UNIT-I: Software and Software Engineering: The Nature of Software, The Unique Nature of WebApps , Software Engineering, Software Process, Software Engineering Practice, Software Myths.
Process Models: A Generic Process Model, Process Assessment and Improvement, Prescriptive Process Models, Specialized Process Models, The Unified Process, Personal and Team Process Models

Introduction

The term Software Engineering is composed of two words, software and engineering.
Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called software product.
Engineering is the application of scientific and practical knowledge to invent, design, build, maintain, and improve frameworks, processes, etc. The outcome of software engineering is an efficient and reliable software product.

Why is Software Engineering required?

- ✓ To manage Large software
- ✓ For more Scalability
- ✓ Cost Management
- ✓ To manage the dynamic nature of software
- ✓ For better quality Management

The Nature of Software

- Defining Software
- Software Application Domains
- Legacy Software

Defining Software

Software is a set of instructions or programs that are designed to perform specific tasks or functions on a computer system. It is a collection of computer code that is written in one or more programming languages and is used to control the behavior of a computer or other digital devices. Software can be classified into different types, including system software, application software, and utility software, and is used in a variety of industries and fields, such as business, education, entertainment, and science.

Characteristics Of Good Software

A software product can be judged by what it offers and how well it can be used.
This software must satisfy on the following grounds:

1) Functionality
2) Usability (User-Friendly)
3) Efficiency
4) Flexibility

5) Reliability
6) Portability
7) Integrity
8) Robustness

1) Functionality:

The functionality of software refers to its ability to perform and function according to design specification. In simple terms, software systems should function correctly, i.e. perform all the functions for which they are designed.

In order to look like the best software product, it must have a clear appearance, components, and functions.

2) Usability (User-Friendly):

The user-friendliness of the software is characterized by its ease of use. In other words, learning how to use the software should require less effort or time. Navigating the software is extremely important since it helps determine the journey the user takes within the software. This is imperative to ensuring visitors remain on your website and have a positive experience, which leads to an increase in sales and brand loyalty.

3) Efficiency:

Essentially, Efficiency refers to the software's ability to utilize human and system resources such as time, effort, CPU, memory, computation power, network bandwidth, files, databases, etc., as effectively and efficiently as possible. For a software project to succeed, efficiency is crucial.

4) Flexibility:

Software Flexibility refers to the ability of the software solution to adapt to potential or future changes in its requirements. When evaluating the flexibility of software, look at how simple it is to add, modify, or remove features without interfering with the current operation.

5) Reliability:

The reliability of a software product describes the likelihood it will operate without failure over a specified period of time under certain conditions. software reliability is measured as the availability of the software. The value should not be less than 99%. In reliability testing, the goal is not perfection, but achieving a level of reliability that is acceptable before a software product is released to customers.

6) Portability:

Software portability is a critical factor that cannot be ignored. Portability refers to the ability of software to work on different hardware platforms without any (or little) modifications needed.

7) Integrity:

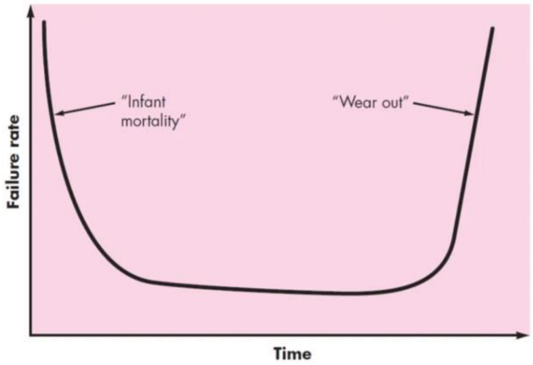
Integrity is the key for demonstrating the safety, security, and maintainability of your software.
The software cannot be modified without authorization. In these days of increased security threats, all software must include this factor.

8) Robustness:

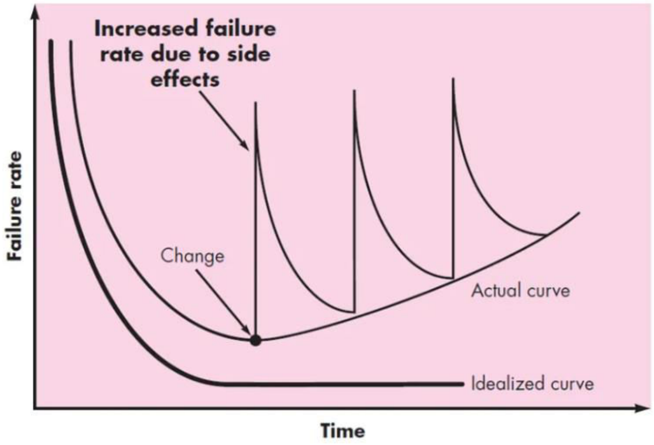
Robustness is the ability of a computer system to cope with errors during execution and cope with erroneous input. In other words, the software can keep on functioning in spite of being provided with invalid data.

1.Software is developed or engineered; it is not manufactured in the classical sense
2.Software doesn't "wear out."
3.Although the industry is moving toward component-based construction, most software continues to be custom built.

Failure curve for hardware:



Failure curve for software:



Software Application Domains

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

1. System Software

Here, the system is which is dealing with the hardware. System software is a collection of programs written to service other programs.

Ex: operating system, compilers, device drivers, etc.

2. Application Software

Application software consists of standalone programs that solve a specific need for end users.

Ex: Microsoft Word, spreadsheets, VLC media player, accounting applications, photo editor, mobile apps such as video games, WhatsApp, etc.

3. Engineering and Scientific software

It is special software to implement Engineering and Scientific applications. Engineering problems and quantitative analysis are carried out using automated tools. Scientific software is typically used to solve mathematical functions and calculations.

Ex: Computer Aided Design(CAM), Computer Aided Manufacturing(CAM), MATLAB,SPPS(for statistics).

4. Embedded software

Embedded means combining or adding. It resides within a product or system and is used to control, monitor the operation of equipment or machinery. The embedded software is residing in read-onlymemory (ROM) and not supposed to write anything on this software.

Ex: key pad control for a microwave oven, washing machine.

Ex: digital functions in an automobile such as fuel control, dashboard displays, and braking systems.

5. Product-line software

A group of products that a company creates under a single brand represents a product-line software. Designed to provide a specific capability for use by many different customers.

Ex: word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, and personal and business financial applications

6. Web-applications

Web application is application software that runs on a webserver, unlike computer-based software which are run on a local device operating system. Web applications are accessed by the user through a web browser with an active network connection. These applications are programmed using a client-server modelled structure.

Ex: web mail, e-commerce sites, online banking, etc...

7. Artificial intelligence software

Artificial Intelligence software is a computer programme which structure is as close as human brain. In the way the human brain thinks, in the same way the AI system is also thinks.

AI combined with machine Learning is used to provide users with the required functionality and make the business process a much simpler one.

With the help of AI, we can develop smart systems that will not only help us in businesses or offices but also at home.

Ex: self-driving cars, manufacturing robots, Google assistant, voice assistant, robotics, expert systems, pattern recognition (image and voice), artificial neural networks, game playing.etc...

Legacy software

Legacy software refers to software applications or systems that are outdated and no longer being actively developed or supported by their original developers. These software systems may have been created years or even decades ago, and have not been updated to meet the latest standards or requirements. Legacy software may still be in use by organizations or individuals, but may present challenges and risks due to compatibility issues with newer systems, security vulnerabilities, or a lack of support and maintenance.

Ex: Microsoft Windows XP, COBOL etc

The Unique Nature of WebApps

In the early days of the World Wide Web (1990 to 1995), websites consisted of little more than a set of linked hypertext files that presented information using text and limited graphics.

Today, WebApps have evolved into sophisticated computing tools that not only provide stand-alone function to the end user, but also have been integrated with corporate databases and business applications due to the development of HTML, JAVA, xml etc.

Attributes of WebApps :

Network Intensiveness

Concurrency
Unpredictable load
Performance
Availability
Data driven
Content Sensitive
Continuous evolution
Security

Network intensiveness.

A WebApp resides on a network and must serve the needs of a diverse community of clients.The network may enable worldwide access and communication (i.e., the Internet) or more limited access and communication(e.g., a corporate Intranet Network Intensiveness)

Concurrency : A large number of users may access the WebApp at one time. In many cases, the patterns of usage among end users will vary greatly.

Unpredictable load :

The number of users of the WebApp may vary by orders of magnitude from day to day. Ex:One hundred users may show up on Monday; 10,000 may use the system on Thursday.

Performance :

If a WebApp user must wait too long (for access, for server side processing, for client-side formatting and display), he or she may decide to go elsewhere.

Availability :

Although expectation of 100 percent availability is unreasonable, users of popular WebApps often demand access on a 24/7/365 basis.

Data driven :

The primary function of many WebApps is to use hypermedia to present text, graphics, audio, and video content to the end user.In addition, WebApps are commonly used to access information that exists on databases that are not an integral part of the Web-based environment (e.g., e-commerce or financial applications).

Content sensitive:

The quality and artistic nature of content remains an important ,Determinant of the quality of a WebApp.

Continuous evolution:

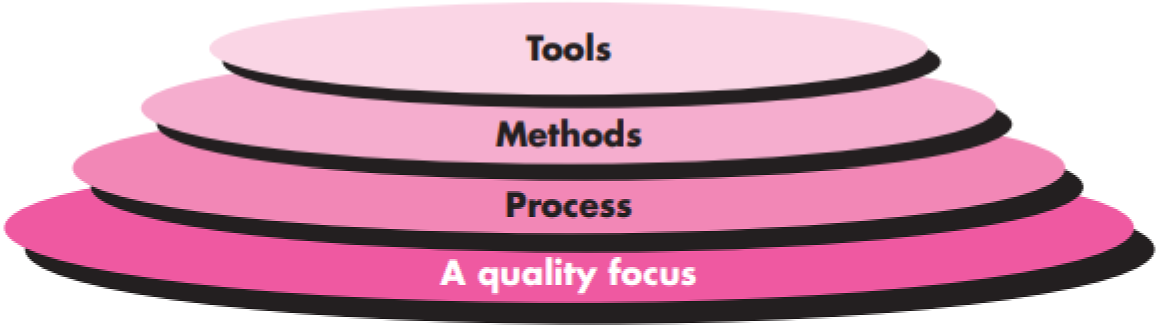
Unlike conventional application software that evolves over a series of planned, chronologically spaced releases, Web applications evolve continuously.

It is not unusual for some WebApps (specifically, their content) to be updated on a minute-by-minute schedule or for content to be independently computed for each request.

Security:

Because WebApps are available via network access, it is difficult, if not impossible, to limit the population of end users who may access the application. In order to protect sensitive content and provide secure mode of data transmission, strong security measures must be implemented.

Software Engineering



Layers in Software Engineering

Software engineering is a layered technology.

Software engineering tools provide automated or semiautomated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established

Software engineering methods provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support.

The foundation for software engineering is the **process layer.** The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework that must be established for effective delivery of software engineering technology. The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Any engineering approach (including software engineering) must rest on an organizational commitment to quality.

Software Process

A process framework establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity. In addition, the process framework encompasses a set of umbrella activities that are applicable across the entire software process.

A generic process framework for software engineering encompasses five activities:

- **Communication –** Communicate with stakeholders and customers to obtain objectives of the system and requirements for the software.
- **Planning –** Software project plan has details of resources needed, tasks and risk factors likely to occur, schedule.
- **Modelling –** Architectural models and design to better understand the problem and for work towards the best solution.
- **Construction –** Generation of code and testing of the system to rectify errors and ensuring all specified requirements are met.
- **Deployment –** Entire software product or partially completed product is delivered to the customer for evaluation and feedback.

Umbrella activities

Activities that occur throughout software process for better management and tracking of the project.

- **Software project tracking and control –** Compare the progress of the project with the plan and take steps to maintain a planned schedule.
- **Risk management –** Evaluate risks that can affect the outcome and quality of the software product.
- **Software quality assurance (SQA) –** Conduct activities to ensure the quality of the product.
- **Technical reviews –** Assessment of errors and correction done at each stage of activity.
- **Measurement –** All the measurements of the project and product features.
- **Software configuration management (SCM) –** Controlling and tracking changes in the software.
- **Reusability management –** Back up work products for reuse and apply the mechanism to achieve reusable software components.
- **Work product preparation and production –** Project planning and other activities used to create work product are documented.

SOFTWARE ENGINEERING PRACTICE

Generic software process model composed of a set of activities that establish a framework for software engineering practice.

Generic framework activities—communication, planning, modelling, construction, and deploymentAnd umbrella activities establish a skeleton architecture for software engineering work.

It Consists of The Essence Of Practice and General Principles

The Essence of Practice

The essence of software engineering practice as follows:
1. Understand the problem (communication and analysis).
2. Plan a solution (modelling and software design).
3. Carry out the plan (code generation).
4. Examine the result for accuracy (testing and quality assurance).

In the context of software engineering, these common-sense steps lead to a series of essential questions.

1) Understand the problem: It's sometimes difficult to admit, but most of us suffer from hubris when we're presented with a problem. We listen for a few seconds and then think. Understanding isn't always that easy. It's worth spending a little time answering a few simple questions:

- Who has a stake in the solution to the problem? That is, who are the stakeholders?
- What are the unknowns? What data, functions, and features are required to properly solve the problem?
- Can the problem be compartmentalized? Is it possible to represent smaller problems that may be easier to understand?
- Can the problem be represented graphically? Can an analysis model be created?

2) Plan the solution: Now you understand the problem (or so you think) and you can't wait to begin coding. Before you do, slow down just a bit and do a little design:

- Have you seen similar problems before? Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?
- Has a similar problem been solved? If so, are elements of the solution reusable?
- Can sub problems be defined? If so, are solutions readily apparent for the Sub problems?
- Can you represent a solution in a manner that leads to effective implementation? Can a design model be created?

3) Carry out the plan: The design you've created serves as a road map for the system you want to build. There may be unexpected detours, and it's possible that you'll discover an even better route as you go, but the "plan" will allow you to proceed without getting lost.

- Does the solution conform to the plan? Is source code traceable to the design model?
- Is each component part of the solution provably correct? Have the design and code been reviewed, or better, have correctness proofs been applied to the algorithm?

4) Examine the result: You can't be sure that your solution is perfect, but you can be sure that you've designed a sufficient number of tests to uncover as many errors as possible.

• Is it possible to test each component part of the solution? Has a reasonable testing strategy been implemented?

• Does the solution produce results that conform to the data, functions, and features that are required? Has the software been validated against all stakeholder requirements? It shouldn't surprise you that much of this approach is common sense. In fact, it's reasonable to state that a common-sense approach to software engineering will never lead you astray.

David Hooker's General Principles

1. The Reason It All Exists
2. Keep It Simple, Stupid!(KISS)
3. Maintain the Vision
4. What You Produce, Others Will Consume
5. Be Open to the Future
6. Plan Ahead for Reuse
7. Think!

1. The Reason It All Exists:

A software system exists for one reason: to provide value to its users. All decisions should be made with this in mind.

2. Keep It Simple, Stupid!(KISS) :

- There are many factors to consider in any design effort.
- Thus, all software design should be as simple as possible but not simpler to ease understanding
- and maintenance of the software.

3. Maintain the Vision:

- A clear vision is essential to the success of a software project.
- Compromising the architectural vision of a software system weakens and break even welldesigned systems.

4. What You Produce, Others Will Consume :

- Always specify, design, and implement knowing someone else will have to understand what you are doing.
- The audience for any product of software development is potentially large.
- Someone may have to debug the code you write, and that makes them a user of your code.
- Making their job easier adds value to the system

5. Be Open to the Future:

- A system with a long lifetime has more value
- Systems must be ready to adapt to these and other changes. And systems do this successfully are those that have been designed this way from the start

6. Plan Ahead for Reuse:

- Reuse saves time and effort
- The reuse of code and designs has been proclaimed as a major benefit of using object- oriented technologies.
- Planning ahead for reuse reduces the cost and increases the value of both the reusable components and the systems into which they are incorporated.

7. Think! :

When you think about something (Placing clear, complete thought before action) you are more likely to do it right (more importantly the first time).

SOFTWARE MYTHS

Myth: A myth is a false belief or wrong assumptions.

Software Myth: In relation to computer software myth is nothing but misinformation, misunderstanding or confusion in software development filed.

Many software problems arise due to myths that are formed during the initial stages of software development.

In software Development myth arises at three levels:

1. Management Level Myth

2. Customer Level Myth

3. Practitioner's / Developer Level Myth

1. Management Myths:

Managers with software responsibility, like managers in most disciplines, are often under pressure to maintain budges, keep schedules from slipping, and improve quality.

Myth (1): Manager thinks that "when needed, we can add more programmers for faster development".

Reality:

- Training must be given to new comers.
- Spend time for training people.
- Cannot be developed fast

Myth (2): We already have a book that's full of standards and procedures for building software. It will provide the developer everything that he needs in the development process.

Reality:

The book Exists but question arises:

- Are software developers aware about this book?
- Does it contain all modern practices?
- Does it focus quality?
- Is it actually used by developer?

Myth (3): If I decide to outsource the software project to a third party. I can just relax and let that firm build it.

Reality:

If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Myth (4): Manager thinks "there is no need to change approach to software development.We can develop same kind of software that we develop 10 year ago"

Reality:

- Quality of software needs to improve according to customer demands.
- Customers' demands change time to time.

2. Customer Myths :

A customer who requests computer software may be a person at the next desk, a technical group down the hall, the marketing/sales department, or an outside company that has requested software under contract.

Myth (1):Only the general statement is sufficient and no need to mention detail project requirements.

Reality:

- Other details are also essential. Unambiguous requirements.
- Customer must provide design, validation criteria.
- During software development customer and developer communication is essential.

Myth (2): software requirements continually change, but change can be easily accommodated because software is flexible.

Reality:

• When requirements changes are requested early(before design or code has been started), the cost impact is relatively small.

• Otherwise cost is so high.

3. Practitioner's/Developer Myths:

Myths that are still believed by software developers by over 50 years of programming culture.

Myth (1): developers think "once we wrote program and get it to work, our job is done".

Reality:

Almost 60 to 80 percent work is required after delivering the project to customer for first time.

Myth (2): Until I get the program "running" I have no way of assessing its quality.

Reality:

• Effective SQA can be applied during project development.

• FTR are conducted to assure quality of software.

• FTR is meeting conducted by technical staff at any stage of software development.

Myth (3): The only deliverable work product for a successful project is the working program.

Reality:

• A working program is only one part of a software configuration that includes many elements.

• A variety of work products(e.g., models, documents, plans)provide a foundation for successful engineering and,
• More important, guidance for software support.

Myth (4):

Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

Reality:

• Software engineering is not about creating documents. It is about creating a quality product.
• Better quality leads to reduced rework. And reduced rework results in faster delivery times.

Process Models:

SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

Software development life cycle is a step-by-step procedure or a standard procedure to develop new software.

