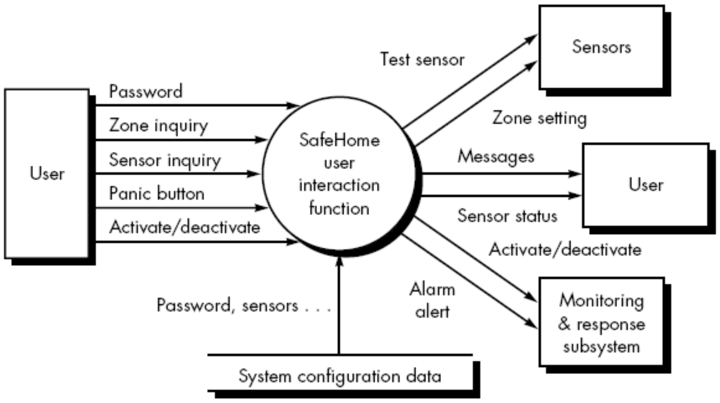
- Co-existence
- Replace ability
- Portability Compliance

Metrics for Analysis model

Technical work in software engineering begins with the creation of the analysis model. It is at this stage that requirements are derived and that a foundation for design is established. Therefore, technical metrics that provide insight into the quality of the analysis model are desirable.

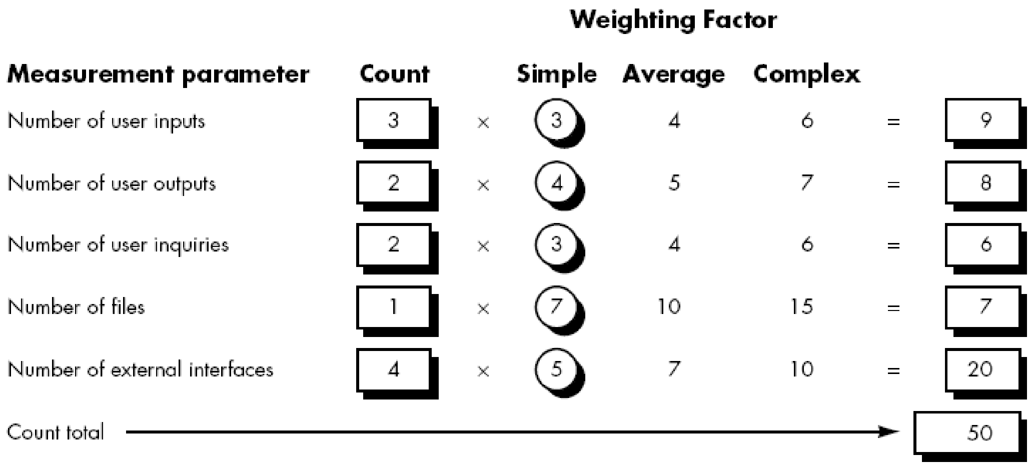
Function-Based Metrics:

The function point metric an be used effectively as a means for predicting the size of a system that will be derived from the analysis model.



The data flow diagram is evaluated to determine the key measures required for computation of the function point metric :

- number of user inputs
- number of user outputs
- number of user inquiries
- number of files
- number of external interfaces



			Weighting Factor				
Measurement parameter	Count		Simple	Average	Complex		
Number of user inputs	3	×	3	4	6	=	9
Number of user outputs	2	×	4	5	7	=	8
Number of user inquiries	2	×	3	4	6	=	6
Number of files	1	×	7	10	15	=	7
Number of external interfaces	4	×	5	7	10	=	20
Count total						→	50

The count total **FP = count total [0.65 + 0.01 (Fi)]**

where count total is the sum of all FP entries obtained from the first figure and Fi (i = 1 to 14) are "complexity adjustment values."

Metrics for Specification Quality

Davis and his colleagues propose a list of characteristics that can be used to assess the quality of the analysis model and the corresponding requirements specification: specificity (lack of ambiguity), completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision, and reusability.

- we assume that there are nr requirements in a specification, such that

nr = nf + nnf

where nf is the number of functional requirements and nnf is the number of nonfunctional (e.g., performance) requirements.

- To determine the specificity (lack of ambiguity) of requirements, Davis et al. suggest a metric that is based on the consistency of the reviewers' interpretation of each requirement:

Q1 = nui/nr

where nui is the number of requirements for which all reviewers had identical interpretations. The closer the value of Q to 1, the lower is the ambiguity of the specification.

- The completeness of functional requirements can be determined by computing the ratio

Q2 = nu/[ni x ns]

where nu is the number of unique function requirements, ni is the number of inputs (stimuli) defined or implied by the specification, and ns is the number of states specified. The Q2 ratio measures the percentage of necessary functions that have been specified for a system.

- **Q3 = nc/[nc + nnv]**

where nc is the number of requirements that have been validated as correct and nnv is the number of requirements that have not yet been validate

Metrics for design model

Design metrics for computer software, like all other software metrics, are not perfect. Debate continues over their efficacy and the manner in which they should be applied. Many experts argue that further experimentation is required before design measures can be used. And yet, design without measurement is an unacceptable alternative .

1.Architectural Design Metrics

Architectural design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules. These metrics are black box in the sense that they do not require any knowledge of the inner workings of a particular software component.

Card and Glass define three software design complexity measures: structural complexity, data complexity, and system complexity.

Structural complexity of a module i is defined in the following manner:

$$S(i) = f 2 out(i)$$

where fout(i) is the fan-out7 of module i.

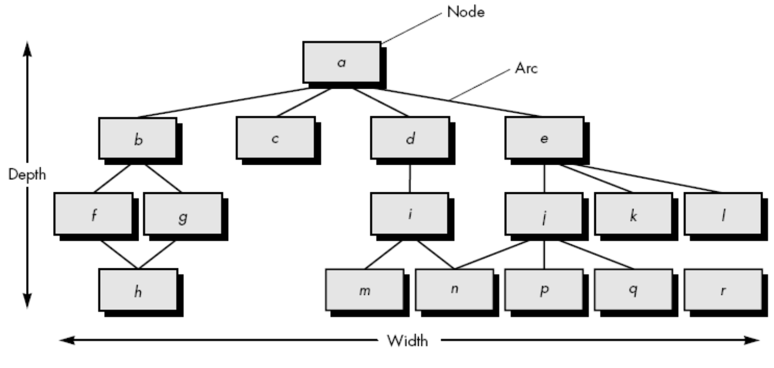
Data complexity provides an indication of the complexity in the internal interface for a module i and is defined as

$$D(i) = v(i)/[fout(i) +1]$$

where v(i) is the number of input and output variables that are passed to and from module i.