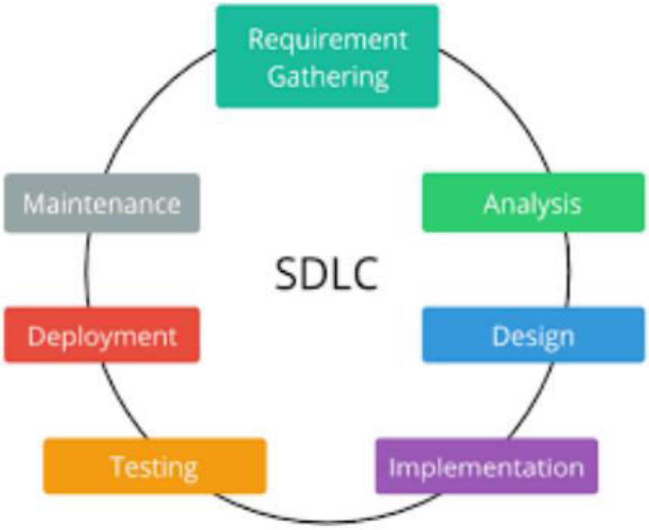
Why SDLC

Some of the reasons why SDLC is important in Software Development are as follows.

- **It provides visibility of a project plan to all the involved stakeholders**
- **It helps us to avoid project risks**
- **It allows us to track and control the project**
- **It doesn't conclude until all the requirements have been achieved**

7 Phases of SDLC (Software Development Life Cycle Phases)

Phase 1: Requirement Phase
Phase 2: Analysis Phase
Phase 3: Design Phase
Phase 4: Development Phase
Phase 5: Testing Phase
Phase 6: Deployment Phase
Phase 7: Maintenance Phase



1. Requirement Phase

Requirement gathering and analysis is the most important phase in the software development lifecycle. Requirement phase is the first step of the SDLC. Business Analyst collects the requirement from the Customer/Client as per the clients business needs and documents the requirements in the Business Requirement Specification (document name varies depends upon the Organization. Some examples are Customer Requirement Specification (CRS), Business Specification (BS), etc., and provides the same to Development Team.

2. Analysis Phase

Once the requirement gathering and analysis is done the next step is to define and document the product requirements and get them approved by the customer. This is done through the SRS (Software Requirement Specification) document. SRS consists of all the product requirements to be designed and developed during the project life cycle. Key people involved in this phase are Project Manager, Business Analyst and Senior members of the Team. The outcome of this phase is the Software Requirement Specification.

3. Design Phase

It has two steps:

- High-Level Design (HLD) – It gives the architecture of the software product to be developed and is done by architects and senior developers.
- Low-Level Design (LLD) – It is done by senior developers. It describes how each and every feature in the product should work and how every component should work. Here, only the

design will be there and not the code. The outcome from this phase is High-Level Document and Low-Level Document which works as an input to the next phase.

4. Development Phase

Developers of all levels (seniors, juniors, fresher's) involved in this phase. This is the phase where we start building the software and start writing the code for the product. The outcome from this phase is Source Code Document (SCD) and the developed product.

5. Testing Phase

When the software is ready, it is sent to the testing department where Test team tests it thoroughly for different defects. They either test the software manually or using automated testing tools depends on the process defined in STLC (Software Testing Life Cycle) and ensure that each and every component of the software works fine. Once the QA makes sure that the software is errorfree, it goes to the next stage, which is Implementation. The outcome of this phase is the Quality Product and the Testing Artifacts.

6. Deployment Phase

After successful testing, the product is delivered/deployed to the customer for their use. Deployment is done by the Deployment/Implementation engineers. Once when the customers startusing the developed system then the actual problems will come up and needs to be solved from time to time.

7. Maintenance Phase

Fixing the issues found by the customer comes in the maintenance phase. 100% testing is not possible – because, the way testers test the product is different from the way customers use the product. Maintenance should be done as per SLA (Service Level Agreement.

SDLC or Process Models Divided into 3 types.They are

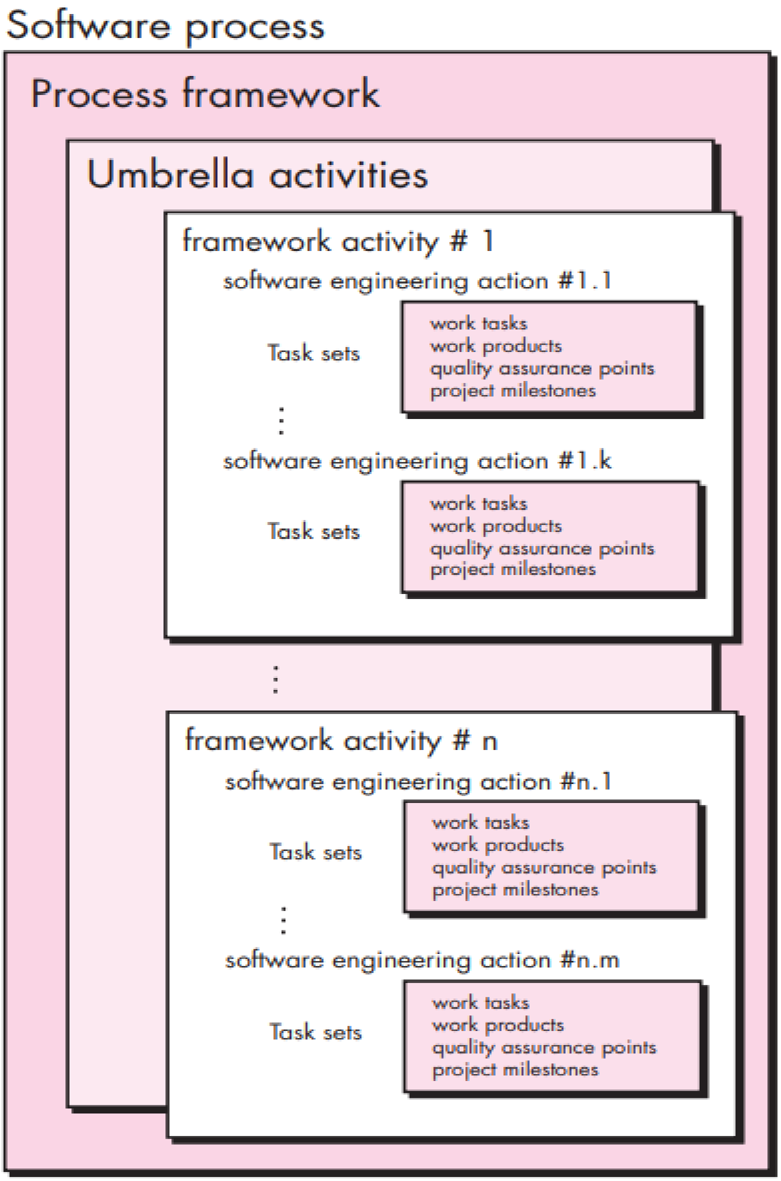
- A Generic Process Model
- Prescriptive Process Models
- Specialized Process Models

A Generic Process Model:

The Generic process model is an abstraction of the software development process. It specifies the stages and order of a process. Generic Process Model will define the following:

i. The tasks to be performed.
ii. The input and output of each task.
iii. The pre and post conditions for each task.
iv. The flow and sequence of each task.

A generic process framework for software engineering defines five framework activities—communication, planning, modeling,construction, and deployment. In addition, a set of umbrella activities—project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others—are applied throughout the process



It is categorized into

- Defining a Framework Activity
- Identifying a Task Set
- Process Patterns

Defining a Framework Activity

1)Make contact with stakeholder via telephone.
2)Discuss requirements and take notes.
3)Organize notes into a brief written statement of requirements.
4)E-mail to stakeholder for review and approval

Identifying a Task Set

Referring above Figure, each software engineering action (e.g., elicitation, an action associated with the communication activity) can be represented by a number of different task sets—each a collection of software engineering work tasks, related work products, quality assurance points, and project milestones. You should choose a task set that best accommodates the needs of the project and the characteristics of your team. This implies that a software engineering action can be adapted to the specific needs of the software project and the characteristics of the project team.

Process Patterns

1.Stage pattern—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following) that are relevant to the stage (framework activity). An example of a stage pattern might be EstablishingCommunication. This pattern would incorporate the task pattern RequirementsGathering and others.

2. Task pattern—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., RequirementsGathering is a task pattern).

3. Phase pattern—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature. An example of a phase pattern might be SpiralModel or Prototyping.3

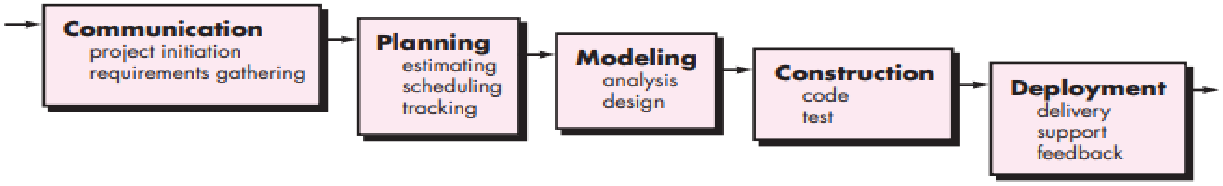
Perspective Process Model

Perspective Process Model are categorized into

- The Waterfall Model
- Incremental Process Models
- Evolutionary Process Models
- Concurrent Models
- A Final Word on Evolutionary Processes

The Waterfall Model

The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach6 to software development that begins with customer specification of requirements and progresses through planning, modeling, construction, and deployment, culminating in ongoing support of the completed software.



- **Communication –** Communicate with stakeholders and customers to obtain objectives of the system and requirements for the software.
- **Planning –** Software project plan has details of resources needed, tasks and risk factors likely to occur, schedule.
- **Modelling –** Architectural models and design to better understand the problem and for work towards the best solution.
- **Construction –** Generation of code and testing of the system to rectify errors and ensuring all specified requirements are met.
- **Deployment –** Entire software product or partially completed product is delivered to the customer for evaluation and feedback.

Advantages of waterfall model:

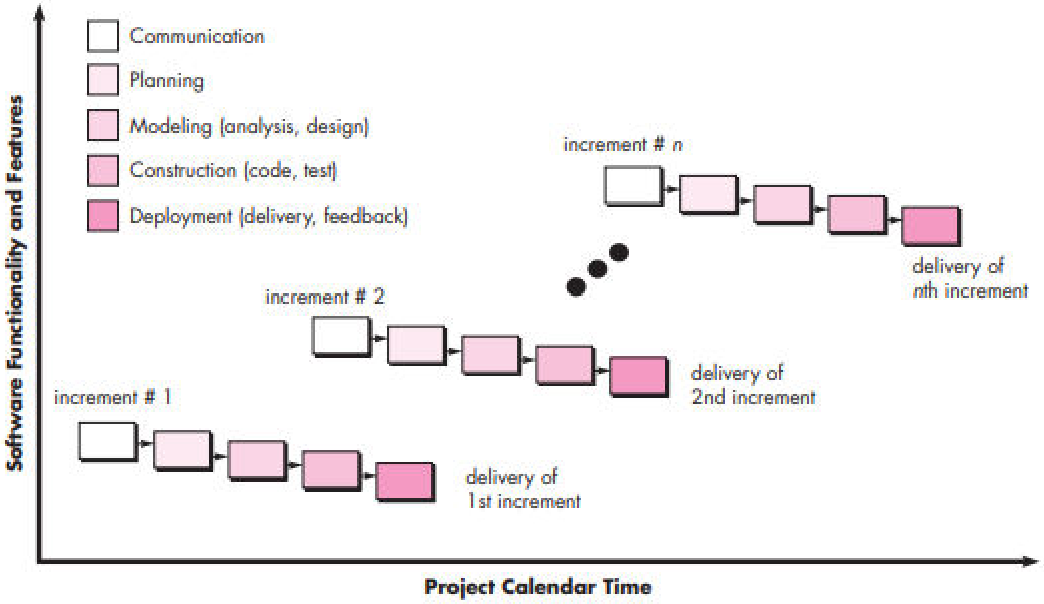
1) In the Waterfall model, the requirement should be clear.
2) It is suitable for a smaller project where needs are well understood.
3) This model is easy to understand, as well as easy to use.
4) It will allow us to arrange the tasks efficiently.
5) In this model, release level changes are allowed.
6) In this model, the procedure and the results are well documented.

Disadvantages of waterfall model:

1) This model has no parallel deliverable, which means that two teams can work together.
2) The waterfall model doesn't provide the requirement changes and requirement review.
3) Previously, when the waterfall is invented, there is no concept of testing, that's why the
developer is used to test the application.
4) In between, changes are not allowed because one phase is dependent on another stage.
5) Backward tracking is not possible.
6) It is a time-consuming process

Incremental Process Models

The incremental model combines elements of linear and parallel process flows



- The incremental model (also known as iterative enhancement model) comprises the features of waterfall model in an iterative manner.
- The first increment in this model is generally a core product.
- Each increment builds the product and submits it to the customer for any suggested modifications.
- The next increment implements on the customer's suggestions and add additional requirements in the previous increment.
- This process is repeated until the product is finished.

For example: word-processing software developed using the incremental paradigm might deliver basic file management editing, and document production functions in the first increment; more sophisticated editing, and document production capabilities in the second increment; spelling and grammar checking in the third increment; and advanced page layout capability in the fourth increment

When to use the Incremental Models???

- Requirements of the system are clearly understood.
- When demand for an early release of a product arises.
- When software engineering team are not very well skilled or trained.
- When high-risk features and goals are involved.
- Such methodology is more in use for web application and product based companies.

Advantages of incremental process model

- Flexible to change requirements.
- Changes can be done throughout the development stages.
- Errors are reduced since the product is tested by the customer in each phase.
- Working software available at the early stage of the process.

- Easy to test because of small iterations.
- The initial cost is lower.

Disadvantages of incremental process model

- Requires good planning and design.
- Modules and interfaces should be well defined.
- The total cost is high.
- Demands a complete planning strategy before commencement.
- Refining requirements in each iteration may affect system architecture.
- Breaking the problem into increments needs skilful management supervising

Evolutionary process Model

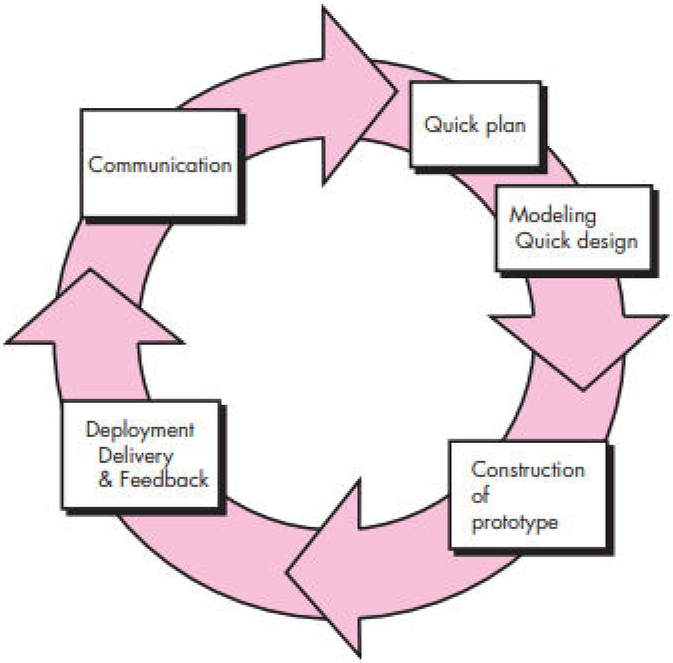
In evolutionary we have two different models

1) Prototype model

2) Spiral model

1) Prototype model

The prototype model requires that before carrying out the development of actual software, a
working prototype of the system should be built. A prototype is a toy implementation of the system



1.Communication(Requirements gathering and analysis): A prototyping model begins with requirements analysis and the requirements of the system are defined in detail. The user is interviewed in order to know the requirements of the system.

2. Planning & Quick design: When requirements are known, a preliminary design or quick design for the system is created. It is not a detailed design and includes only the important aspects of the system, which gives an idea of the system to the user. A quick design helps in developing the prototype.

3. Construction of prototype: Information gathered from quick design is modified to form the first prototype, which represents the working model of the required system Design.

Deployement,Delivery & feedback:

this stage undergoes the following

1. User evaluation: Next, the proposed system is presented to the user for thorough evaluation of the prototype to recognize its strengths and weaknesses such as what is to be added or removed. Comments and suggestions are collected from the users and provided to the developer.

2. Refining prototype: Once the user evaluates the prototype and if he is not satisfied, the current prototype is refined according to the requirements. That is, a new prototype is developed with the additional information provided by the user. The new prototype is evaluated just like the previous prototype. This process continues until all the requirements specified by the user are met. Once the user is satisfied with the developed prototype, a final system is developed on the basis of the final prototype.

3. Engineer product: Once the requirements are completely met, the user accepts the final prototype. The final system is evaluated thoroughly followed by the routine maintenance on regular basis for preventing large-scale failures and minimizing downtime.

Advantage of Prototype Model

1.Reduce the risk of incorrect user requirement

2.Good where requirement are changing/uncommitted

3.Regular visible process aids management

4.Support early product marketing

5.Reduce Maintenance cost.

6.Errors can be detected much earlier as the system is made side by side.

Disadvantage of Prototype Model

1.An unstable/badly implemented prototype often becomes the final product.

2.Require extensive customer collaboration

- Costs customer money
- Needs committed customer
- Difficult to finish if customer withdraw
- May be too customer specific, no broad market

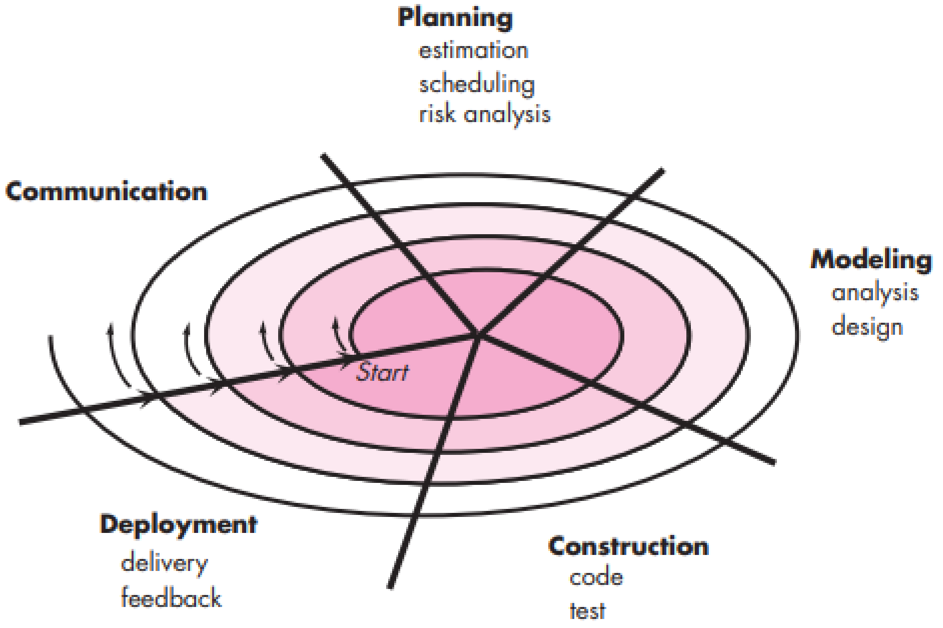
3.Difficult to know how long the project will last.

4.Easy to fall back into the code and fix without proper requirement analysis, design,

customer evaluation, and feedback.
5.Prototyping tools are expensive.
6.Special tools & techniques are required to build a prototype.
7.It is a time-consuming process.

2. Spiral Model

The spiral model is a realistic approach to the development of large-scale systems and software.
Spiral model is also known as Meta Model because it subsumes all the other SDLC models. In its diagrammatic representation, it looks like a spiral with many loops, that's the reason it's called as Spiral. Each loop of the spiral is called a Phase of the software development process. This model has capability to handle risks.



When to use Spiral model:

- When costs and risk evaluation is important
- For medium to high-risk projects
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- New product line
- Significant changes are expected (research and exploration)

Advantages of Spiral model:

- High amount of risk analysis hence, avoidance of Risk is enhanced.
- Good for large and mission-critical projects.
- Strong approval and documentation control.

- Additional Functionality can be added at a later date.
- Continuous customer involvement so better customer satisfaction.
- Software is produced early in the software life cycle.

Disadvantages of Spiral model:

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

SPECIALIZED PROCESS MODEL

Special process models take many features from one or more conventional models. However these special models tend to be applied when a narrowly defined software engineering approach is chosen.

Types in Specialized process models:

1. Component based development (Promotes reusable components)
2. The formal methods model (Mathematical formal methods are backbone here)
3. Aspect oriented software development (Uses crosscutting technology)

1. Component based development:

- ➢ Available component-based products are researched and evaluated for the application domain in question.
- ➢ Component integration issues are considered.
- ➢ A software architecture is designed to accommodate the components.
- ➢ Components are integrated into the architecture.
- ➢ Comprehensive testing is conducted to ensure proper functionality

- The component based development model incorporates many of the characteristics of the spiral model.
- It is evolutionary in nature, demanding an iterative approach to the creation of software.
- However, the model focuses on pre-packaged software components that are from existing software modules.
- It promotes software reusability.

Advantages:

1. Leads to software reuse, which provides number of benefits
2. 70% reduction in the development cycle time.

Disadvantages:

1. Components library must be robust.
2. Performance may degrade.

2. The Formal methods model:

- ➢ Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous mathematical notation
- ➢ Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily discover and correct errors

- The development of formal models is currently quite time consuming and expensive.
- Because few software developers have the necessary background to apply formal methods, extensive training is required.
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers

- Comprises set of activities that leads to formal mathematical specification of computer software.
- Enables a software engineer to specify, develop and verify a computer based system by applying mathematical notation.

  Advantages

  1. Problems of ambiguity, incompleteness and inconsistency can be managed by this method.
  2. Provides a base for verification therefore enable a software engineer to discover and correct undetected errors also.

  DisAdvantages:

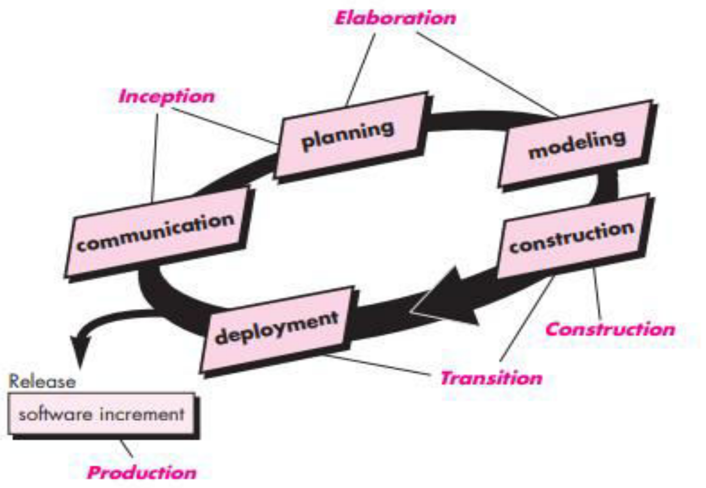
  1. Using this method in current scenario is time consuming and expensive.
  2. Extensive training is required for applying this method as few developers have the necessary background to work with this method.
  3. It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

3. Aspect oriented software development

- Aspect Oriented Software Development (AOSD) often referred to as aspect oriented programming (AOP), a relatively new paradigm that provides process and methodology for defining, specifying designing and constructing aspects.
- It addresses limitations inherent in other approaches, including object-oriented programming. AOSD aims to address crosscutting concerns by providing means for systematic identification, separation, representation and composition.
- This results in better support for modularization hence reducing development, maintenance and evolution costs

The Unified Process

Rational Unified Process (RUP) is a software development process for object-oriented models. It is also known as the Unified Process Model. It is created by Rational corporation and is designed and documented using UML (Unified Modeling Language). This process is included in IBM Rational Method Composer (RMC) product. IBM (International Business Machine Corporation) allows us to customize, design, and personalize the unified process. RUP is proposed by Ivar Jacobson, Grady Bootch, and James Rambaugh. Some characteristics of RUP include use-case driven, Iterative (repetition of the process), and Incremental (increase in value) by nature, delivered online using web technology, can be customized or tailored in modular and electronic form, etc. RUP reduces unexpected development costs and prevents wastage of resources.



Phases of RUP: There is total of five phases of the life cycle of RUP:

1. **Inception –**
   - Communication and planning are the main ones.
   - Identifies the scope of the project using a use-case model allowing managers to estimate costs and time required.
   - Customers' requirements are identified and then it becomes easy to make a plan for the project.
   - The project plan, Project goal, risks, use-case model, and Project description, are made.
   - The project is checked against the milestone criteria and if it couldn't pass these criteria then the project can be either canceled or redesigned.
2. **Elaboration –**
   - Planning and modeling are the main ones.
   - A detailed evaluation and development plan is carried out and diminishes the risks.
   - Revise or redefine the use-case model (approx. 80%), business case, and risks.
   - Again, checked against milestone criteria and if it couldn't pass these criteria then again project can be canceled or redesigned.
   - Executable architecture baseline.
3. **Construction –**
   - The project is developed and completed.
   - System or source code is created and then testing is done.
   - Coding takes place.
4. **Transition –**

- The final project is released to the public.
- Transit the project from development into production.
- Update project documentation.
- Beta testing is conducted.
- Defects are removed from the project based on feedback from the public.

5. **Production –**
   - The final phase of the model.
   - The project is maintained and updated accordingly.

Advantages:

1. It provides good documentation, it completes the process in itself.
2. It provides risk-management support.
3. It reuses the components, and hence total time duration is less.
4. Good online support is available in the form of tutorials and training.

Disadvantages:

1. Team of expert professional is required, as the process is complex.
2. Complex and not properly organized process.
3. More dependency on risk management.
4. Hard to integrate again and again.

PERSONAL SOFTWARE PROCESS (PSP):

Personal Software Process (PSP) is the skeleton or the structure that assist the engineers in finding a way to measure and improve the way of working to a great extent. It helps them in developing their respective skills at a personal level and the way of doing planning, estimations against the plans.

PSP helps software engineers to engineers to identify the errors early and to understand the types of errors.

Goal of PSP:

The goal of PSP is to provide software engineers with disciplined methods for improving personal software development.

PSP Activities

1.Planning: This activity isolates requirements and specifications to be decided prior to the development and estimates the size and the cost of the project. Defect estimation is also made. Development tasks are identified and scheduled is created.

2.High-Level Design: External specifications and requirements for each component to be constructed and developed. Component design is also created. Prototypes are developed if requirements are complex.

3.High Level Design Review: Formal verification methods are applied to uncover errors in the design.

4.Development: The component level design is reviewed and refined. The code is generated, reviewed, compiled and tested.

5.Post-mortem: Using Metrics and measures the effectiveness of the process is determined. It provides guidance for modification of process and its improvement.

TEAM SOFTWARE PROCESS MODEL (TSP)

What is TSP?

- TSP (Team Software Process) is a guideline for software product development teams.
- TSP focuses on helping development teams to improve their quality and productivity to better meet goals of cost and progress.
- TSP is designed for groups ranging from 2 persons to 20 persons. TSP can be applied to large multiple-group processes for up to 150 persons.
- There are 8 steps for implementing PSP and TSP. Each step is focused on solving particular process problems.

Goal of TSP

The goal of the TSP is to build a ―Self-Directed‖ project teams that organizes itself to produce high quality software.

TSP Activities:

1) Project Launch: It reviews project objective and describes the TSP structure and content. It assigns need and roles to the team members and describes the customers need statement.

2) High Level Design: it creates the high level design, specify the design, inspect the design and develop the integration plan.

3) Implementation: This uses the PSP to implement the modules and the functions.

4) Integration and Testing: Testing builds and integrates the system.

5) Post-mortem: Writes the cycle report and produces peer and team review

Requirements Analysis and Specification: Requirements Gathering and Analysis, Software Requirement Specification (SRS), Formal System Specification.
Software Design: Overview of the Design Process, How to Characterize of a Design? Cohesion and Coupling, Layered Arrangement of Modules, Approaches to Software Design, Developing the DFD Model of a System

Requirements Analysis and Specification

Introduction:

- Experienced developers take considerable time to understand the exact requirements of the customer and to meticulously document those.
- They know that without a clear understanding of the problem and proper documentation of the same, it is impossible to develop a satisfactory solution.
- The goal of the requirements analysis and specification phase is to clearly understand the customer requirements and to systematically organise the requirements into a document called the Software Requirements Specification (SRS) document.

Who carries out requirements analysis and specification???

- System analysts will collect data from the customers and analyse the collected data to conceptualise what exactly needs to be done.
- After understanding the precise user requirements, the analyst analyse the requirements to remove the inconsistencies and incompleteness.
- Then proceed to write software requirement specification document.
- The SRS document is the final outcome of the requirements analysis and specification phase.

Requirements analysis and specification phase mainly involves carrying out the following two important activities:

1. Requirements gathering and analysis
2. Software Requirement Specification (SRS)
3.Formal System Specification.

---

1. REQUIREMENTS GATHERING AND ANALYSIS

We can conceptually divide the requirements gathering and analysis activity to two separate tasks:

a) Requirements gathering
b) Requirements analysis

Requirements Gathering:

- Requirements gathering is also popularly known as requirements elicitation.
- The primary objective of the requirements gathering task is to collect the requirements from the stakeholders.
- A stakeholder is a source of the requirements and is usually a person, or a group of persons
- who either directly or indirectly are concerned with the software.

How to gather the requirements???

1. Studying existing documentation
2. Interview
3. Prototyping
4. Survey/Questionnaire

5. Task analysis
6. Scenario analysis

2) Requirements Analysis

The main purpose of the requirements analysis activity is to analyse the gathered requirements to remove all ambiguities,incompleteness, and inconsistencies from the gathered customer requirements and to obtain a clear understanding of the software to be developed.

How to analyse the requirements???

The following basic questions pertaining to the project should be clearly understood by the analyst before carrying out analysis:

- What is the problem?
- Why is it important to solve the problem?
- What exactly are the data input to the system and what exactly are the data output by the system?
- What are the possible procedures that need to be followed to solve the problem?
- What are the likely complexities that might arise while solving the problem?
- If there are external software or hardware with which the developed software has to interface, then what should be the data interchange formats with the external systems?

During requirements analysis, the analyst needs to identify and resolve three main types of problems in the requirements:

- Anomaly
- Inconsistency
- Incompleteness

Anomaly: It is an anomaly is an ambiguity in a requirement.

Inconsistency: Two requirements are said to be inconsistent, if one of the requirements contradicts the other .

Incompleteness: An incomplete set of requirements is one in which some requirements have been overlooked.

An experienced analyst can detect most of these missing features and suggest them to the customer for his consideration and approval for incorporation in the requirements.

Software Requirements Specification (SRS)

The SRS document is the final outcome of the requirements analysis and specification phase.

How is the SRS document validated?

- Once the SRS document is ready, it is first reviewed internally by the project team to ensure that it accurately captures all the user requirements, and that it is understandable, consistent, unambiguous, and complete.
- The SRS document is then given to the customer for review.
- After the customer has reviewed the SRS document and agrees to it, it forms the basis for all future development activities and also serves as a contract document between the customer and the development organisation.

Users of SRS Document

- Users, customers, and marketing personnel
- Software developers
- Test engineers

- User documentation writers
- Project managers
- Maintenance engineers

Characteristics of a Good SRS Document

1. **Concise:**The SRS document should be concise and at the same time unambiguous, consistent, and complete.
2. **Implementation-independent:** The SRS should be free of design and implementation decisions unless those decisions reflect actual requirements.
3. **Traceable:** It should be possible to trace a specific requirement to the design elements that implement it and vice versa. Similarly, it should be possible to trace a requirement to the code segments that implement it and the test cases that test this requirement and vice versa.
4. **Verifiable:** All requirements of the system as documented in the SRS document should be verifiable.
5. **Modifiable**
6. **Identification of response to undesired events**

Attributes of Bad SRS Documents:

The most damaging problems are incompleteness, ambiguity, and contradictions. There are many other types problems that a specification document might suffer from. By knowing these problems, one can try to avoid them while writing an SRS document.

- **Over-specification:** It occurs when the analyst tries to address the "how to" aspects in the SRS document. For example, in the library automation problem, one should not specify whether the library membership records need to be stored indexed on the member's first name or on the library member's identification (ID) number . Over-specification restricts the freedom of the designers in arriving at a good design solution.
- **Forward references:** One should not refer to aspects that are discussed much later in the SRS document. Forward referencing seriously reduces readability of the specification.
- **Wishful thinking:** This type of problems concern description of aspects which would be difficult to implement.
- **Noise:**The term noise refers to presence of material not directly relevant to the software development process.

Example for SRS(Note: In examination write the headings)

Format of the SRS Document:

1. Introduction

1.1 Purpose

<Identify the product whose software requirements are specified in this document, including the revision or release number. Describe the scope of the product that is covered by this SRS, particularly if this SRS describes only part of the system or a single subsystem.>

1.2 Document Conventions

<Describe any standards or typographical conventions that were followed when writing this SRS, such as fonts or highlighting that have special significance. For example, state whether priorities for higher-level requirements are assumed to be inherited by detailed requirements, or whether every requirement statement is to have its own priority.>

1.3 Intended Audience and Reading Suggestions

<Describe the different types of reader that the document is intended for, such as developers, project managers, marketing staff, users, testers, and documentation writers. Describe what the rest of this SRS contains and how it is organized. Suggest a sequence for reading the document, beginning with the overview sections and proceeding through the sections that are most pertinent to

each reader type.>

1.4 Product Scope

<Provide a short description of the software being specified and its purpose, including relevant benefits, objectives, and goals. Relate the software to corporate goals or business strategies. If a separate vision and scope document is available, refer to it rather than duplicating its contents here.>

1.5 References

<List any other documents or Web addresses to which this SRS refers. These may include user interface style guides, contracts, standards, system requirements specifications, use case documents, or a vision and scope document. Provide enough information so that the reader could access a copy of each reference, including title, author, version number, date, and source or location.>

2. Overall Description

2.1 Product Perspective

<Describe the context and origin of the product being specified in this SRS. For example, state whether this product is a follow-on member of a product family, a replacement for certain existing systems, or a new, self-contained product. If the SRS defines a component of a larger system, relate the requirements of the larger system to the functionality of this software and identify interfaces between the two. A simple diagram that shows the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.>

2.2 Product Functions

<Summarize the major functions the product must perform or must let the user perform. Details will be provided in Section 3, so only a high level summary (such as a bullet list) is needed here. Organize the functions to make them understandable to any reader of the SRS. A picture of the major groups of related requirements and how they relate, such as a top level data flow diagram or object class diagram, is often effective.>

2.3 User Characteristics

<Identify the various user classes that you anticipate will use this product. User classes may be differentiated based on frequency of use, subset of product functions used, technical expertise, security or privilege levels, educational level, or experience. Describe the pertinent characteristics of each user class. Certain requirements may pertain only to certain user classes. Distinguish the most important user classes for this product from those who are less important to satisfy.>

2.4 Operating Environment

<Describe the environment in which the software will operate, including the hardware platform, operating system and versions, and any other software components or applications with which it must peacefully coexist.>

2.5 Design and Implementation Constraints

<Describe any items or issues that will limit the options available to the developers. These might include: corporate or regulatory policies; hardware limitations (timing requirements, memory requirements); interfaces to other applications; specific technologies, tools, and databases to be used; parallel operations; language requirements; communications protocols; security considerations; design conventions or programming standards (for example, if the customer's organization will be responsible for maintaining the delivered software).>

2.6 Assumptions and Dependencies

<List any assumed factors (as opposed to known facts) that could affect the requirements stated in the SRS. These could include third-party or commercial components that you plan to use, issues around the development or operating environment, or constraints. The project could be affected if these assumptions are incorrect, are not shared, or change. Also identify any dependencies the project has on external factors, such as software components that you intend to reuse from another project, unless they are already documented elsewhere (for example, in the vision and scope document or the project plan).>

2.8.USER DOCUMENTATION

For user manuals and help use help option in the main menu or visit http://www.snapchat.com/ and click on contact us option.

3. External Interface Requirements

3.1 User Interfaces

<Describe the logical characteristics of each interface between the software product and the users. This may include sample screen images, any GUI standards or product family style guides that are to be followed, screen layout constraints, standard buttons and functions (e.g., help) that will appear on every screen, keyboard shortcuts, error message display standards, and so on. Define the software components for which a user interface is needed. Details of the user interface design should be documented in a separate user interface specification.>

3.2 Hardware Interfaces

<Describe the logical and physical characteristics of each interface between the software product and the hardware components of the system. This may include the supported device types, the nature of the data and control interactions between the software and the hardware, and communication protocols to be used.>

3.3 Software Interfaces

<Describe the connections between this product and other specific software components (name and version), including databases, operating systems, tools, libraries, and integrated commercial components. Identify the data items or messages coming into the system and going out and describe the purpose of each. Describe the services needed and the nature of communications. Refer to documents that describe detailed application programming interface protocols. Identify data that will be shared across software components. If the data sharing mechanism must be implemented in a specific way (for example, use of a global data area in a multitasking operating system), specify this as an implementation constraint.>

3.4 Communications Interfaces

<Describe the requirements associated with any communications functions required by this product, including e-mail, web browser, network server communications protocols, electronic forms, and so on. Define any pertinent message formatting. Identify any communication standards that will be used, such as FTP or HTTP. Specify any communication security or encryption issues, data transfer rates, and synchronization mechanisms.

Formal System Specification:

Formal methods provide us with tools to precisely describe a system and show that a system is correctly implemented. We say a system is correctly implemented when it satisfies its given specification.