Finally, system complexity is defined as the sum of structural and data complexity, specified as

$$C(i) = S(i) + D(i)$$



size = n + a
where n is the number of nodes and a is the number of arcs. For the architecture shown in figure,

size = 17 + 18 = 35

depth = the longest path from the root (top) node to a leaf node. For the architecture shown infigure, depth = 4.
width = maximum number of nodes at any one level of the architecture. For the architecture shown in figure, width = 6.
arc-to-node ratio, r = a/n,

the Air Force uses information obtained from data and architectural design to derive a design structure quality index (DSQI) that ranges from 0 to 1. The following values must be ascertained to compute the DSQI :

S1 = the total number of modules defined in the program architecture.
S2 = the number of modules whose correct function depends on the source of data input or that produce data to be used elsewhere (in general, control modules, among others, would not be counted as part of S2).
S3 = the number of modules whose correct function depends on prior processing.
S4 = the number of database items (includes data objects and all attributes that define objects).
S5 = the total number of unique database items.
S6 = the number of database segments (different records or individual objects).
S7 = the number of modules with a single entry and exit (exception processing is not considered to be a multiple exit).

Once values S1 through S7 are determined for a computer program, the following intermediate values can be computed:

Program structure: D1, where D1 is defined as follows: If the architectural design was developed using a distinct method (e.g., data flow-oriented design or object-oriented design), then D1 = 1, otherwise D1 = 0.
Module independence: D2 = 1 (S2/S1)
Modules not dependent on prior processing: D3 = 1 (S3/S1)
Database size: D4 = 1 (S5/S4)
Database compartmentalization: D5 = 1 (S6/S4)
Module entrance/exit characteristic: D6 = 1 (S7/S1)
With these intermediate values determined, the DSQI is computed in the following manner:

DSQI = wiDi
where i = 1 to 6, wi is the relative weighting of the importance of each of the intermediate values, and wi = 1 (if all Di are weighted equally, then wi = 0.167).

2. Metrics for object – oriented design

- Size
- Complexity
- Coupling
- Sufficiency
- Completeness

- Cohesion
- Primitiveness
- Similarity
- Volatility

Metrics for source code

HSS(Halstead Software science)

Primitive measure that may be derived after the code is generated or estimated once design is

Complete.

n1 = the number of distinct operators that appear in a program

n2 = the number of distinct operands that appear in a program

N1 = the total number of operator occurrences.

N2 = the total number of operand occurrence.

Overall program length N can be computed:

$N = n1 \log2 n1 + n2 \log2 n2$

$V = N \log2(n1+ n2)$

V will vary with programming language and represent the volume of information required to specify a program.

Halstead defines a volume ratio L as the ratio of volume of the most compact from of a program to the volume of the actual program. In actuality, L must be less than 1.

In terms of primitive measures, the volume ratio may be expressed as

$L=2/n1 * n2/N2$

METRIC FOR TESTING

- Halstead metrics applied to testing:

n1 = the number of distinct operators that appear in a program

n2 = the number of distinct operands that appear in a program

N1 = the total number of operator occurrences.

N2 = the total number of operand occurrence.

Program Level and Effort

$PL = 1/[(n1 / 2) x (N2 / n2 l)]$

$e = V/PL$

- Metrics for object oriented testing
- Lack of cohesion in method(LCOM)
- Percent public and protected(PAP)
- public access to data members(PAD)
- Number of root classes(NOR)
- Fan-in (FIN)

METRICS FOR MAINTENANCE

Mt = the number of modules in the current release

Fc = the number of modules in the current release that have been changed

Fa = the number of modules in the current release that have been added.

Fd = the number of modules from the preceding release that were deleted in the current release

The Software Maturity Index, SMI, is defined as:

$SMI = [Mt–(Fc + Fa +Fd)/ Mt]$

UNIT-V

METRICS FOR PROCESS

Software measurement:

It can be categorized in two ways:

- Direct measures of the software process – cost and effort applied
- Indirect measures of the product – include functionality, quality, complexity, efficiency, reliability, maintainability.

Size oriented metrics:

LOC Metrics

It is one of the earliest and simpler metrics for calculating the size of the computer program. It is generally used in calculating and comparing the productivity of programmers. These metrics are derived by normalizing the quality and productivity measures by considering the size of the product as a metric.

Following are the points regarding LOC measures:

1. In size-oriented metrics, LOC is considered to be the normalization value.
2. It is an older method that was developed when FORTRAN and COBOL programming were very popular.
3. Productivity is defined as KLOC / EFFORT, where effort is measured in person-months.
4. Size-oriented metrics depend on the programming language used.
5. As productivity depends on KLOC, so assembly language code will have more productivity.
6. LOC measure requires a level of detail which may not be practically achievable.
7. The more expressive is the programming language, the lower is the productivity.
8. LOC method of measurement does not apply to projects that deal with visual (GUI-based) programming. As already explained, Graphical User Interfaces (GUIs) use forms basically. LOC metric is not applicable here.
9. It requires that all organizations must use the same method for counting LOC. This is so because some organizations use only executable statements, some useful comments, and some do not. Thus, the standard needs to be established.
10. These metrics are not universally accepted.

Based on the LOC/KLOC count of software, many other metrics can be computed:

a. Errors/KLOC.

b. $/ KLOC.

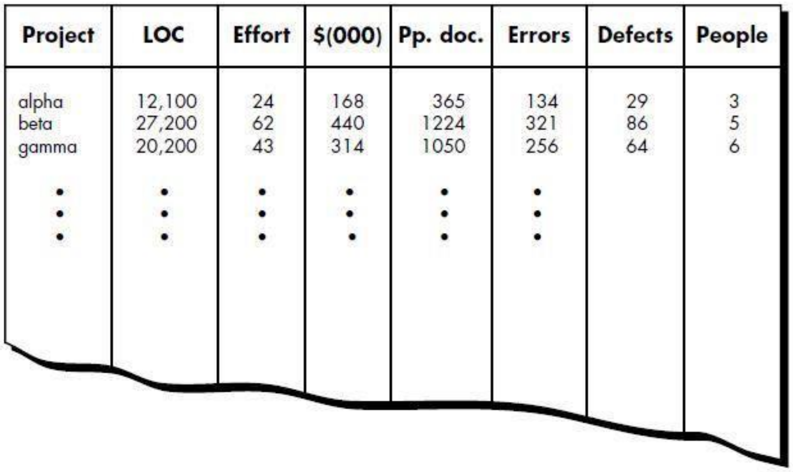
c. Defects/KLOC.

d. Pages of documentation/KLOC.

e. Errors/PM.

f. Productivity = KLOC/PM (effort is measured in person-months).

g. $/ Page of documentation.