What is a Formal Technique?

A formal technique is a mathematical method to specify a hardware and/or software system,verify whether a specification is realisable, verify that an implementation satisfies its specification, prove properties of a system without necessarily running the system, etc.

More precisely, a formal specification language consists of two sets—syn and sem, and a relation sat between them. The set **syn** is called the syntactic domain, the set sem is called the semantic domain, and the relation sat is called the satisfaction relation.

Syntactic domains

The syntactic domain of a formal specification language consists of an alphabet of symbols and a set of formation rules to construct well- formed formulas from the alphabet. The well formed formulas are used to specify a system.

Semantic domains

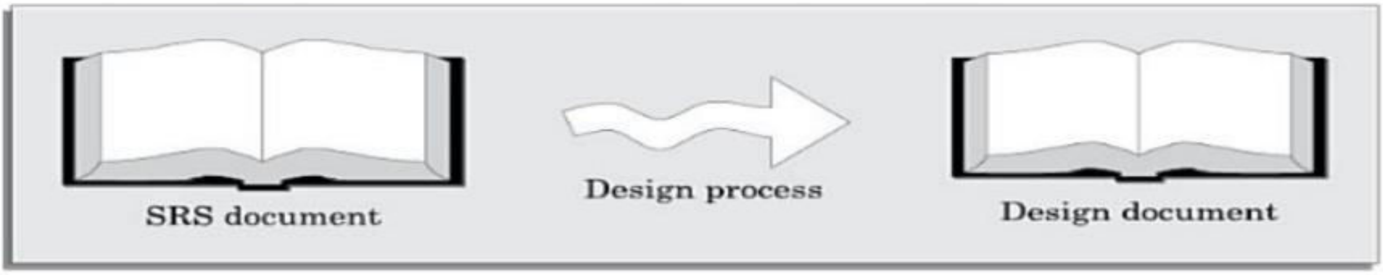
Formal techniques can have considerably different semantic domains. Abstract data type specification languages are used to specify algebras, theories, and programs. Programming languages are used to specify functions from input to output values. Concurrent and distributed system specification languages are used to specify state sequences, event sequences, state-transition sequences, synchronisation trees, partial orders, state machines, etc.

Satisfaction relation

It is important to determine whether an element of the semantic domain satisfies the specifications. This satisfaction is determined by using a homomorphism known as semantic abstract on function. The semantic abstraction function maps the elements of the semantic domain into equivalent classes.

SOFTWARE DESIGN

The activities carried out during the design phase (called as design process)
transform the SRS document into the design document.



Levels of design/ types of design:

1) Architectural design
2) Physical/high-level/macro level design
3) Detailed/ low-level/micro level design

1.Architectural Design - This is the first level of the designing. The architectural design is the highest abstract version of the system. It identifies the software as a system with many components
interacting with each other. At this level, the designers get the idea of proposed solution domain.

2. High-level Design:- This is the second level of the designing. The high-level design breaks the 'single entity-multiple component' concept of architectural design into less-abstracted view of subsystems and modules and depicts their interaction with each other. High-level design focuses on how the system along with all of its components can be implemented in forms of modules. It recognizes modular structure of each sub-system and their relation and interaction among each other

3. Detailed Design-This is the third level of the designing. Detailed design deals with the implementation part of what is seen as a system and its sub-systems in the previous two designs. It is more detailed towards modules and their implementations. It defines logical structure of each module and their interfaces to communicate with other modules.

Characteristics of a good software design:

a) Correctness: The software which we are making should meet all the specifications stated by the customer.

b) Usability/Learnability: The amount of efforts or time required to learn how to use the software should be less. This makes the software user-friendly even for illiterate people.

c) Integrity: Just like medicines have side-effects, in the same way a software may have a sideeffect i.e. it may affect the working of another application. But a quality software should not have side effects.

d) Reliability: The software product should not have any defects. Not only this, it shouldn't fail while execution.

e) Efficiency: This characteristic relates to the way software uses the available resources. The software should make effective use of the storage space and execute command as per desired timing
requirements.

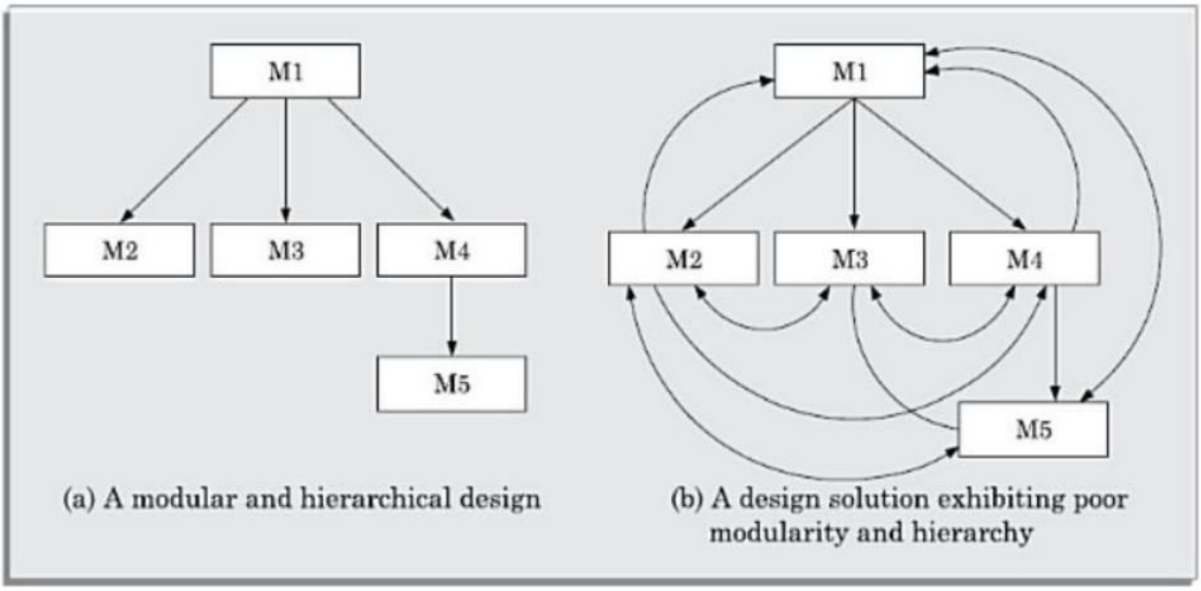
f) Security: With the increase in security threats nowadays, this factor is gaining importance. The software shouldn't have ill effects on data / hardware. Proper measures should be taken to keep data secure from external threats.

g) Safety: The software should not be hazardous to the environment/life.

Understandability of a Design: A Major Concern

It should assign consistent and meaningful names to various design components.

It should make use of the principles of decomposition and abstraction in good measures to simplify the design.
A design solution should be modular and layered to be understandable.



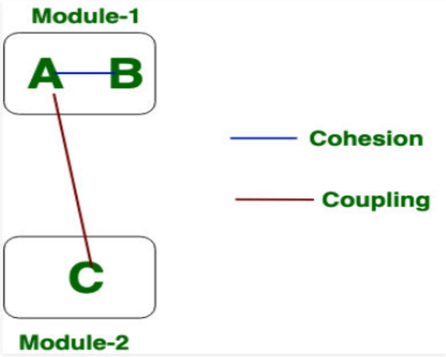
(a) A modular and hierarchical design

(b) A design solution exhibiting poor modularity and hierarchy

---

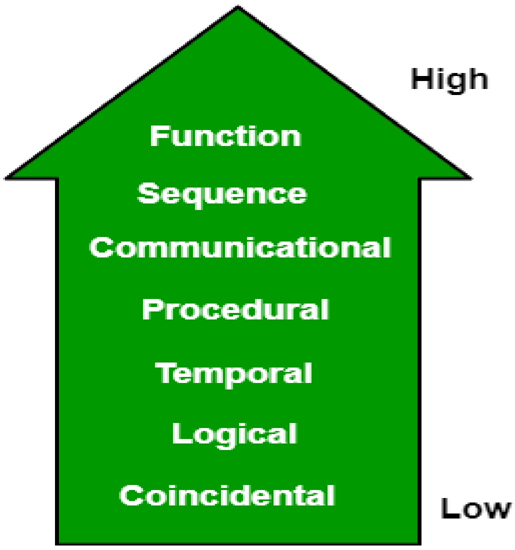
COHESION

Cohesion is a measure of the functional strength of a module
Coupling between two modules is a measure of the degree of interaction (or interdependence) between the two modules.

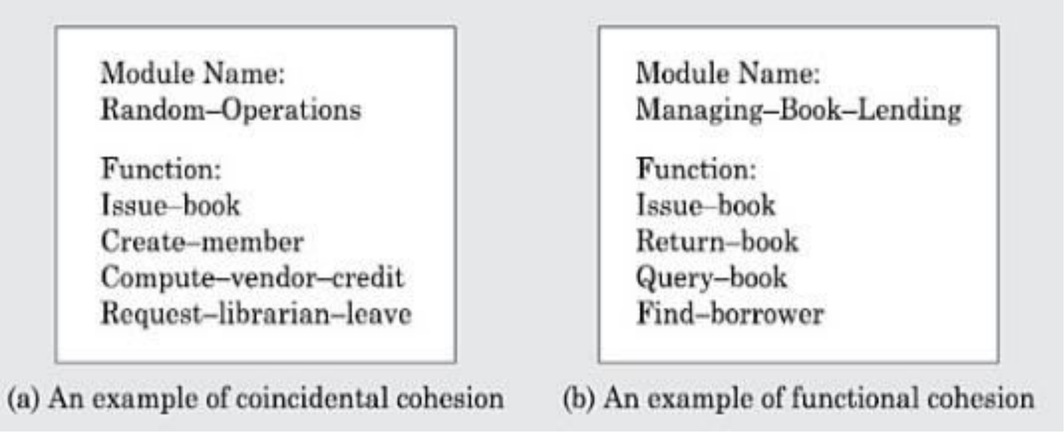


Increasing cohesion is good for software.
Increasing coupling is avoided for software.

Types of Cohesion



Functional Cohesion: Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.
Sequential Cohesion: An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
Communicational Cohesion: Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer.
Procedural Cohesion: Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, and print cumulative GPA.
Temporal Cohesion: The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time.
Logical Cohesion: The elements are logically related and not functionally.
Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.
Coincidental Cohesion: The elements are not related (unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion.
Ex- print next line and reverse the characters of a string in a single component.



Module Name:
Random–Operations

Function:
Issue–book
Create–member
Compute–vendor–credit
Request–librarian–leave

(a) An example of coincidental cohesion

Module Name:
Managing–Book–Lending

Function:
Issue–book
Return–book
Query–book
Find–borrower

(b) An example of functional cohesion

Advantages of high cohesion:
● Improved readability and understandability: making it easier for developers to

understand the code and make changes.

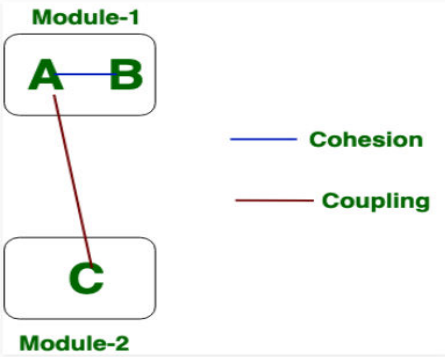
- Better error isolation: High cohesion reduces the likelihood that a change in one part of a module will affect other parts.
- Fix errors easily: High cohesion leads to modules that are less prone to errors and that function more consistently,
- It leads to an overall improvement in the reliability of the system.

Disadvantages of low cohesion:

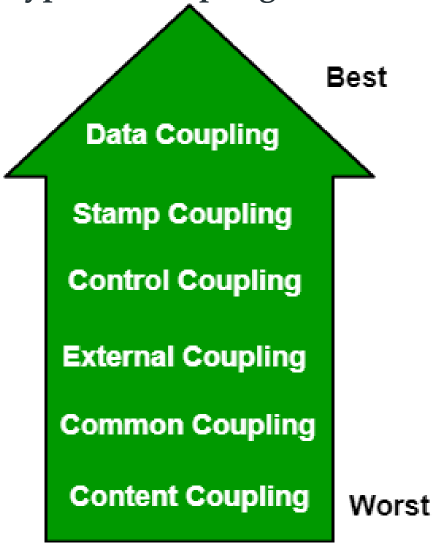
- Increased code duplication
- Reduced functionality
- Difficulty in understanding the module: Low cohesion can make it harder for developers to understand the purpose and behavior of a module, leading to errors and a lack of clarity.

Coupling:

Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.



Types of Coupling:



- **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent of each other and communicate through data. Module communications don't contain tramp data. Example-customer billing system.

- **Stamp Coupling** In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice was made by the insightful designer, not a lazy programmer.
- **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.
- **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.
- **Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has disadvantages like difficulty in reusing modules, reduced ability to control data accesses, and reduced maintainability.
- **Content Coupling:** In a content coupling, one module can modify the data of another module, or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

Advantages of low coupling:

- Improved maintainability
- Enhanced modularity
- Better scalability: Low coupling facilitates the addition of new modules and the removal of existing ones, making it easier to scale the system as needed.

Disadvantages of high coupling:

- Increased complexity
- Reduced flexibility: High coupling makes it more difficult to modify or replace individual components without affecting the entire system.
- Decreased modularity: reducing reusability of code.

Layered Arrangement of Modules

An important characteristic feature of a good design solution is layering of the modules. A layered software design is one in which when the relationships among different modules are represented graphically, it would result in a tree-like diagram with clear layering. In the layered design, the modules are arranged in the hierarchy of layers.

In such a design, a module can only invoke functions of the modules in the layer immediately below it. The higher layer module can be considered similar to management who can invoke a lower layer module to get a certain task done.

Layered arrangement of modules makes the work easily understandable, since the layered design easily reflects the relationships among different layers and clears which layer invokes the other layers. This is also helpful in debugging in case of error. When failure is detected then it is obvious that only the lower layer can possibly be the source of the error.

Top most module is called as manager

- Modules in intermediate layer
  - Provide services to their higher layer

- Invokes services of lower layer

Adavantages:

• Understandability
• Error isolated
• Failure of one module do not affect others

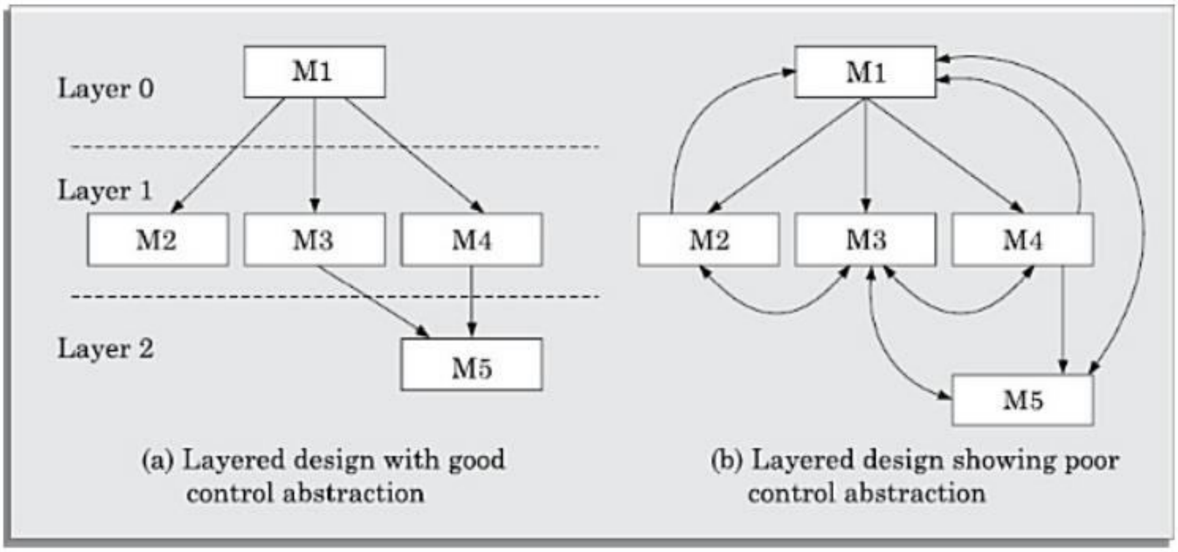

Layer 0
Layer 1
Layer 2
M1
M2
M3
M4
M5
(a) Layered design with good control abstraction
(b) Layered design showing poor control abstraction

Figure 5.6: Examples of good and poor control abstraction.

In the following, we discuss some important concepts and terminologies associated with a layered design:

Superordinate and subordinate modules:

In a control hierarchy, a module that controls another module is said to be superordinate to it. Conversely, a module controlled by another module is said to be subordinate to the controller.

Visibility:

A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visible to a module.

Control abstraction:

In a layered design, a module should only invoke the functions of the modules that are in the layer immediately below it. In other words, the modules at the higher layers, should not be visible (that is, abstracted out) to the modules at the lower layers. This is referred to as control abstraction.

Depth and width:

Depth and width of a control hierarchy provide an indication of the number of layers and the overall span of control respectively. For the design of Figure 5.6(a), the depth is 3 and width is also 3.

Fan-out:

Fan-out is a measure of the number of modules that are directly controlled by a given module.

In Figure 5.6(a), the fan-out of the module M1 is 3. A design in which the modules have very high fan-out numbers is not a good design. The reason for this is that a very high fan-out is an indication that the module lacks cohesion. A module having a large fan-out (greater than 7) is likely to implement several different functions and not just a single cohesive function.

Fan-in:

Fan-in indicates the number of modules that directly invoke a given module. High fan-in represents code reuse and is in general, desirable in a good design.

In Figure 5.6(a), the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.

Approaches to Software Design

Function-oriented Design	Object-oriented Design
1.Function oriented design is the result of focusing attention to the function of the program	1.Object oriented design is the result of focusing attention to the objects,classes of the program
2.It follows Top Down Approach	2.It follows Bottom-up Approach
3.It is represented by Data Flow Diagram	3.It is represented by Class diagram-UML
4.In this approach the state information is often represented in a centralized shared memory.	4.In this approach the state information is not represented is not represented in a centralized memory but is implemented or distributed among the objects of the system.
5.In function oriented design we decompose in function/procedure level	5.In Object-oriented Design ,We decompose in class level

Data Flow Diagram (DFD)

Data Flow Diagram (DFD) of a system represents how input data is converted to output data graphically. Level 0 also called context level represents most fundamental and abstract view of the system. Subsequently other lower levels can be decomposed from it. DFD model of a system contains multiple DFDs but there is a single data dictionary for entire DFD model. Data dictionary comprises definitions of data items used in DFD.

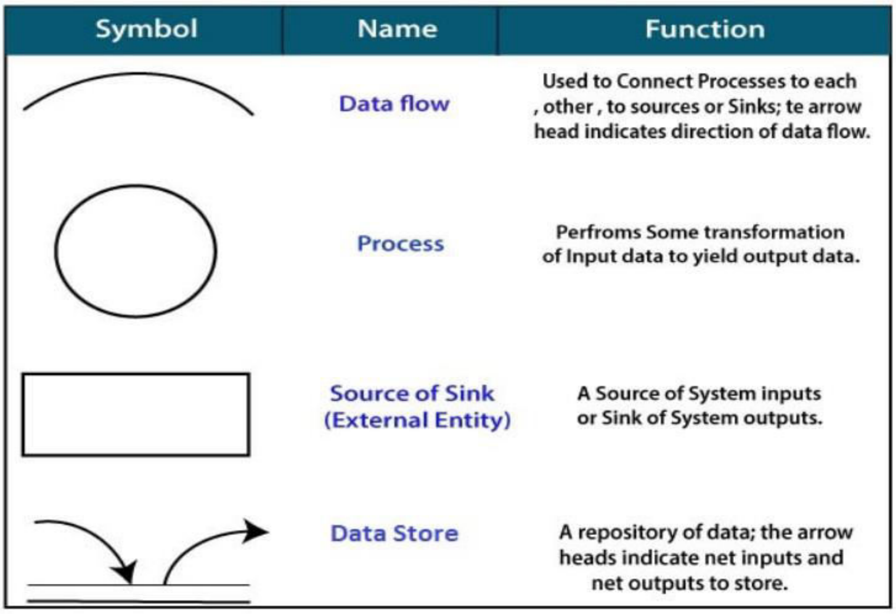
It shows how data enters and leaves the system, what changes the information, and where data is stored.

Levels in Data Flow Diagrams (DFD)

0-level DFDM

1-level DFD

2-level DFD



Symbol	Name	Function
	Data flow	Used to Connect Processes to each , other , to sources or Sinks; te arrow head indicates direction of data flow.
	Process	Perfroms Some transformation of Input data to yield output data.
	Source of Sink (External Entity)	A Source of System inputs or Sink of System outputs.
	Data Store	A repository of data; the arrow heads indicate net inputs and net outputs to store.

0-level DFDM

It is also known as the fundamental system model, or context diagram that represents the entire software requirement as a single bubble with input and output data denoted by incoming and outgoing arrows.

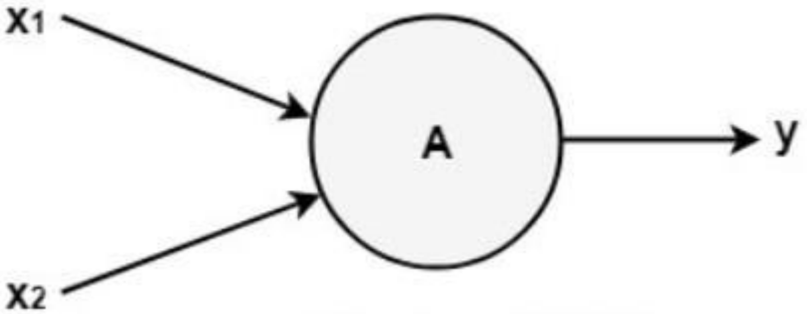
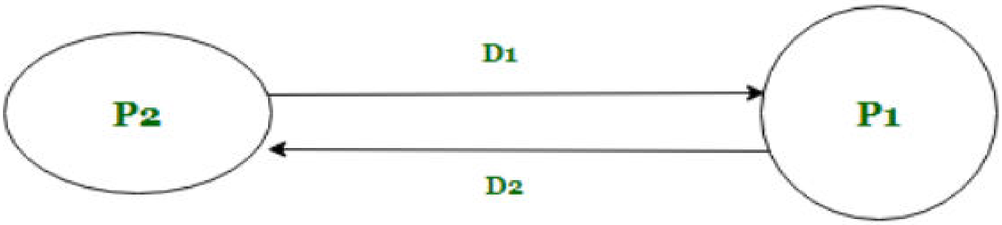


Fig: Level-0 DFD.

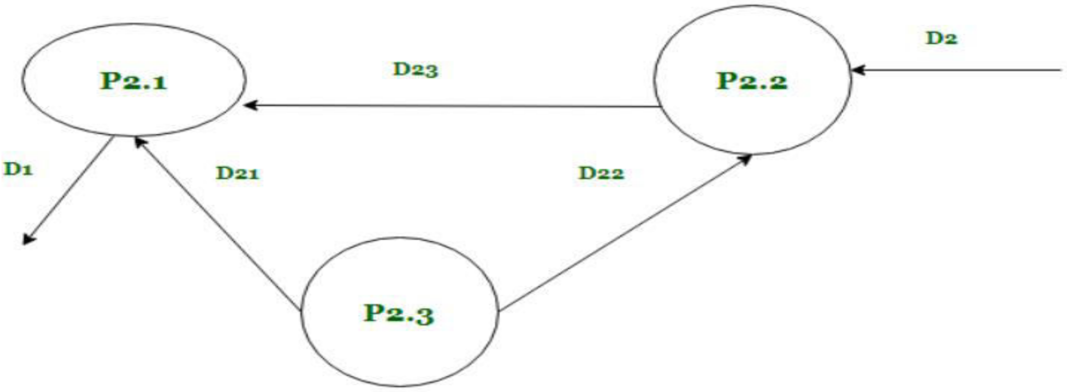
1-level DFDM

In 1-level DFD, a context diagram is decomposed into multiple bubbles/processes.



2-level DFD goes one process deeper into parts of 1-level DFD. It can be used to project or record the specific/necessary detail about the system's functioning.

2-Level DFD



Numbering :

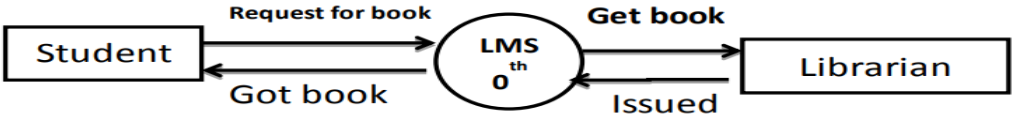
Each process symbol must utilize a unique reference number to differentiate from one other. When a bubble x is decomposed, its children are numbered as x.1, x.2 and so on. It can help determine its ancestors, successors, and precisely its level. For example level 0 DFD is numbered as 0. Level 1 are numbered as 0.1, 0.2, 0.3 or 1, 2, 3 and so on. Level 2 are marked as 1.1, 1.2, 1.3 etc.

Common mistakes to avoid while constructing DFDs :

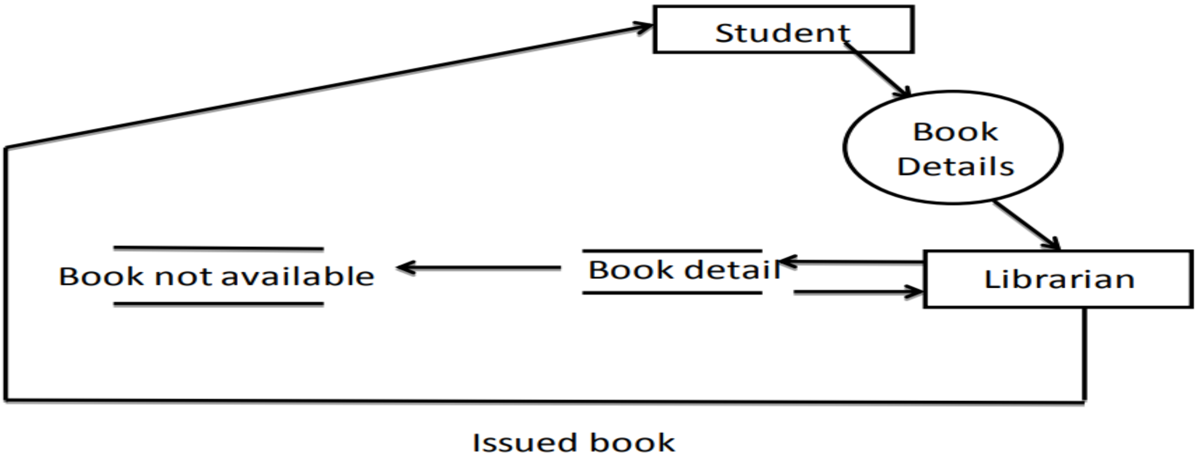
- DFDs should always represent data flow and there should be no control flow.
- All external entities should be represented at context level.
- All functionality of system must be captured in dfd and none should be overlooked. Also, only those functions specified in SRS should be represented.
- Arrows connecting to data store need not be annotated with any data name.

Example: create a DFD for library management system

i) 0th level



ii) 1st level



UNIT – III:

Unified Modeling Language (UML): Introduction to UML, why we model, Standard Diagrams: Structural Diagrams- Class diagram, Object diagram, Component diagram, Deployment diagram, Behavioural Diagrams- Use case diagram, Sequence diagram, Collaboration diagram, State chart diagram, Activity diagram.

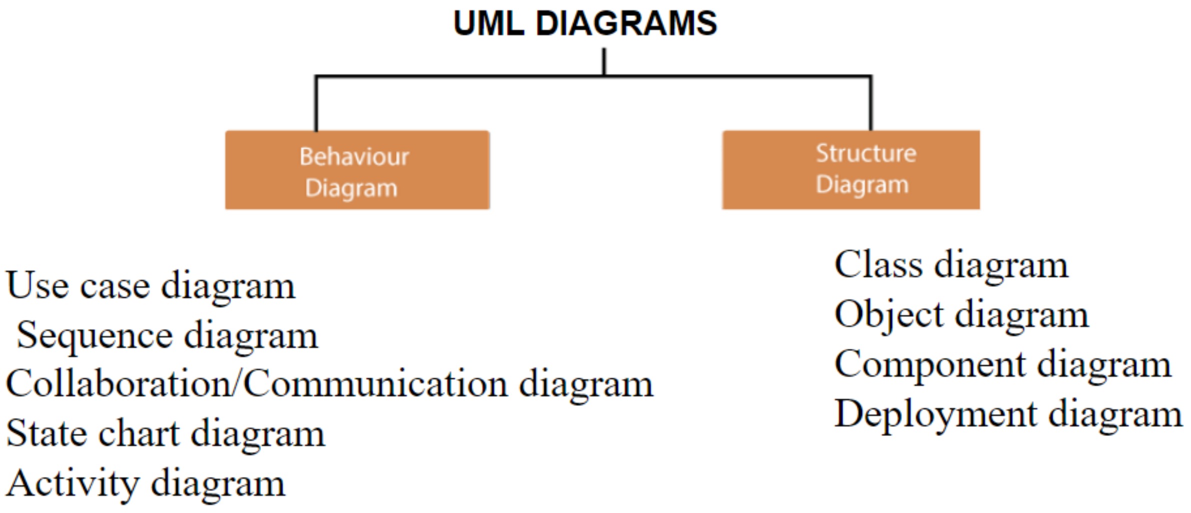
Introduction to UML

- UML (Unified Modeling Language) is a general-purpose, graphical modeling language in the field of Software Engineering.
- UML is used to specify, visualize, construct, and document the artifacts (major elements) of the software system
- It was initially developed by Grady was carried out through 1996. In 1997, it got adopted as a standard by the Object Management Group. Booch, Ivar Jacobson, and James Rumbaugh in 1994-95 at Rational software, and its further development
- It is quite similar to blueprints used in other fields of engineering.
- UML is not a programming language, it is rather a visual language. We use UML diagrams to portray the behavior and structure of a system. UML helps software engineers, businessmen and system architects with modelling, design and analysis. The Object Management Group (OMG) adopted Unified Modelling Language as a standard in 1997. Its been managed by OMG ever since. International Organization for Standardization (ISO) published UML as an approved standard in 2005. UML has been revised over the years and is reviewed periodically.

The UML has the following features:

- It is a generalized modeling language.
- It is distinct from other programming languages like C++, Python, etc.
- It is interrelated to object-oriented analysis and design.
- It is used to visualize the workflow of the system.
- It is a pictorial language, used to generate powerful modeling artifacts.

UML is linked with object oriented design and analysis. UML makes the use of elements and forms associations between them to form diagrams. Diagrams in UML can be broadly classified as:



Structural Diagrams – Capture static aspects or structure of a system. Structural Diagrams include: Class Diagrams ,Object Diagrams, Component Diagrams, and Deployment Diagrams.

Behavior Diagrams – Capture dynamic aspects or behavior of the system. Behavior diagrams include: Use Case Diagrams, State Diagrams, Activity Diagrams and Interaction Diagrams(communication diagram and Sequence Diagram).

Class Diagram – The most widely use UML diagram is the class diagram. It is the building block of all object oriented software systems. We use class diagrams to depict the static structure of a system by showing system's classes,their methods and attributes. Class diagrams also help us identify relationship between different classes or objects.

Object Diagram – An Object Diagram can be referred to as a screenshot of the instances in a system and the relationship that exists between them. Since object diagrams depict behaviour when objects have been instantiated, we are able to study the behaviour of the system at a particular instant. An object diagram is similar to a class diagram except it shows the instances of classes in the system. We depict actual classifiers and their relationships making the use of class diagrams. On the other hand, an Object Diagram represents specific instances of classes and relationships between them at a point of time.

Component Diagram – Component diagrams are used to represent how the physical components in a system have been organized. We use them for modelling implementation details. Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development. Component Diagrams become essential to use when we design and build complex systems. Interfaces are used by components of the system to communicate with each other.

Deployment Diagram – Deployment Diagrams are used to represent system hardware and its software.It tells us what hardware components exist and what software components run on them.We illustrate system architecture as distribution of software artifacts over distributed targets. An artifact is the information that is generated by system software. They are primarily used when software is being used, distributed or deployed over multiple machines with different configurations.

Behavior Diagrams –

State Machine Diagrams – A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It's a behavioral diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as State machines and State-chart Diagrams . These terms are often used interchangeably.So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli.

Activity Diagrams – We use Activity Diagrams to illustrate the flow of control in a system. We can also use an activity diagram to refer to the steps involved in the execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram.An activity diagram focuses on condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.

Use Case Diagrams – Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors). A use case is basically a diagram representing different scenarios where the system can be used. A use case diagram gives us a high level view of what the system or a part of the system does without going into implementation details.

Sequence Diagram – A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place.We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.

Communication Diagram – A Communication Diagram(known as Collaboration Diagram) is used to show sequenced messages exchanged between objects. A communication diagram focuses primarily on objects and their relationships. We can represent similar information using Sequence diagrams,however, communication diagrams represent objects and links in a free form.

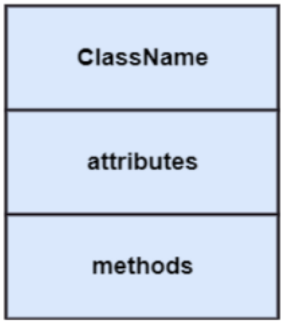
Class diagram

It represents the types of objects residing in the system and the relationships between them.
A class consists of its objects, and also it may inherit from other classes.
It analyses and designs a static view of an application

Benefits/uses of Class Diagrams

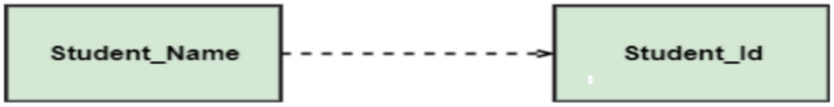
- It can represent the object model for complex systems.
- It reduces the maintenance time by providing an overview of how an application is structured before coding
- Used for better understanding.
- It is helpful for the stakeholders and the developers.
- This is the only UML that can appropriately depict various aspects of the OOPs concept.
- Proper design and analysis of applications can be faster and efficient.
- It is the base for deployment and component diagram.

Components of a Class Diagram

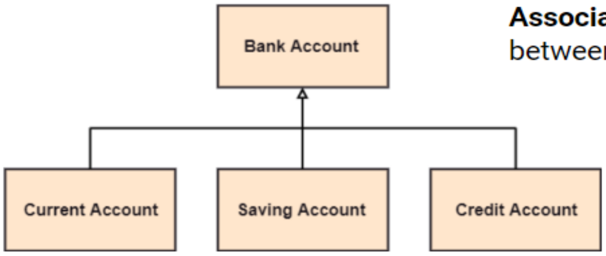


Relationships

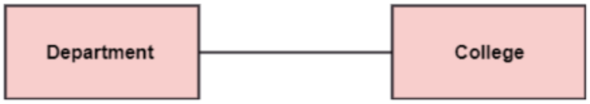
Dependency:change in one class cause changes in another class



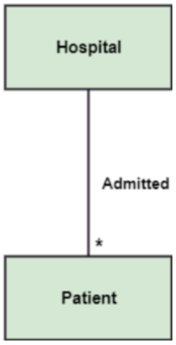
Generalization: relationship between a parent class (superclass) and a child class (subclass).



Association: It describes a static or physical connection between two or more objects.

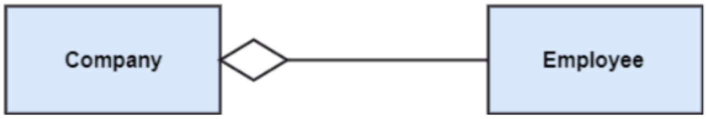


Multiplicity:



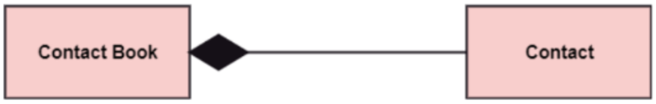
Aggregation: subset of association, which represents has a relationship

even if one employee resigns, the company still exists

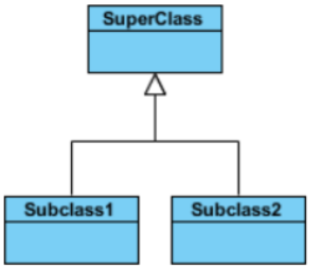


Composition: subset of aggregation.

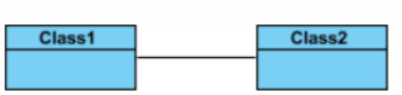
if one part is deleted, then the other part also gets discarded.