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
⋮	⋮	⋮	⋮	⋮	⋮		

Function – oriented metrics:

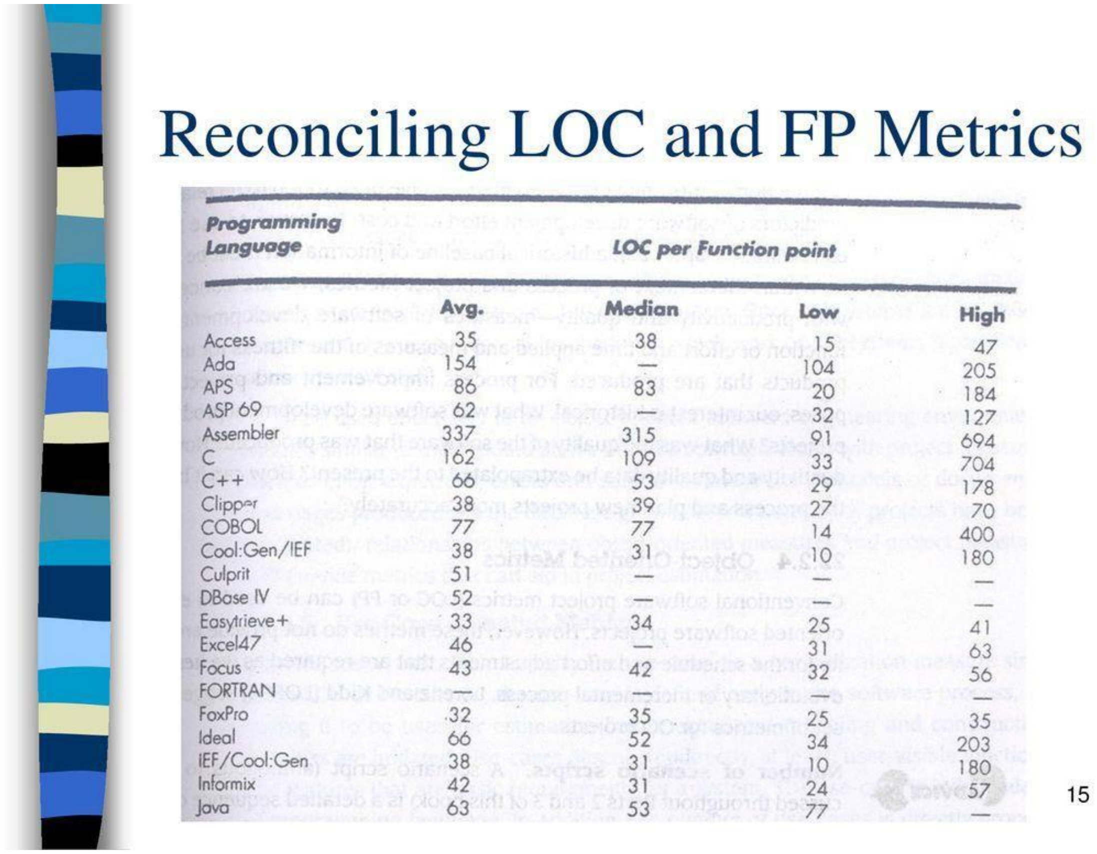
Function-Oriented Metrics are also known as **Function Point Model**. This model generally focuses on the functionality of the software application being delivered.

These methods are actually independent of the programming language that is being used in software applications and based on calculating the Function Point (FP). A function point is a unit of measurement that measures the business functionality provided by the business product.

Calculating Function Point :

Function Point (FP)= Count total * [0.65 + (0.01 * Sum(F_i))]

Reconciling LOC and FP metrics:



Reconciling LOC and FP Metrics

Programming Language	LOC per Function point			
	Avg.	Median	Low	High
Access	35	38	15	47
Ada	154	—	104	205
APS	86	83	20	184
ASP 69	62	—	32	127
Assembler	337	315	91	694
C	162	109	33	704
C++	66	53	29	178
Clipper	38	39	27	70
COBOL	77	77	14	400
Cool:Gen/IEF	38	31	10	180
Culprit	51	—	—	—
DBase IV	52	—	—	—
Easytrieve+	33	34	25	41
Excel47	46	—	31	63
Focus	43	42	32	56
FORTRAN	—	—	—	—
FoxPro	32	35	25	35
Ideal	66	52	34	203
IEF/Cool:Gen	38	31	10	180
Informix	42	31	24	57
Java	63	53	77	—

15

Object – oriented metrics:

Lines of code and functional point metrics can be used for estimating object-oriented software projects.

object-oriented projects, different sets of metrics have been proposed. These are listed below.

- **Number of scenario scripts:** Scenario scripts are a sequence of steps, which depict the interaction between the user and the application. A number of scenarios is directly related to application size and number of test cases that are developed to test the software, once it is developed. Note that scenario scripts are analogous to use-cases.
- **Number of key classes:** Key classes are independent components, which are defined in object -oriented analysis. As key classes form the core of the problem domain, they indicate the effort required to develop software and the amount of 'reuse' feature to be applied during the development process.
- **Number of support classes:** Classes, which are required to implement the system but are indirectly related to the problem domain, are known as support classes. For example, user interface classes and computation class are support classes. It is possible to develop a support class for each key class. Like key classes, support classes indicate the effort required to develop software and the amount of 'reuse' feature to be applied during the development process.

- **Average number of support classes per key class:** Key classes are defined early in the software project while support classes are defined throughout the project. The estimation process is simplified if the average number of support classes per key class is already known.
- **Number of subsystems:** A collection of classes that supports a function visible to the user is known as a subsystem. Identifying subsystems makes it easier to prepare a reasonable schedule in which work on subsystems is divided among project members.

Web engineering project metrics:

- Number of static web pages
- Number of static web pages
- Number of internal page links
- Number of persistent data objects
- Number of external systems interfaced
- Number of static content objects
- Number of dynamic content objects
- Number of executable functions

Metrics for software quality:

Software Measurement is done based on some Software Metrics where these software metrics are referred to as the measure of various characteristics of a Software.

In Software engineering Software Quality Assurance (SAQ) assures the quality of the software. Set of activities in SAQ are continuously applied throughout the software process. Software Quality is measured based on some software quality

1.Code Quality – Code quality metrics measure the quality of code used for the software project development. Maintaining the software code quality by writing Bug-free and semantically correct code is very important for good software project development.

2. Reliability – Reliability metrics express the reliability of software in different conditions. The software is able to provide exact service at the right time or not is checked. Reliability can be checked using Mean Time Between Failure (MTBF) and Mean Time To Repair (MTTR).

3. Performance – Performance metrics are used to measure the performance of the software. Each software has been developed for some specific purposes. Performance metrics measure the performance of the software by determining whether the software is fulfilling the user requirements or not, by analyzing how much time and resource it is utilizing for providing the service.

4. Usability – Usability metrics check whether the program is user-friendly or not. Each software is used by the end-user. So it is important to measure that the end-user is happy or not by using this software.

5. Correctness – Correctness is one of the important software quality metrics as this checks whether the system or software is working correctly without any error by satisfying the user. Correctness gives the degree of service each function provides as per developed.

6. Maintainability – Each software product requires maintenance and up-gradation. Maintenance is an expensive and time-consuming process. So if the software product provides easy maintainability then we can say software quality is up to mark. Maintainability metrics include time requires to adapt to new features/functionality, Mean Time to Change (MTTC), performance in changing environments, etc.

7. Integrity – Software integrity is important in terms of how much it is easy to integrate with other required software's which increases software functionality and what is the control on integration from unauthorized software's which increases the chances of cyber attacks.

8. Security – Security metrics measure how much secure the software is? In the age of cyber terrorism, security is the most essential part of every software. Security assures that there are no unauthorized changes, no fear of cyber attacks, etc when the software product is in use by the end-user.

RISK MANAGEMENT

A software project can be concerned with a large variety of risks. In order to be adept to systematically identify the significant risks which might affect a software project, it is essential to classify risks into different classes. The project manager can then check which risks from each class are relevant to the project.

There are three main classifications of risks which can affect a software project:

1. Project risks
2. Technical risks
3. Business risks

1. Project risks: Project risks concern differ forms of budgetary, schedule, personnel, resource, and customer-related problems. A vital project risk is schedule slippage. Since the software is intangible, it is very tough to monitor and control a software project. It is very tough to control something which cannot be identified. For any manufacturing program, such as the manufacturing of cars, the plan executive can recognize the product taking shape.

2. Technical risks: Technical risks concern potential method, implementation, interfacing, testing, and maintenance issue. It also consists of an ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks appear due to the development team's insufficient knowledge about the project.

3. Business risks: This type of risks contain risks of building an excellent product that no one need, losing budgetary or personnel commitments, etc.

Reactive vs proactive risk strategies:

Reactive risk management:

One fundamental point about reactive risk management is that the disaster or threat must occur before management responds. Proactive risk management is all about taking preventative measures before the event to decrease its severity, and that's a good thing to do.

At the same time, however, organizations should develop reactive risk management plans that can be deployed ***after*** the event. Otherwise management is making decisions about how to respond as the event happens, which can be a costly and stressful ordeal.

Proactive Risk Management

As the name suggests, proactive risk management means that you identify risks before they happen and figure out ways to avoid or alleviate the risk. It seeks to reduce the hazard's risk potential or, even better, prevent the threat altogether.

A good example here is vulnerability testing and remediation. Any organization of appreciable size is likely to have vulnerabilities in its software, which attackers could find an exploit. So regular testing (or, even better, continuous testing) can help to repair those vulnerabilities and eliminate that particular threat.

Software risks

A software project can be concerned with a large variety of risks. In order to be adept to systematically identify the significant risks which might affect a software project, it is essential to classify risks into different classes. The project manager can then check which risks from each class are relevant to the project.

There are three main classifications of risks which can affect a software project:

1. Project risks
2. Technical risks
3. Business risks

1. Project risks: Project risks concern differ forms of budgetary, schedule, personnel, resource, and customer-related problems. A vital project risk is schedule slippage. Since the software is intangible, it is very tough to monitor and control a software project. It is very tough to control something which cannot be identified. For any manufacturing program, such as the manufacturing of cars, the plan executive can recognize the product taking shape.

2. Technical risks: Technical risks concern potential method, implementation, interfacing, testing, and maintenance issue. It also consists of an ambiguous specification, incomplete specification, changing specification, technical uncertainty, and technical obsolescence. Most technical risks appear due to the development team's insufficient knowledge about the project.

3. Business risks: This type of risks contain risks of building an excellent product that no one need, losing budgetary or personnel commitments, etc.

Other risk categories

1. Known risks: Those risks that can be uncovered after careful assessment of the project program, the business and technical environment in which the plan is being developed, and more reliable data sources (e.g., unrealistic delivery date)

2. Predictable risks: Those risks that are hypothesized from previous project experience (e.g., past turnover)

3. Unpredictable risks: Those risks that can and do occur, but are extremely tough to identify in advance.

Risk identification

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks : generic risks and product-specific risks.

Generic risks are a potential threat to every software project. Product-specific risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the project at hand.

One method for identifying risks is to create a risk item checklist. The checklist can beused for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

• **Product size**—risks associated with the overall size of the software to be built or modified.

• **Business impact—**risks associated with constraints imposed by management or the marketplace.

- **Customer characteristics—**risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.

- **Process definition—**risks associated with the degree to which the software process has been defined and is followed by the development organization.

- **Development environment—**risks associated with the availability and quality of the tools to be used to build the product.

- **Technology to be built—**risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.

Staff size and experience—risks associated with the overall technical and project experience of the software engineers who will do the work.

Assessing Overall Project Risk

The following questions have derived from risk data obtained by surveying experienced software project managers in different part of the world. The questions are ordered by their relative importance to the success of a project.
1. Have top software and customer managers formally committed to support the project?
2. Are end-users enthusiastically committed to the project and the system/product to be built?
3. Are requirements fully understood by the software engineering team and their customers?
4. Have customers been involved fully in the definition of requirements?
5. Do end-users have realistic expectations?
6. Is project scope stable?
7. Does the software engineering team have the right mix of skills?
8. Are project requirements stable?
9. Does the project team have experience with the technology to be implemented?
10. Is the number of people on the project team adequate to do the job?
11. Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

Risk Components and Drivers

software risk components—performance, cost, support, and schedule. In the context of this discussion, the risk components are defined in the following manner:

- **Performance risk**—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.

- **Cost risk**—the degree of uncertainty that the project budget will be maintained.

• **Support risk**—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.

• **Schedule risk**—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

The impact of each risk driver on the risk component is divided into one of four impact categories—negligible, marginal, critical, or catastrophic.

Risk projection

Risk projection, also called risk estimation, attempts to rate each risk in two ways—the likelihood or probability that the risk is real and the consequences of the problems associated with the risk, should it occur. The project planner, along with other managers and technical staff, performs four risk projection activities:

(1) establish a scale 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, and

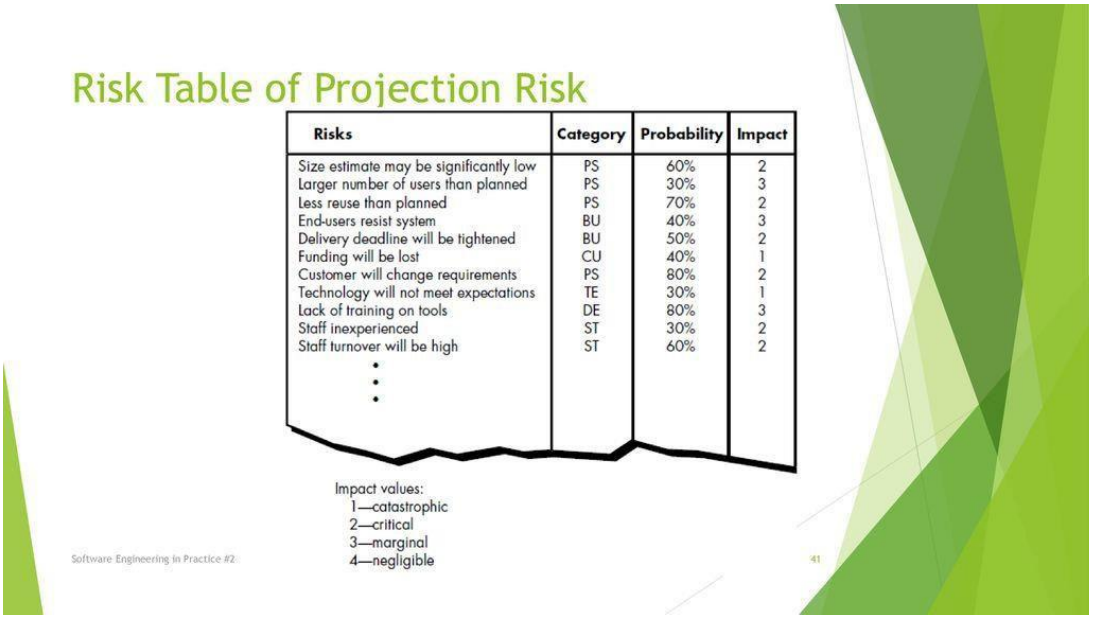
(4)note the overall accuracy of the risk projection so that there will be no misunderstandings.