+ denotes public attributes or operations
- denotes private attributes or operations
# denotes protected attributes or operations
~ denotes package attributes or operations

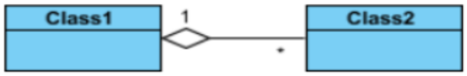


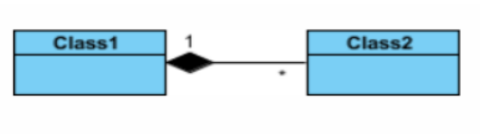
Inheritance (or Generalization):



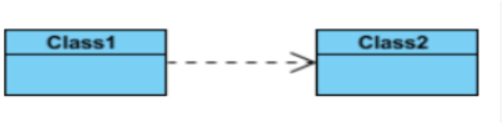
Simple Association

Aggregation



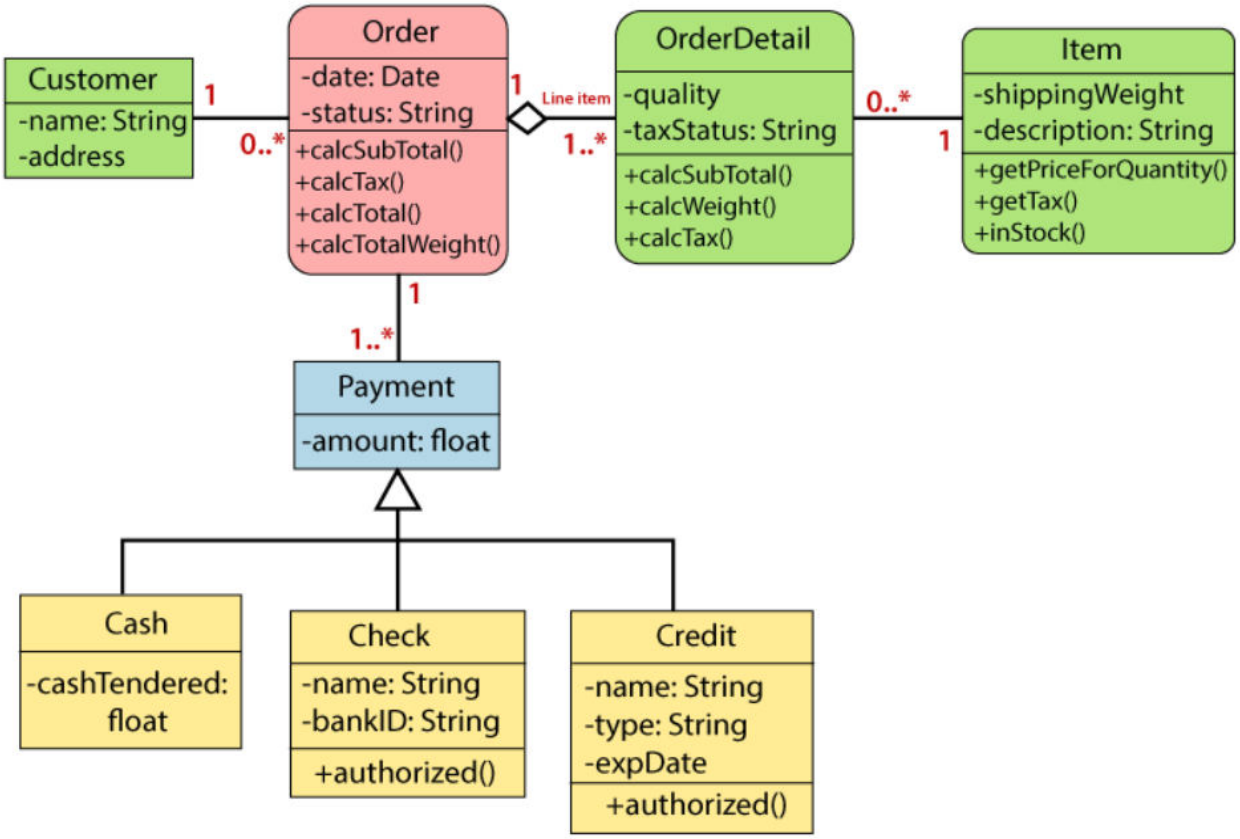


Composition



Dependency:

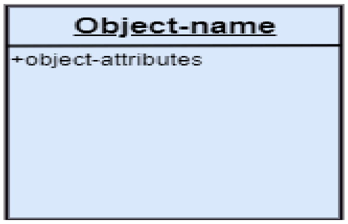
Class Diagram for Online Shopping



Object Diagram

Object diagrams are dependent on the class diagram as they are derived from the class diagram. It represents an instance of a class diagram.

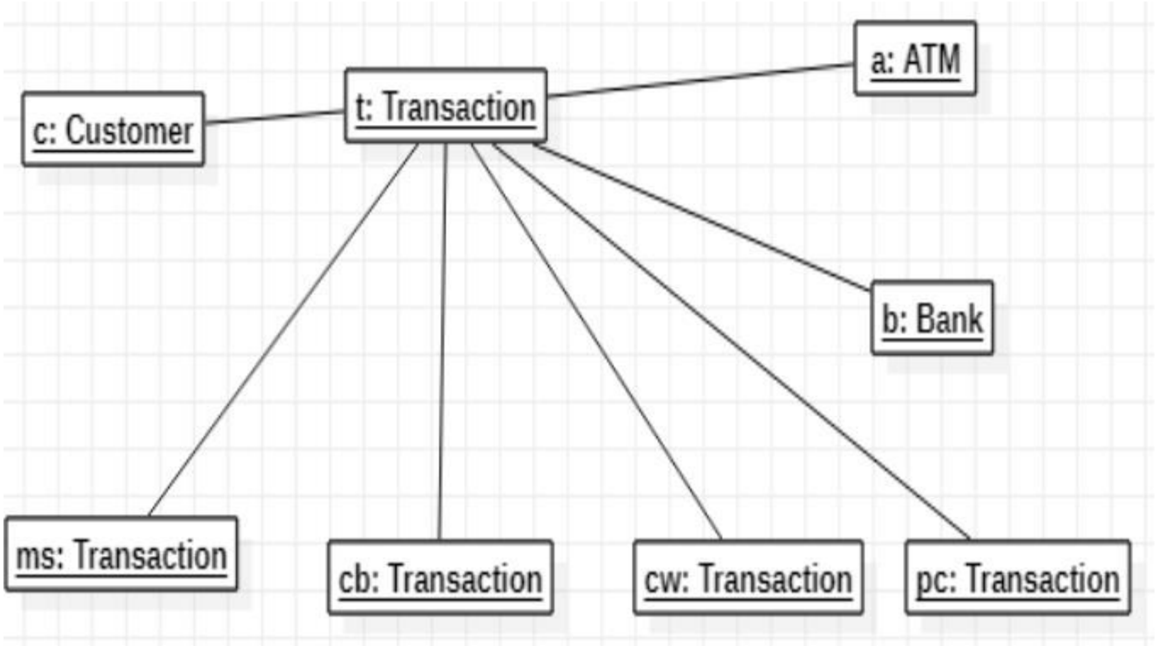
Representation



Uses of Object Diagram

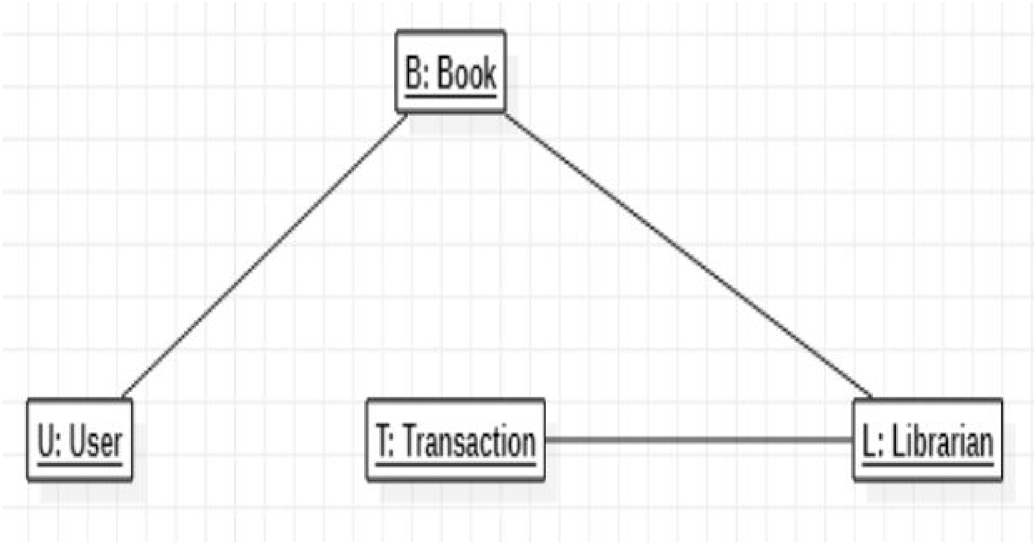
- It is used to perform forward and reverse engineering.
- It is used to understand object behavior and their relationship
- Dynamic changes are captured in the object diagram.
- It is used to represent an instance of a system.
- It incorporates data values and attributes of an entity.

Object Diagram ATM Transactions



ms:Mini Statement
cb:Check Balance
cw:Cash Withdrawl
pc:Password change

Object Diagram Library Management

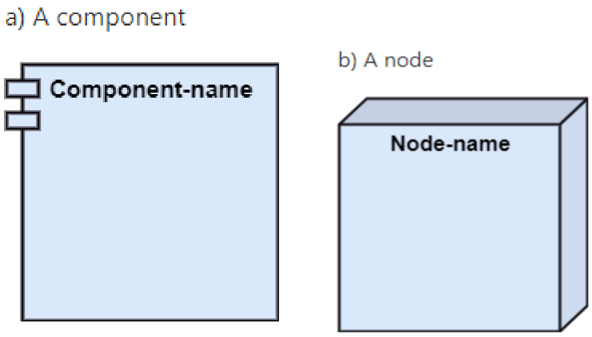


Component Diagram

A component diagram is used to break down a large object-oriented system into the smaller components, so as to make them more manageable.

Component diagrams are used to represent how the physical components in a system have been organized. We use them for modelling implementation details. Component Diagrams depict the structural relationship between software system elements and help us in understanding if functional requirements have been covered by planned development. Component Diagrams become essential to use when we design and build complex systems. Interfaces are used by components of the system to communicate with each other.

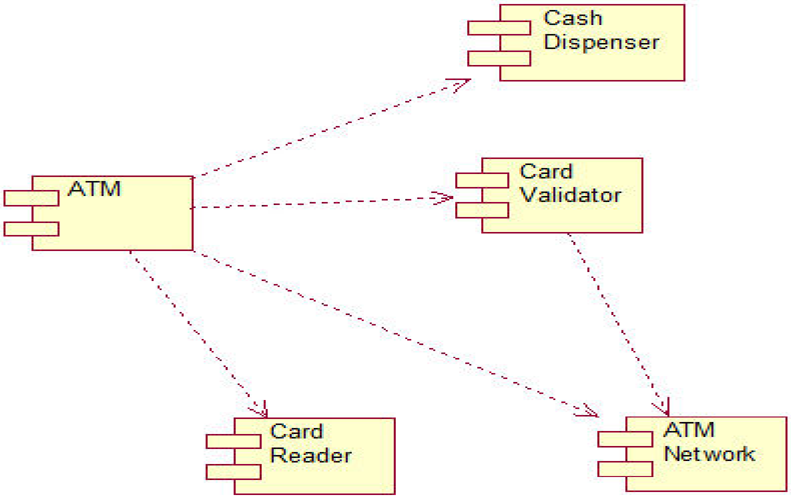
Representation of a Component Diagram



Uses

- To divide a single system into multiple components according to the functionality.
- To represent the component organization of the system.

Component Diagram for ATM Transactions

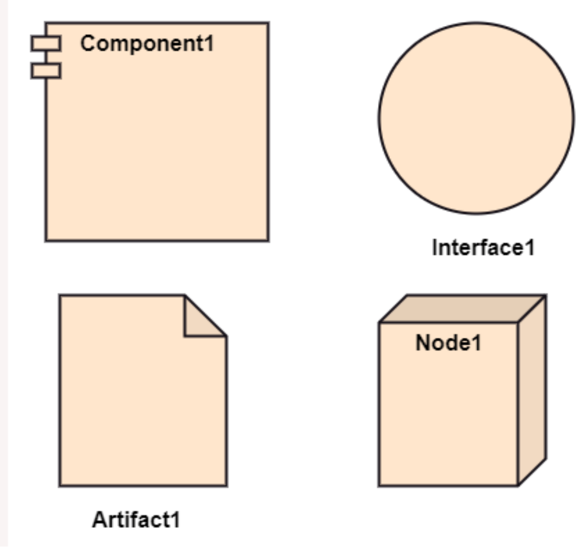


Deployment Diagram

The deployment diagram visualizes the physical hardware on which the software will be deployed. It portrays the static deployment view of a system. It involves the nodes and their relationships.

Deployment Diagrams are used to represent system hardware and its software.It tells us what hardware components exist and what software components run on them.We illustrate system architecture as distribution of software artifacts over distributed targets. An artifact is the information that is generated by system software. They are primarily used when software is being used, distributed or deployed over multiple machines with different configurations.

Representation

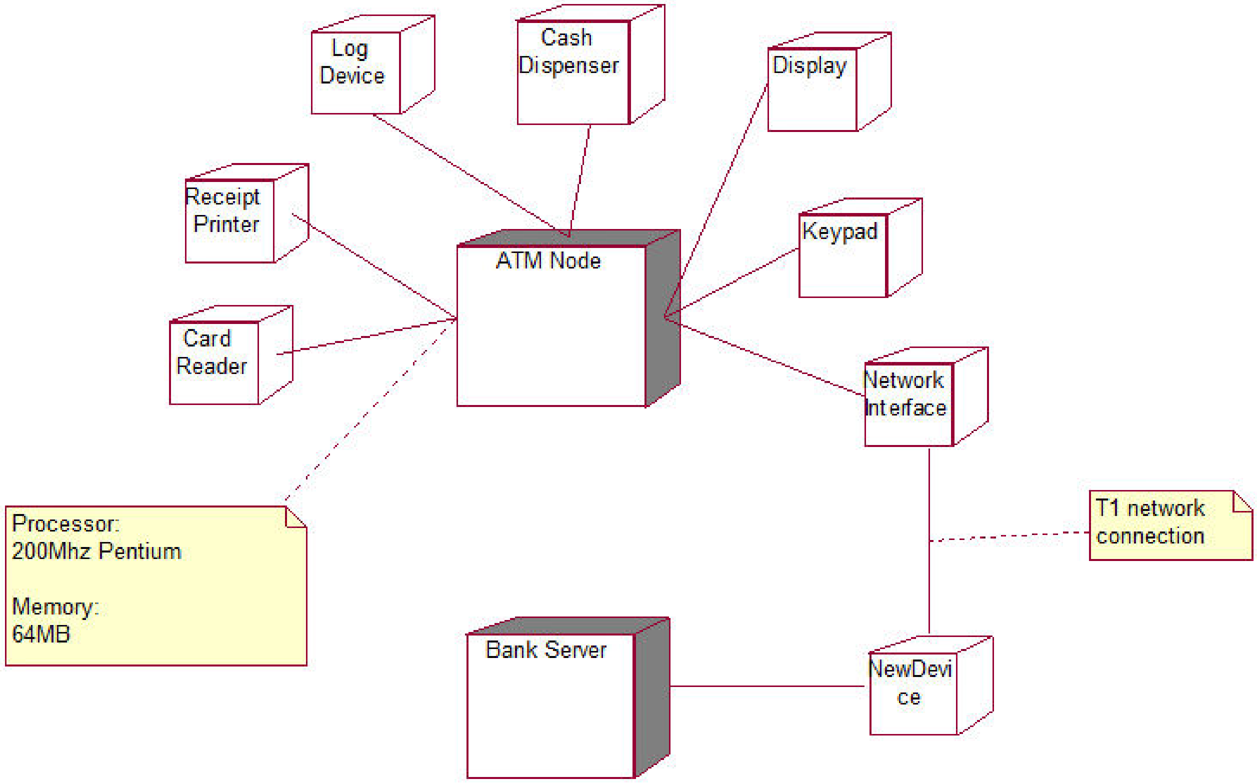


Deployment diagrams can be used for the followings:

- To model the network and hardware topology of a system.

- To model the distributed networks and systems.
- Implement forwarding and reverse engineering processes.
- To model the hardware details for a client/server system.
- For modeling the embedded system.

Deployment diagram for ATM Transaction



UNIT-IV

Coding

The coding is the process of transforming the design of a system into a computer language format. This coding phase of software development is concerned with software translating design specification into the source code. It is necessary to write source code & internal documentation so that conformance of the code to its specification can be easily verified.

Coding is done by the coder or programmers who are independent people than the designer. The goal is not to reduce the effort and cost of the coding phase, but to cut to the cost of a later stage. The cost of testing and maintenance can be significantly reduced with efficient coding.

Goals of Coding

To translate the design of system into a computer language format:

The coding is the process of transforming the design of a system into a computer language format, which can be executed by a computer and that perform tasks as specified by the design of operation during the design phase.

To reduce the cost of later phases:

The cost of testing and maintenance can be significantly reduced with efficient coding.

Making the program more readable:

Program should be easy to read and understand. It increases code understanding having readability and understandability as a clear objective of the coding activity can itself help in producing more maintainable software.

The Coding Standards we have to follow while coding are as follows.

- Indentation
- Inline Comments
- Use of Global
- Structured Programming

- Naming Conventions
- Errors & Exception Handling

1. **Indentation:** Proper and consistent indentation is essential in producing easy to read and maintainable programs.
   Indentation should be used to:
   - Emphasize the body of a control structure such as a loop or a select statement.
   - Emphasize the body of a conditional statement
   - Emphasize a new scope block
2. **Inline comments:** Inline comments analyze the functioning of the subroutine, or key aspects of the algorithm shall be frequently used.
3. **Rules for limiting the use of global:** These rules file what types of data can be declared global and what cannot.
4. **Structured Programming:** Structured (or Modular) Programming methods shall be used. "GOTO" statements shall not be used as they lead to "spaghetti" code, which is hard to read and maintain, except as outlined line in the FORTRAN Standards and Guidelines.
5. **Naming conventions for global variables, local variables, and constant identifiers:** A possible naming convention can be that global variable names always begin with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
6. **Error return conventions and exception handling system:** Different functions in a program report the way error conditions are handled should be standard within an organization. For example, different tasks while encountering an error condition should either return a 0 or 1 consistently.

Code Review:

Code Review is a systematic examination, which can find and remove the vulnerabilities in the code such as memory leaks and buffer overflows.

- Technical reviews are well documented and use a well-defined defect detection process that includes peers and technical experts.
- This kind of review is usually performed as a peer review without management participation.
- Reviewers prepare for the review meeting and prepare a review report with a list of findings.
- Technical reviews may be quite informal or very formal and can have a number of purposes but not limited to discussion, decision making, and evaluation of alternatives, finding defects and solving technical problems.

Code reviews are quality assurance measures conducted to examine a developer’s code in relation to several objectives. The primary aim of code reviews is to find code defects, but also to verify compliance with QA standards as relates to logic, structure, style, and readability. Code reviews are also indispensable to cultivating teamwork, knowledge sharing, finding new solutions, and ultimately increasing code quality. Software developers and engineers consider code reviews as the number one means of improving code quality, followed by unit testing and continuous integration

Most probably the reviews are as follows

- Informal
- Walkthrough
- Technical Review
- Inspection

Informal Review:

Unlike Formal Reviews, Informal reviews are applied multiple times during the early stages of software development process. The major difference between the formal and informal reviews is that the former follows a formal agenda, whereas the latter is conducted as per the need of the team and follows an informal agenda.

Walkthrough:

It is used to review documents with peers, managers, and fellow team members who are guided by the author of the document to gather feedback and reach a consensus.

Technical Review:

A technical review is the primary method for communicating progress, coordinating tasks, monitoring risk, and transferring products and knowledge between the team members of a project.

Inspection:

Inspections are a formal type of review that involves checking the documents thoroughly before a meeting and is carried out mostly by moderators. A meeting is then held to review the code and the design. Inspection meetings can be held both physically and virtually.

Software Documentation

Software documentation is a written piece of text that is often accompanied by a software program.

This makes the life of all the members associated with the project easier. It may contain anything from API documentation, build notes or just help content. It is a very critical process in software development.

It's primarily an integral part of any computer code development method. Moreover, computer code practitioners are a unit typically concerned with the worth, degree of usage, and quality of the actual documentation throughout the development and its maintenance throughout the total method.

For example, before the developments of any software product requirements are documented which is called Software Requirement Specification (SRS).

Requirement gathering is considered a stage of Software Development Life Cycle (SDLC).

Another example can be a user manual that a user refers to for installing, using, and providing maintenance to the software application/product.

Types of Software Documentation:

1. **Requirement Documentation:** It is the description of how the software shall perform and which environment setup would be appropriate to have the best out of it. These are generated while the software is under development and is supplied to the tester groups too.

2. **Architectural Documentation:** Architecture documentation is a special type of documentation that concerns the design. It contains very little code and is more focused on the components of the system, their roles, and working. It also shows the data flow throughout the system.

3. **Technical Documentation:** These contain the technical aspects of the software like API, algorithms, etc. It is prepared mostly for software developers.

4. **End-user Documentation:** As the name suggests these are made for the end user. It contains support resources for the end user.

Advantages of software documentation:

- The presence of documentation helps in keeping the track of all aspects of an application and also improves the quality of the software product.

- The main focus is based on the development, maintenance, and knowledge transfer to other developers.

- Helps development teams during development.

- Helps end-users in using the product.
- Improves overall quality of software product
- It cuts down duplicative work.
- Makes easier to understand code.
- Helps in establishing internal coordination in work.

Disadvantages of software documentation:

- The documenting code is time-consuming.
- The software development process often takes place under time pressure, due to which many times the documentation updates don't match the updated code.
- The documentation has no influence on the performance of an application.
- Documenting is not so fun; it's sometimes boring to a certain extent.

Testing:

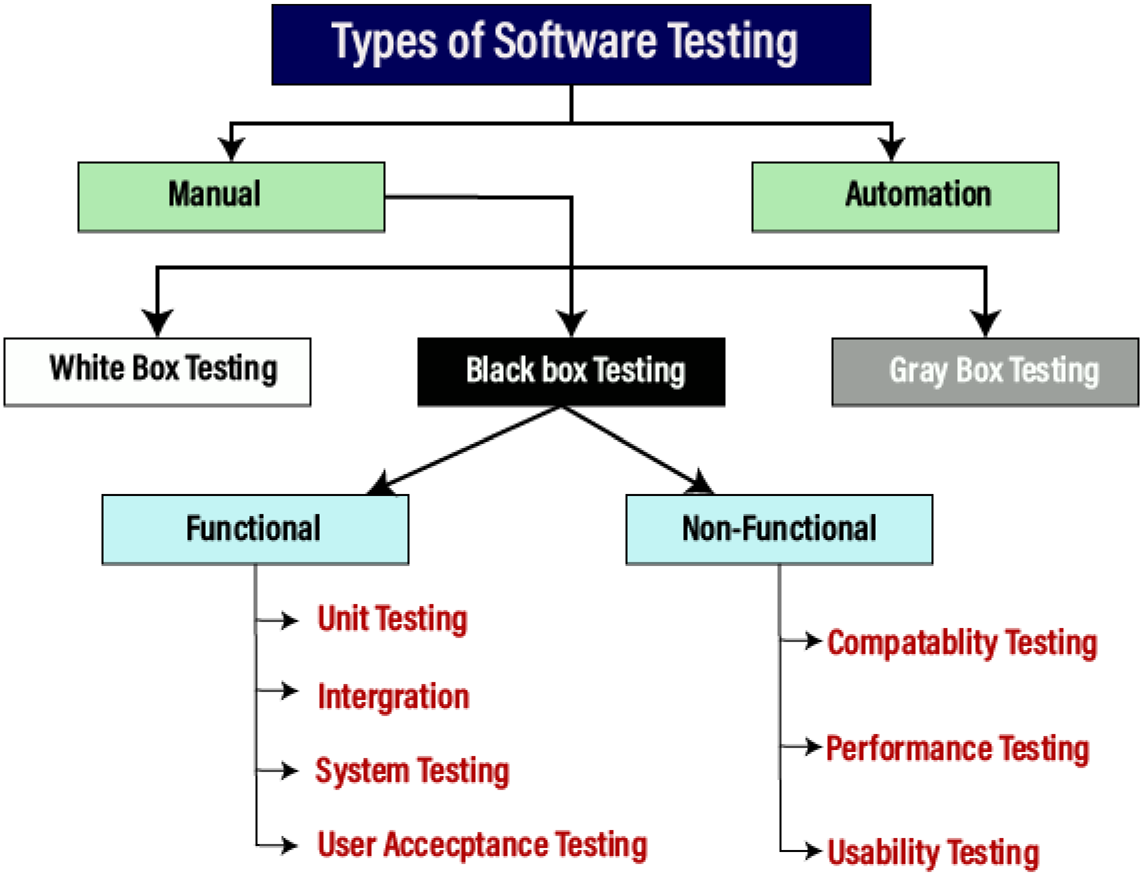
Software testing is a process of identifying the correctness of software by considering its all attributes (Reliability, Scalability, Portability, Re-usability, Usability) and evaluating the execution of software components to find the software bugs or errors or defects.

Testing is a group of techniques to determine the correctness of the application under the predefined script but, testing cannot find all the defect of application. The main intent of testing is to detect failures of the application so that failures can be discovered and corrected. It does not demonstrate that a product functions properly under all conditions but only that it is not working in some specific conditions.

Testing includes an examination of code and also the execution of code in various environments, conditions as well as all the examining aspects of the code. In the current scenario of software development, a testing team may be separate from the development team so that Information derived from testing can be used to correct the process of software development.

Types of Testing:

The various types of testing available in the market are as follows.



Manual testing

Manual testing is a software testing process in which test cases are executed manually without using any automated tool. All test cases executed by the tester manually according to the end user's perspective. It ensures whether the application is working, as mentioned in the requirement document or not. Test cases are planned and implemented to complete almost 100 percent of the software application. Test case reports are also generated manually.

The process of checking the functionality of an application as per the customer needs without taking any help of automation tools is known as manual testing. While performing the manual testing on any application, we do not need any specific knowledge of any testing tool, rather than have a proper understanding of the product so we can easily prepare the test document.

Manual Testing is one of the most fundamental testing processes as it can find both visible and hidden defects of the software. The difference between expected output and output, given by the software, is defined as a defect. The developer fixed the defects and handed it to the tester for retesting.

Manual testing is mandatory for every newly developed software before automated testing. This testing requires great efforts and time, but it gives the surety of bug-free software. Manual Testing requires knowledge of manual testing techniques but not of any automated testing tool.

Need of Manual Testing:

Whenever an application comes into the market, and it is unstable or having a bug or issues or creating a problem while end-users are using it.

If we don't want to face these kinds of problems, we need to perform one round of testing to make the application bug free and stable and deliver a quality product to the client, because if the application is bug free, the end-user will use the application more conveniently.

If the test engineer does manual testing, he/she can test the application as an end-user perspective and get more familiar with the product, which helps them to write the correct test cases of the application and give the quick feedback of the application.

Manual testing can be further divided into three types of testing, which are as follows:

- **White box testing**
- **Black box testing**
- **Gray box testing**

White box Testing:

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 term 'white box' is used because of the internal perspective of the system. The clear box or white box or transparent box name denote the ability to see through the software's outer shell into its inner workings.

Developers do white box testing. In this, the developer will test every line of the code of the program. The developers perform the White-box testing and then send the application or the software to the testing team, where they will perform the black box testing and verify the application along with the requirements and identify the bugs and sends it to the developer.

The developer fixes the bugs and does one round of white box testing and sends it to the testing team. Here, fixing the bugs implies that the bug is deleted, and the particular feature is working fine on the application.

Here, the test engineers will not include in fixing the defects for the following reasons:

- Fixing the bug might interrupt the other features. Therefore, the test engineer should always find the bugs, and developers should still be doing the bug fixes.
- If the test engineers spend most of the time fixing the defects, then they may be unable to find the other bugs in the application.

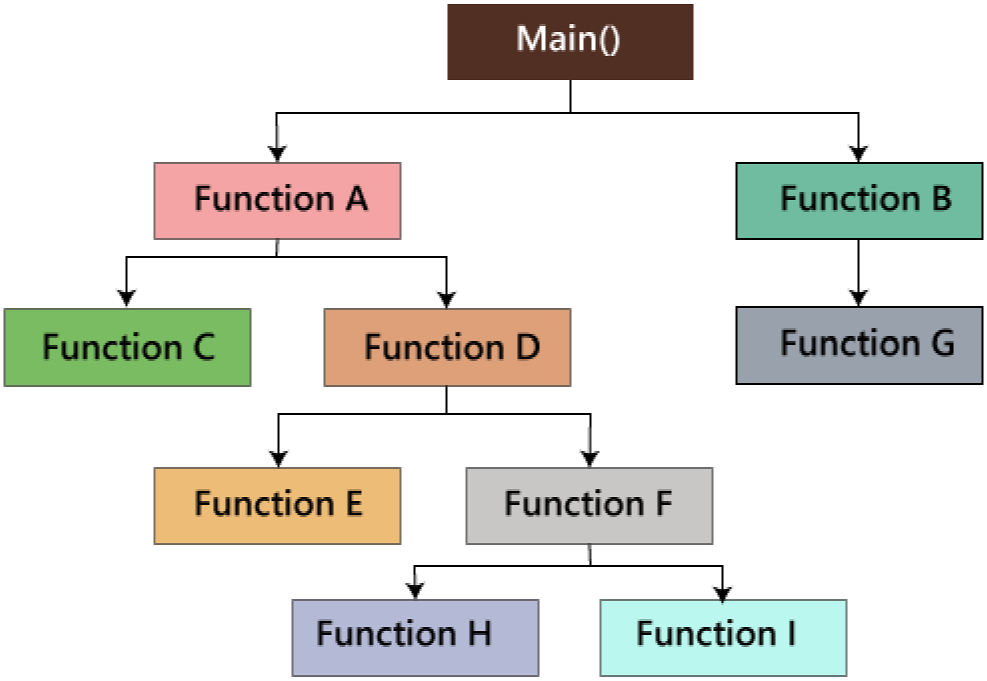
The white box testing contains various tests, which are as follows:

- 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:



And test all the independent paths implies that suppose a path from main() to function G, first set the parameters and test if the program is correct in that particular path, and in the same way test all other paths and fix the bugs.

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.

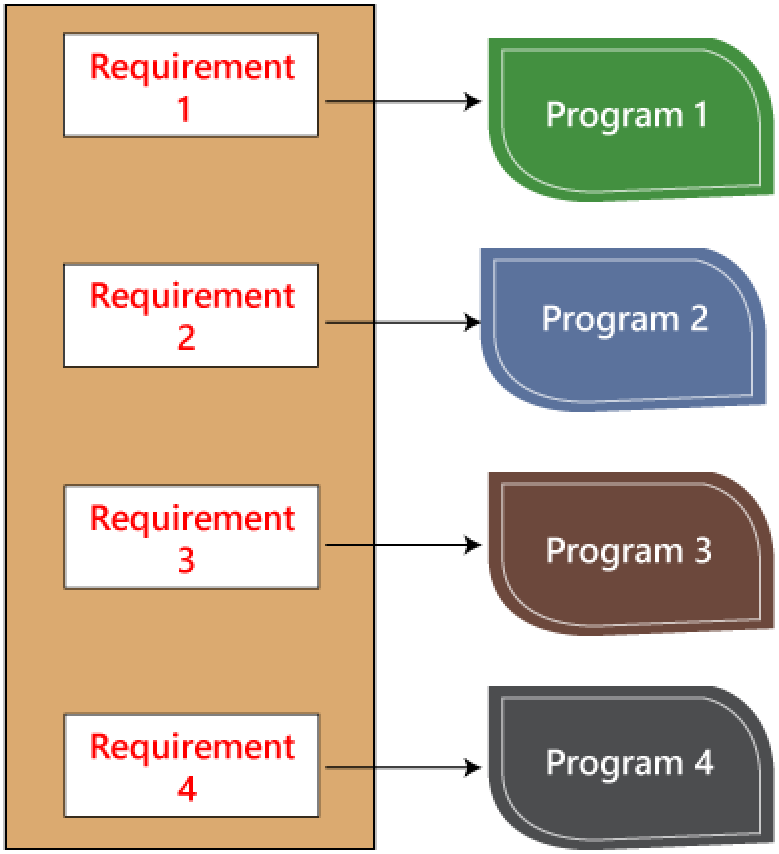
For example: we have one program where the developers have given about 50,000 loops.

```
{
while(50,000)
……
……
}
```

We cannot test this program manually for all the 50,000 loops cycle. So we write a small program that helps for all 50,000 cycles, as we can see in the below program, that test P is written in the similar language as the source code program, and this is known as a Unit test. And it is written by the developers only.

```
Test P
{
……
…… }
```

As we can see in the below image that, we have various requirements such as 1, 2, 3, 4. And then, the developer writes the programs such as program 1,2,3,4 for the parallel conditions. Here the application contains the 100s line of codes.

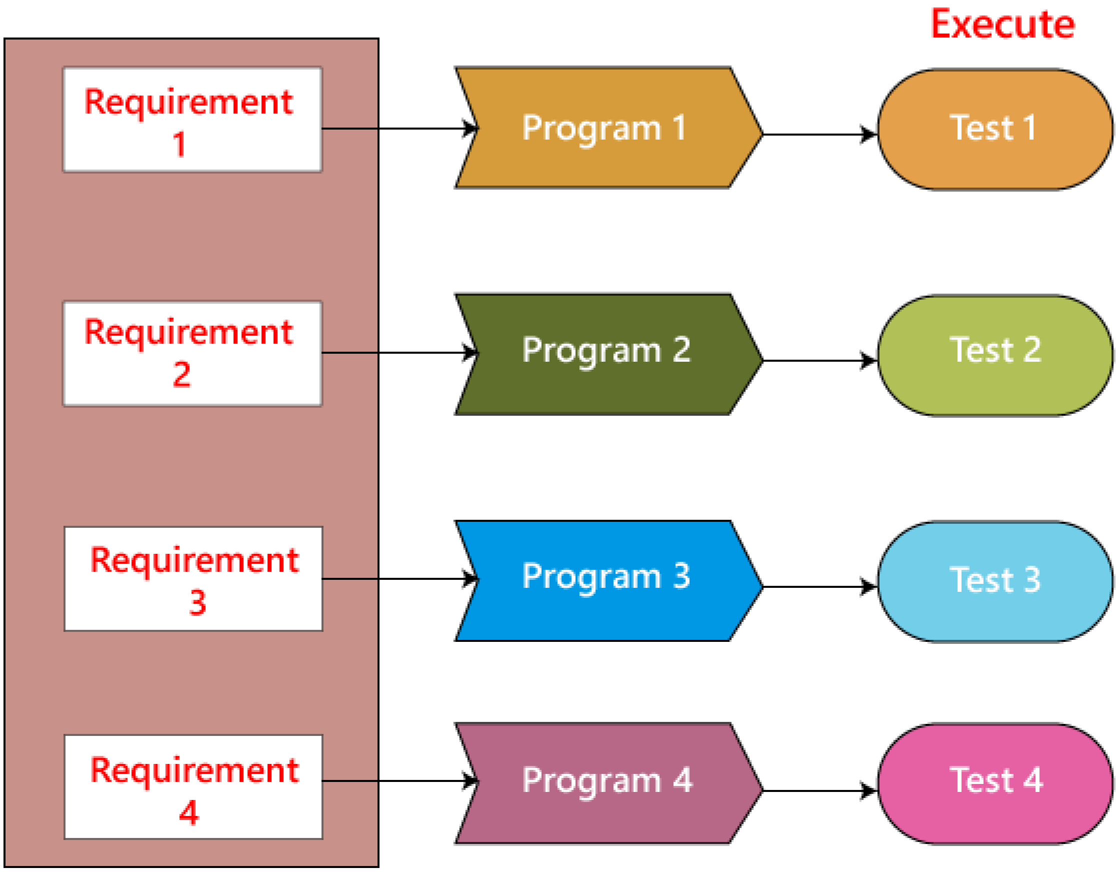


The developer will do the white box testing, and they will test all the five programs line by line of code to find the bug. If they found any bug in any of the programs, they will correct it. And they again have to test the system then this process contains lots of time and effort and slows down the product release time.

Now, suppose we have another case, where the clients want to modify the requirements, then the developer will do the required changes and test all four program again, which take lots of time and efforts.

These issues can be resolved in the following ways:

In this, we will write test for a similar program where the developer writes these test code in the related language as the source code. Then they execute these test code, which is also known as **unit test programs**. These test programs linked to the main program and implemented as programs.



Therefore, if there is any requirement of modification or bug in the code, then the developer makes the adjustment both in the main program and the test program and then executes the test program.

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.

For example:

```
if(condition) - true

{

…..

……
……
}
else - false
{
…..
……
……
}
```

The above program will work fine for both the conditions, which means that if the condition is accurate, and then else should be false and conversely.

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.

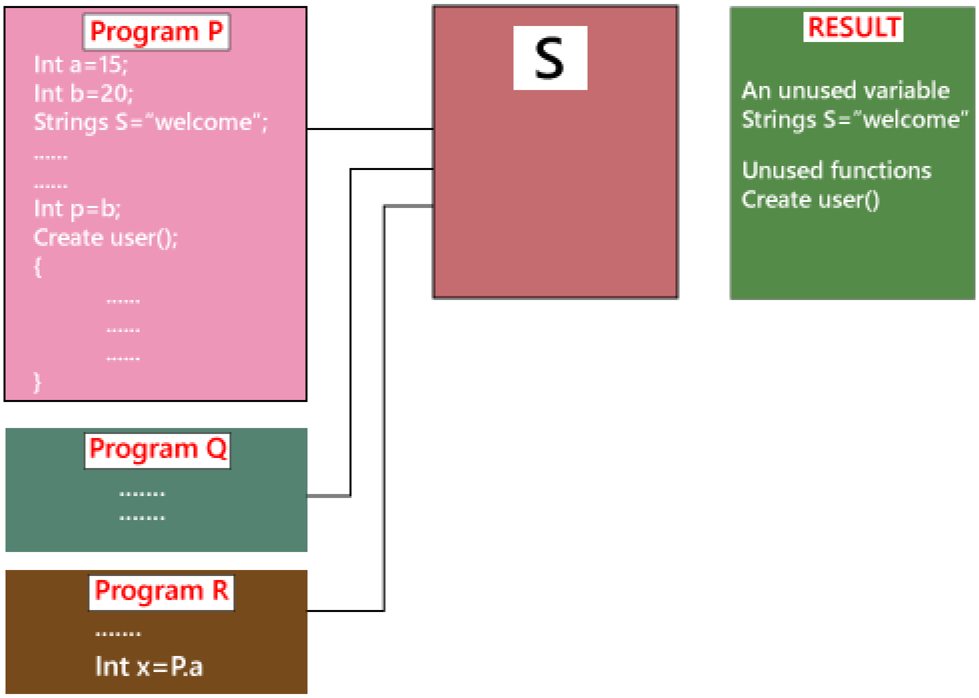
For example,

```
Int a=15;
Int b=20;
```

```
String S= "Welcome";
….
…..
…..
….
…..
Int p=b;
Create user()
{
……
……
….. 200's line of code
}
```

In the above code, we can see that the **integer a** has never been called anywhere in the program, and also the function **Create user** has never been called anywhere in the code. Therefore, it leads us to memory consumption.