Developing a Risk Table

A risk table provides a project manager with a simple technique for risk projection .A project team begins by listing all risks (no matter how remote) in the first column of the table. This can be accomplished with the help of the risk item checklists Each risk is categorized in the second column (e.g., PS implies a project size risk, BU implies a business risk).

All risks that lie above the cutoff line must be managed. The column labeled RMMM contains a pointer into a Risk Mitigation, Monitoring and Management Plan or alternatively, a collection of risk information sheets developed for all risks that lie above the cutoff.

Risk probability can be determined by making individual estimates and then developing a single consensus value. Although that approach is workable, more sophisticated techniques for determining risk probability have been developed.



Risk Table of Projection Risk

Risks	Category	Probability	Impact
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

Software Engineering in Practice #2

41

Assessing Risk Impact

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing. The nature of the risk indicates the problems that are likely if it occurs.

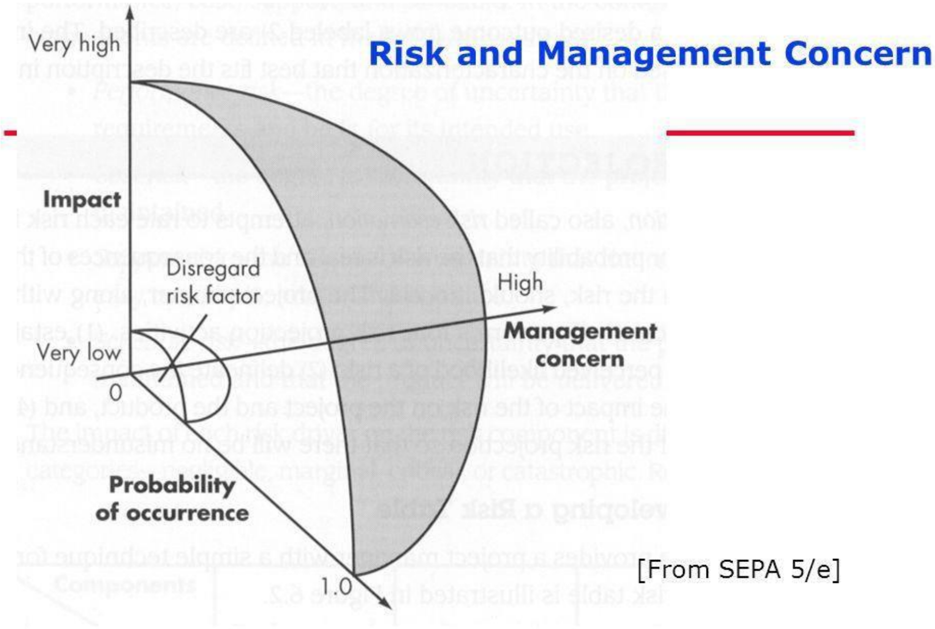
Returning once more to the risk analysis approach proposed by the U.S. Air Force , the following steps are recommended to determine the overall consequences of a risk:
1. Determine the average probability of occurrence value for each risk component.
2. Determine the impact for each component based on the criteria .
3. Complete the risk table and analyze the results as described in the preceding sections.
The overall risk exposure, RE, is determined using the following relationship:

$$RE = P \times C$$

Risk identification. Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Risk probability. 80% (likely).
Risk impact. 60 reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch (in addition to other custom software that has been scheduled for development). Since the average component is 100 LOC and local data indicate that the software engineering cost for each LOC is $14.00, the overall cost (impact) to develop the components would be 18 x 100 x 14 = $25,200.

Risk exposure. RE = 0.80 x 25,200 ~ $20,200.



Risk and Management Concern

[From SEPA 5/e]

RISK REFINEMENT

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage.

One way to do this is to represent the risk in condition-transition-consequence (CTC) format . That is, the risk is stated in the following form: Given that <condition> then there is concern that (possibly) <consequence>.

Using the CTC format for the reuse risk noted in Section 6.4.2, 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.

The consequences associated with these refined subconditions remains the same (i.e., 30 percent of software components must be customer engineered), but the refinement helps to isolate the underlying risks and might lead to easier analysis and response.

RISK MIGRATION, MONITORING, AND MANAGEMENT

Risk Migration:
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.

To mitigate this risk, project management must develop a strategy for reducing turnover. The possible steps to be taken are:

- Meet the current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
- Mitigate those causes that are under our control before the project starts.
- Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
- Organize project teams so that information about each development activity is widely dispersed.
- Define documentation standards and establish mechanisms to ensure that documents are developed in a timely manner.
- Assign a backup staff member for every critical technologist.

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.

As the project proceeds, risk monitoring activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. In the case of high staff turnover, the following factors can be monitored:

- General attitude of team members based on project pressures.

- Interpersonal relationships among team members.
- Potential problems with compensation and benefits.
- The availability of jobs within the company and outside it.

Risk Management and planning :

- **Maintain a worldwide perspective:** view software risks within the context of a system and therefore the business drawback planned to solve.
- **Take an advanced view:** ink regarding the risk which can occur in the longer term and make future plans for managing the future events.
- **Encourage open communication:** Encourage all the stakeholders and users for suggesting risks at any time.
- **Integrate:** A thought of risk should be integrated into the software process.
- **Emphasize never-ending process:** Modify the known risk than a lot of info is understood and add new risks as higher insight is achieved.
- **Develop a shared product vision:** If all the stakeholders share a similar vision of the software then it's easier for better risk identification.
- **Encourage teamwork:** whereas conducting risk management activities pool the skills and knowledge of all stakeholders.

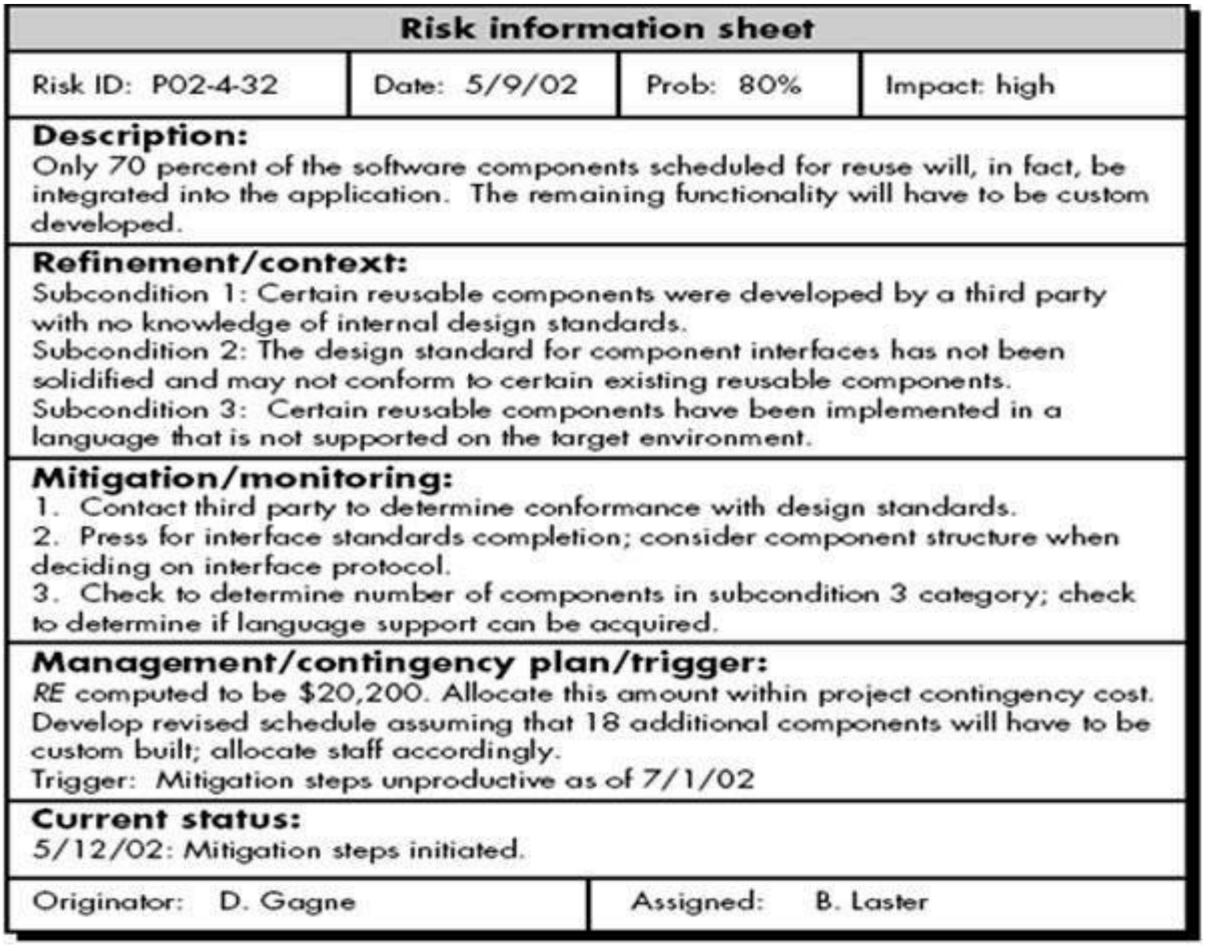
Drawbacks of RMMM:

- It incurs additional project costs.
- It takes additional time.
- For larger projects, implementing an RMMM may itself turn out to be another tedious project.
- RMMM does not guarantee a risk-free project, infact, risks may also come up after the project is delivered.

THE RMMM PLAN

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 information sheet

Risk ID: P02-4-32	Date: 5/9/02	Prob: 80%	Impact: high

Description:
Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.

Refinement/context:
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.

Mitigation/monitoring:
1. Contact third party to determine conformance with design standards.
2. Press for interface standards completion; consider component structure when deciding on interface protocol.
3. Check to determine number of components in subcondition 3 category; check to determine if language support can be acquired.

Management/contingency plan/trigger:
RE computed to be $20,200. Allocate this amount within project contingency cost. Develop revised schedule assuming that 18 additional components will have to be custom built; allocate staff accordingly.
Trigger: Mitigation steps unproductive as of 7/1/02

Current status:
5/12/02: Mitigation steps initiated.

Originator: D. Gagne	Assigned: B. Laster

QUALITY MANAGEMENT

QUALITY CONCEPTS:

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. For software products, the fitness of use is generally explained in terms of satisfaction of the requirements laid down in the SRS document. Although "fitness of purpose" is a satisfactory interpretation of quality for many devices such as a car, a table fan, a grinding machine, etc.for software products, "fitness of purpose" is not a wholly satisfactory definition of quality.

Example: Consider a functionally correct software product. That is, it performs all tasks as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally right, we cannot consider it to be a quality product.

User satisfaction= compliant product + good quality + delivery within budget and schedule

Quality control

It can be compared to having a senior manager walk into a production department and pick a random car for an examination and test drive. Testing activities, in this case, refer to the process of checking every joint, every mechanism separately, as well as the whole product, whether manually or automatically, conducting crash tests, performance tests, and actual or simulated test drives.

Quality Assurance is a broad term, explained on "*the continuous and consistent improvement and maintenance of process that enables the QC job*". As follows from the definition, QA focuses more on organizational aspects of quality management, monitoring the consistency of the production process.

Cost of Quality :
It is the most established, effective measure of quantifying and calculating the business value of testing. There are four categories to measure cost of quality: Prevention costs, Detection costs, Internal failure costs, and External failure costs.
These are explained as follows below.

1. **Prevention costs** include cost of training developers on writing secure and easily maintainable code
2. **Detection costs** include the cost of creating test cases, setting up testing environments, revisiting testing requirements.
3. **Internal failure costs** include costs incurred in fixing defects just before delivery.
4. **External failure costs** include product support costs incurred by delivering poor quality software.

Major parts of total cost are detecting defects and internal failure cost. But, these costs less than external failure costs. That's why testing provides good business value.

Software quality assurance

Software Quality: Software Quality is defined as the conformance to explicitly state functional and performance requirements, explicitly documented development standards, and inherent characteristics that are expected of all professionally developed software.

Quality Control: Quality Control involves a series of inspections, reviews, and tests used throughout the software process to ensure each work product meets the requirements place upon it. Quality control includes a feedback loop to the process that created the work product.

Quality Assurance: Quality Assurance is the preventive set of activities that provide greater confidence that the project will be completed successfully.

Quality Assurance focuses on how the engineering and management activity will be done?

As anyone is interested in the quality of the final product, it should be assured that we are building the right product.

It can be assured only when we do inspection & review of intermediate products, if there are any bugs, then it is debugged. This quality can be enhanced.

Software Quality Assurance

Software quality assurance is a planned and systematic plan of all actions necessary to provide adequate confidence that an item or product conforms to establish technical requirements.

A set of activities designed to calculate the process by which the products are developed or manufactured.

SQA Encompasses

- A quality management approach
- Effective Software engineering technology (methods and tools)
- Formal technical reviews that are tested throughout the software process
- A multitier testing strategy
- Control of software documentation and the changes made to it.
- A procedure to ensure compliances with software development standards
- Measuring and reporting mechanisms.

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 reviewed by all stakeholders. The plan governs quality assurance activities performed by the software engineering team and the SQA group. The plan identifies calculation to be performed, audits and reviews to be performed, standards that apply to the project, techniques for error reporting and tracking, documents to be produced by the SQA team, and amount of feedback provided to the software project team.
2. **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.

3. **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.
4. **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.
5. **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.
6. **Records any noncompliance and reports to senior management:** Non- compliance items are tracked until they are resolved.

Software reviews

Software Review

It 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).

Software review is an essential part of Software Development Life Cycle (SDLC) that helps software engineers in validating the quality, functionality and other vital features and components of the software. It is a whole process that includes testing the software product and it makes sure that it meets the requirements stated by the client.

Usually performed manually, software review is used to verify various documents like requirements, system designs, codes, test plans and test cases.

Objectives of Software Review:

The objective of software review is:

1. To improve the productivity of the development team.

2. To make the testing process time and cost effective.

3. To make the final software with fewer defects.

4. To eliminate the inadequacies.

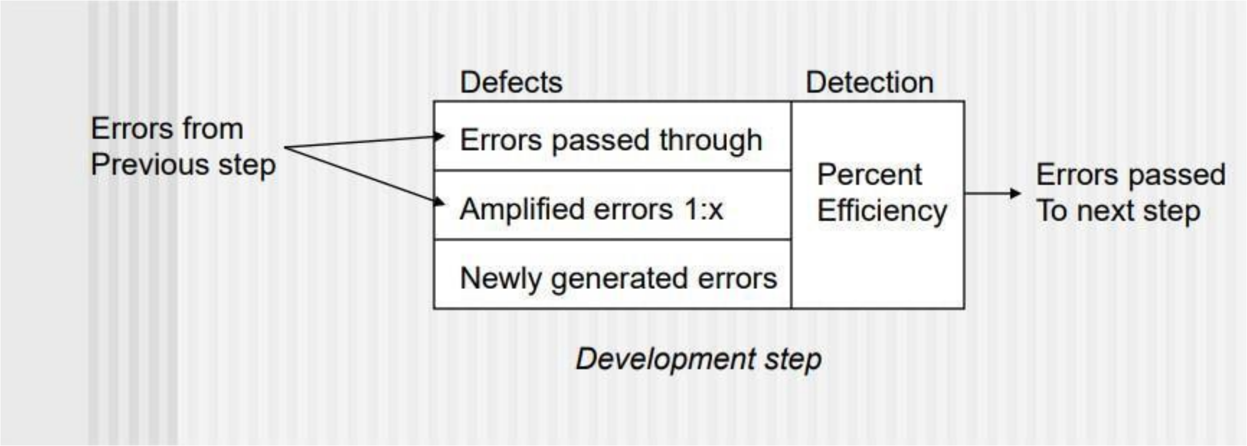
Cost impact of software defects

The primary objective of formal technical reviews is to find errors during the process so that they do not become defects after release of the software.

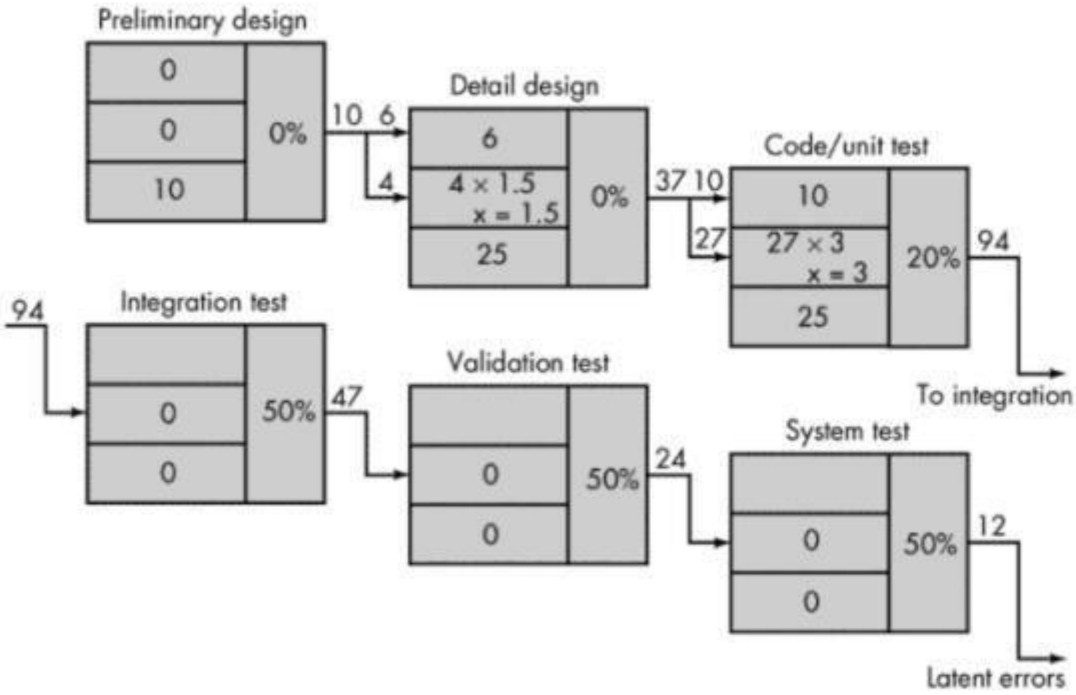
Defect amplification and removal

A defect amplification model can be used to illustrate the generation and detection of errors during the preliminary design, detail design, and coding steps of a software engineering process.

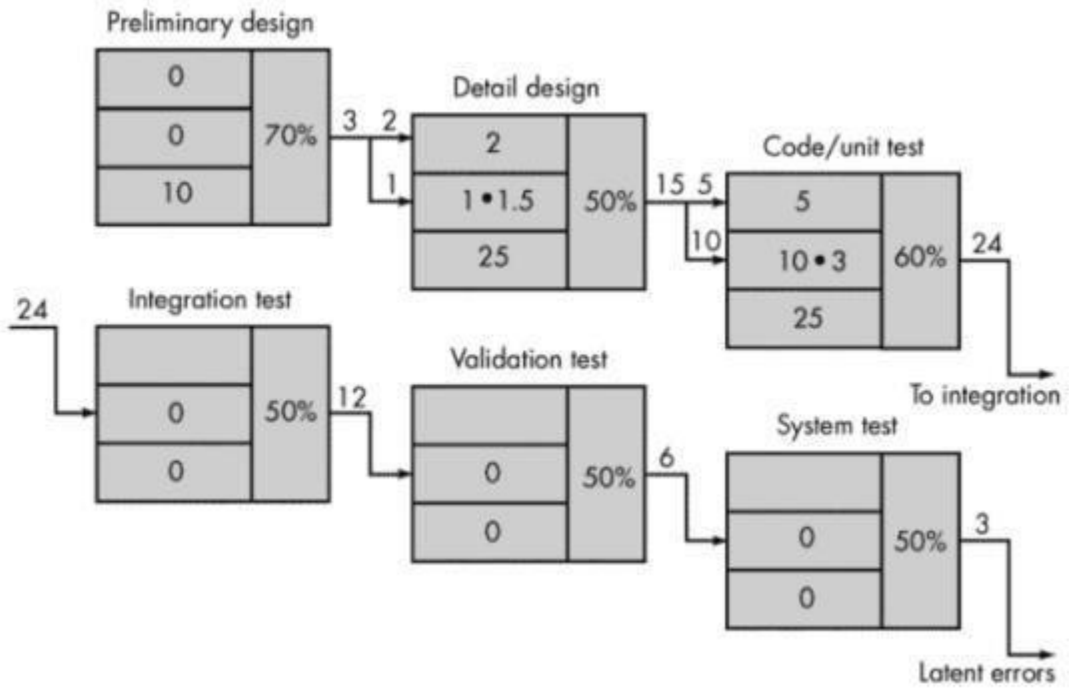
Defect amplification model:



Defect amplification – no reviews:



Defect amplification – reviews conducted:



Formal technical reviews

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.
- It helps to review the uniformity in software that is development in a uniform manner.
- To makes the project more manageable.

In addition, the purpose of FTR is to enable junior engineer to observer the analysis, design, coding and testing approach more closely. FTR also works to promote back up and continuity become familiar with parts of software they might not have seen otherwise. Actually, FTR is a class of reviews that include walkthroughs, inspections, round robin reviews and other small group technical assessments of software. Each FTR is conducted as meeting and is considered successful only if it is properly planned, controlled and attended.

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.

At the end of the review, all attendees of FTR must decide what to do.

1. Accept the product without any modification.
2. Reject the project due to serious error (Once corrected, another app need to be reviewed), or
3. Accept the product provisional (minor errors are encountered and are should be corrected, but no additional review will be required).

The decision was made, with all FTR attendees completing a sign-of indicating their participation in the review and their agreement with the findings of the review team.

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.

It answers three questions :-

1. What was reviewed ?
2. Who reviewed it ?
3. What were the findings and conclusions ?

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 software quality assurance

- Collect and categorize information (i.e., causes) about software defects that occur

- Attempt to trace each defect to its underlying cause (e.g., nonconformance to specifications, design error, violation of standards, poor communication with the customer)
- Using the Pareto principle (80% of defects can be traced to 20% of all causes), isolate the 20%

A generic example

- A sample of possible causes for defects:
- Incomplete or erroneous specifications
- Misinterpretation of customer communication
- Intentional deviation from specifications
- Violation of programming standards
- Errors in data representation
- Inconsistent component interface
- Errors in design logicϒ Incomplete or erroneous testing
- Inaccurate or incomplete documentation
- Errors in programming language translation of design
- Ambiguous or inconsistent human/computer interface

Six sigma

- Popularized by Motorola in the 1980s Is the most widely used strategy for statistical quality assurance
- Uses data and statistical analysis to measure and improve a company's operational performance Identifies and eliminates defects in manufacturing and servicerelated processes
- The "Six Sigma" refers to six standard deviations (3.4 defects per a million occurrences)

Three core steps

- <u>Define </u>customer requirements, deliverables, and project goals via well-defined
- methods of customer communication
- <u>Measure </u>the existing process and its output to determine current quality performance (collect defect metrics)
- <u>Analyze </u>defect metrics and determine the vital few causes (the 20%)
- Two additional steps are added for existing processes (and can be done inϒ parallel)
- <u>Improve </u>the process by eliminating the root causes of defects
- <u>Control </u>the process to ensure that future work does not reintroduce the causes of defects

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.

Measures of reliability and availability

Two meaningful metrics used in this evaluation are **Reliability** and **Availability**. Often mistakenly used interchangeably, both terms have different meanings, serve different purposes, and can incur different cost to maintain desired standards of service levels.

Availability refers to the percentage of time that the infrastructure, system, or solution remains operational under normal circumstances in order to serve its intended purpose. For cloud infrastructure solutions, availability relates to the time that the data center is accessible or delivers the intend IT service as a proportion of the duration for which the service is purchased.
The mathematical formula for Availability is :

Percentage of availability = (total elapsed time – sum of downtime)/total elapsed time

Reliability refers to the probability that the system will meet certain performance standards in yielding correct output for a desired time duration.

Reliability can be used to understand how well the service will be available in context of different real-world conditions. For instance, a cloud solution may be available with an SLA commitment of 99.999 percent, but vulnerabilities to sophisticated cyber-attacks may cause IT outages beyond the control of the vendor. As a result, the service may be compromised for several days, thereby reducing the effective availability of the IT service.

MTBF = (total elapsed time – sum of downtime)/number of failures

Where MTBF means mean time between failures

Software safety:

As systems and products become more and more dependent on software components it is no longer realistic to develop a system safety program that does not include the software elements.

Does software fail? We tend to believe that well written, well tested, safety critical software never fails. Experience proves otherwise with software making headlines when it actually does fail, sometimes critically. Software does not fail the same way hardware does, and the various failure behaviors we are accustomed to from the world of hardware are often not applicable to software. However, software does fail, and when it does, it can be just as catastrophic as hardware failures.

Safety-critical software

Safety-critical software is a creature very different from both non-critical software and safety-critical hardware. The difference lies in the massive testing program that such software undergoes.

The ISO 9000 QUALITY STANDARDS

A quality assurance system may be defined as the organizational structure, responsibilities, procedures, processes, and resources for implementing quality management.

Quality assurance systems are created to help organizations ensure their products and services satisfy customer expectations by meeting their specifications. These systems cover a wide variety of activities encompassing a product's entire life cycle including planning, controlling, measuring, testing and reporting, and improving quality levels throughout the development and manufacturing process. ISO 9000 describes quality assurance elements in generic terms that can be applied to any business regardless of the products or services offered.

The ISO 9000 standards have been adopted by many countries including all members of the European Community, Canada, Mexico, the United States, Australia, New Zealand, and the Pacific Rim. Countries in Latin and South America have also shown interest in the standards.

The ISO Approach to Quality Assurance Systems

The ISO 9000 quality assurance models treat an enterprise as a network of interconnected processes. For a quality system to be ISO compliant, these processes must address the areas identified in the standard and must be documented and practiced as described.

The ISO 9001 Standard

ISO 9001 is the quality assurance standard that applies to software engineering. The standard contains 20 requirements that must be present for an effective quality assurance system. Because the ISO 9001 standard is applicable to all engineering disciplines, a special set of ISO guidelines (ISO 9000-3) have been developed to help interpret the standard for use in the software process.