We cannot remember this type of mistake manually by verifying the code because of the large code. So, we have a built-in tool, which helps us to test the needless variables and functions. And, here we have the tool called **Rational purify**.



Suppose we have three programs such as Program P, Q, and R, which provides the input to S. And S goes into the programs and verifies the unused variables and then gives the outcome. After that, the developers will click on several results and call or remove the unnecessary function and the variables.

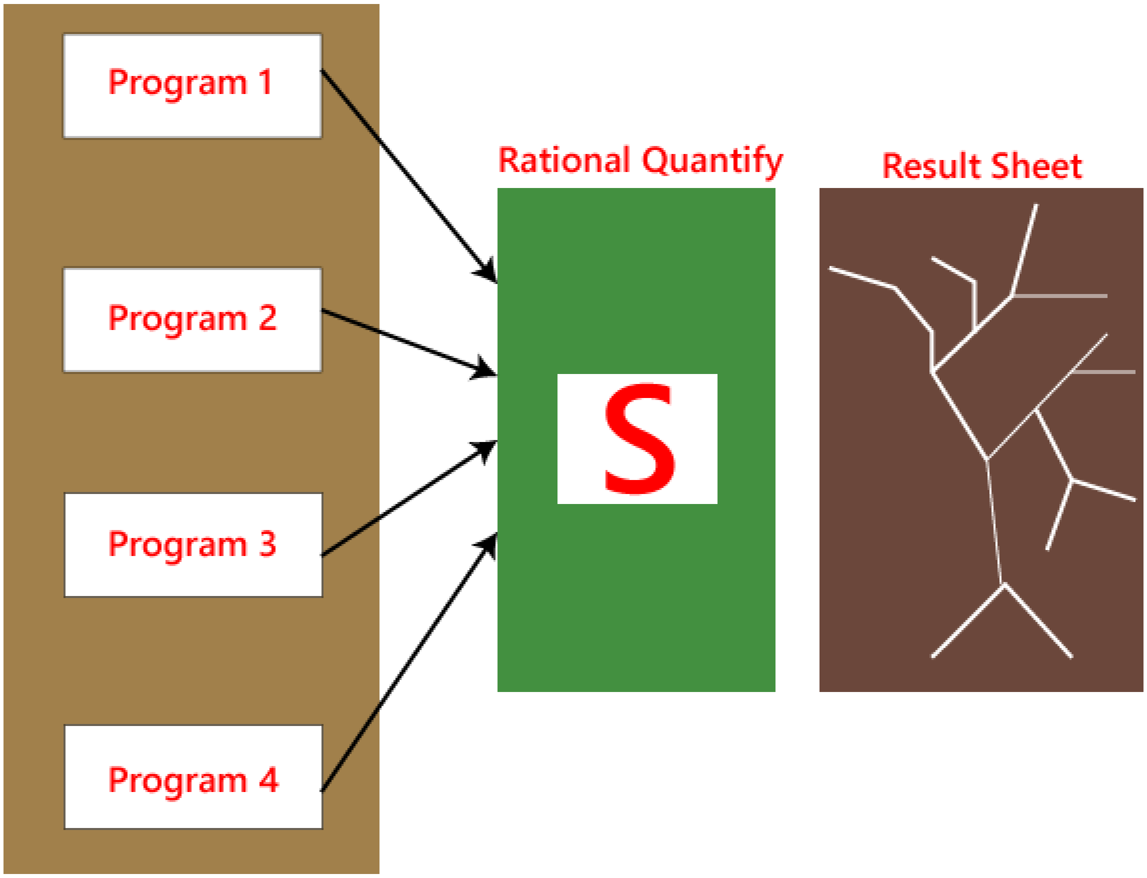
This tool is only used for the C programming language and C++ programming language; for another language, we have other related tools available in the market.

- The developer does not use the available in-built functions; instead they write the full features using their logic. Therefore, it leads us to waste of time and also postpone the product releases.

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.

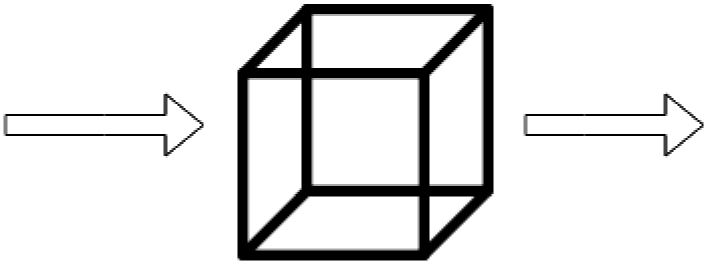


As we know that the developer is performing white box testing, they understand that the code is running slow, or the performance of the program is also getting deliberate. And the developer cannot go manually over the program and verify which line of the code is slowing the program.

To recover with this condition, we have a tool called **Rational Quantify**, which resolves these kinds of issues automatically. Once the entire code is ready, the rational quantify tool will go through the code and execute it. And we can see the outcome in the result sheet in the form of thick and thin lines.

Here, the thick line specifies which section of code is time-consuming. When we double-click on the thick line, the tool will take us to that line or piece of code automatically, which is also displayed in a different color. We can change that code and again and use this tool. When the order of lines is all thin, we know that the presentation of the program has enhanced. And the developers will perform the white box testing automatically because it saves time rather than performing manually.

Test cases for white box testing are derived from the design phase of the software development lifecycle. Data flow testing, control flow testing, path testing, branch testing, statement and decision coverage all these techniques used by white box testing as a guideline to create an error-free software.



Whitebox Testing

White box testing follows some working steps to make testing manageable and easy to understand what the next task to do. There are some basic steps to perform white box testing.

Generic steps of white box testing

- Design all test scenarios, test cases and prioritize them according to high priority number.
- This step involves the study of code at runtime to examine the resource utilization, not accessed areas of the code, time taken by various methods and operations and so on.
- In this step testing of internal subroutines takes place. Internal subroutines such as nonpublic methods, interfaces are able to handle all types of data appropriately or not.
- This step focuses on testing of control statements like loops and conditional statements to check the efficiency and accuracy for different data inputs.
- In the last step white box testing includes security testing to check all possible security loopholes by looking at how the code handles security.

Reasons for white box testing

- It identifies internal security holes.
- To check the way of input inside the code.
- Check the functionality of conditional loops.
- To test function, object, and statement at an individual level.

Advantages of White box testing

- White box testing optimizes code so hidden errors can be identified.
- Test cases of white box testing can be easily automated.
- This testing is more thorough than other testing approaches as it covers all code paths.
- It can be started in the SDLC phase even without GUI.

Disadvantages of White box testing

- White box testing is too much time consuming when it comes to large-scale programming applications.
- White box testing is much expensive and complex.
- It can lead to production error because it is not detailed by the developers.
- White box testing needs professional programmers who have a detailed knowledge and understanding of programming language and implementation.

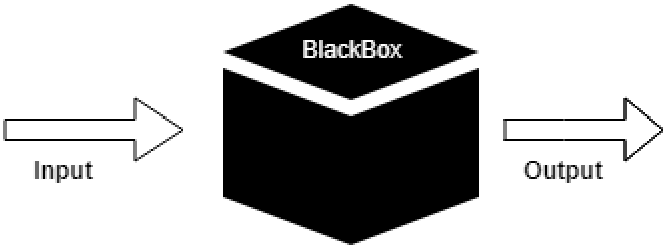
Techniques Used in White Box Testing

Data Flow Testing	Data flow testing is a group of testing strategies that examines the control flow of programs in order to explore the sequence of variables according to the sequence of events.
Control Flow Testing	Control flow testing determines the execution order of statements or instructions of the program through a control structure. The control structure of a program is used to develop a test case for the program. In this technique, a particular part of a large program is selected by the tester to set the testing path. Test cases represented by the control graph of the program.
Branch Testing	Branch coverage technique is used to cover all branches of the control flow graph. It covers all the possible outcomes (true and false) of each condition of decision point at least once.
Statement Testing	Statement coverage technique is used to design white box test cases. This technique involves execution of all statements of the source code at least once. It is used to calculate the total number of executed statements in the source code, out of total statements present in the source code.
Decision Testing	This technique reports true and false outcomes of Boolean expressions. Whenever there is a possibility of two or more outcomes from the statements like do while statement, if statement and case statement (Control flow statements), it is considered as decision point because there are two outcomes either true or false.

Black-Box Testing:

Black box testing is a type of software testing in which the functionality of the software is not known. The testing is done without the internal knowledge of the products.

In this method, tester selects a function and gives input value to examine its functionality, and checks whether the function is giving expected output or not. If the function produces correct output, then it is passed in testing, otherwise failed. The test team reports the result to the development team and then tests the next function. After completing testing of all functions if there are severe problems, then it is given back to the development team for correction.



Generic steps of black box testing

- The black box test is based on the specification of requirements, so it is examined in the beginning.
- In the second step, the tester creates a positive test scenario and an adverse test scenario by selecting valid and invalid input values to check that the software is processing them correctly or incorrectly.
- In the third step, the tester develops various test cases such as decision table, all pairs test, equivalent division, error estimation, cause-effect graph, etc.
- The fourth phase includes the execution of all test cases.
- In the fifth step, the tester compares the expected output against the actual output.
- In the sixth and final step, if there is any flaw in the software, then it is cured and tested again.

Techniques Used in Black Box Testing:

1. **Decision Table**

Decision Table Technique is a systematic approach where various input combinations and their respective system behaviors are captured in a tabular form. It is appropriate for the functions that have a logical relationship between two and more than two inputs.

Ex:

Most of us use an email account, and when you want to use an email account, for this you need to enter the email and its associated password.

If both email and password are correctly matched, the user will be directed to the email account's homepage; otherwise, it will come back to the login page with an error message specified with "Incorrect Email" or "Incorrect Password."

Decision Table for login function is as follows.

Email(Condition1)	T	T	F	F
Password(Condition2)	T	F	T	F
Expected Result(Action)	Emails Account Homepage	Incorrect Password	Incorrect Email	Incorrect Email and Password

While using the decision table technique, a tester determines the expected output, if the function produces expected output, then it is passed in testing, and if not then it is failed. Failed software is sent back to the development team to fix the defect.

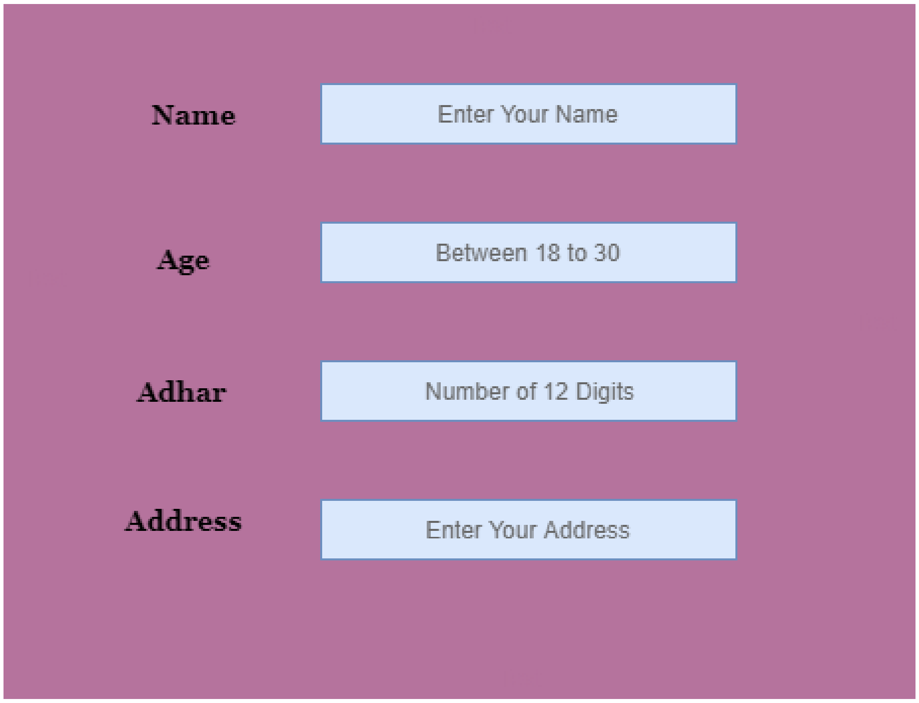
2. **Boundary Value Technique**

Boundary Value Technique is used to test boundary values; boundary values are those that contain the upper and lower limit of a variable. It tests, while entering boundary value whether the software is producing correct output or not.

Ex:

Boundary values are those that contain the upper and lower limit of a variable. Assume that, age is a variable of any function, and its minimum value is 18 and the maximum value is 30, both 18 and 30 will be considered as boundary values.

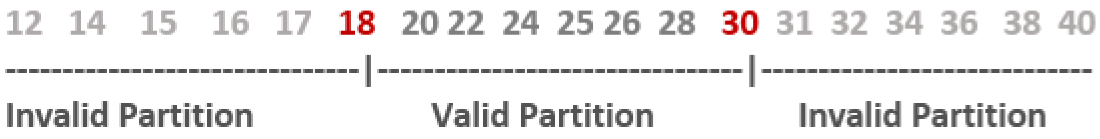
There is 18 and 30 are the boundary values that's why tester pays more attention to these values, but this doesn't mean that the middle values like 19, 20, 21, 27, 29 are ignored. Test cases are developed for each and every value of the range.



Name	Enter Your Name
Age	Between 18 to 30
Adhar	Number of 12 Digits
Address	Enter Your Address

Testing of boundary values is done by making valid and invalid partitions. Invalid partitions are tested because testing of output in adverse condition is also essential.

Ex:



```
12  14   15   16  17  18  20 22  24  25 26  28  30  31  32  34  36  38  40
------------------------------|------------------------------|----------------------------
Invalid Partition              Valid Partition               Invalid Partition
```

Invalid test cases	Valid test cases	Invalid test cases
11, 13, 14, 15, 16, 17	18, 19, 24, 27, 28, 30	31, 32, 36, 37, 38, 39

The software system will be passed in the test if it accepts a valid number and gives the desired output, if it is not, then it is unsuccessful. In another scenario, the software system should not accept invalid numbers, and if the entered number is invalid, then it should display error massage.

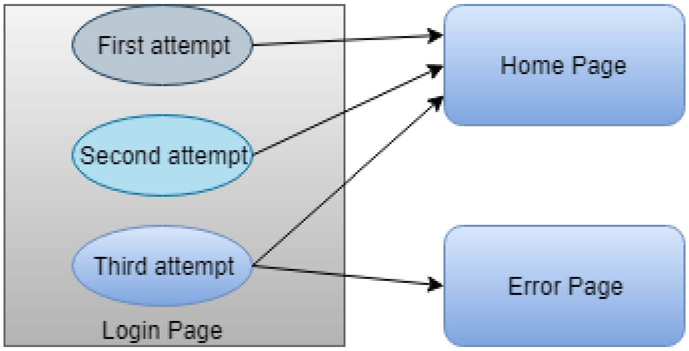
3. State Transition Technique

State Transition Technique is used to capture the behavior of the software application when different input values are given to the same function. This applies to those types of applications that provide the specific number of attempts to access the application.

Ex:

The ATMs, when we withdraw money from it, it displays account details at last. Now we again do another transaction, then it again displays account details, but the details displayed after the second transaction are different from the first transaction, but both details are displayed by using the same function of the ATM. So the same function was used here but each time the output was different, this is called state transition. In the case of testing of a software application, this method tests whether the function is following state transition specifications on entering different inputs.

Ex:



The above example shows the application which provides a maximum three number of attempts, and after exceeding three attempts, it will be directed to an error page.

State transition table

STATE	LOGIN	VALIDATION	REDIRECTED
S1	First Attempt	Invalid	S2
S2	Second Attempt	Invalid	S3
S3	Third Attempt	Invalid	S5
S4	Home Page		
S5	Error Page		

Let's see state transition table if third attempt is valid:

STATE	LOGIN	VALIDATION	REDIRECTED
S1	First Attempt	Invalid	S2
S2	Second Attempt	Invalid	S3
S3	Third Attempt	Valid	S4
S4	Home Page		
S5	Error Page		

4. **All-pair Testing Technique**

All-pair testing Technique is used to test all the possible discrete combinations of values. This combinational method is used for testing the application that uses checkbox input, radio button input, list box, text box, etc.

Ex:

Suppose, you have a function of a software application for testing, in which there are 10 fields to input the data, so the total number of discrete combinations is 10 ^ 10 (100 billion), but testing of all combinations is complicated because it will take a lot of time.

So, let's understand the testing process with an example:

Assume that there is a function with a list box that contains 10 elements, text box that can accept 1 to 100 characters, radio button, checkbox and OK button.

The input values are given below that can be accepted by the fields of the given function.

1. Check Box - Checked or Unchecked
2. List Box - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
3. Radio Button - On or Off
4. Text Box - Number of alphabets between 1 to 100.
5. OK - Does not accept any value, only redirects to the next page.

Calculation of all the possible combinations:

Check Box = 2

List Box = 10

Radio Button = 2

Text Box = 100

Total number of test cases = 2*10*2*100 = 4000

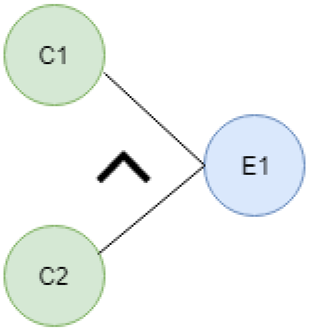
5. **Cause-Effect Technique**

Cause-Effect Technique underlines the relationship between a given result and all the factors affecting the result. It is based on a collection of requirements.

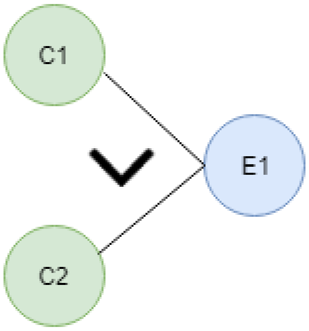
Cause-Effect graph technique is based on a collection of requirements and used to determine minimum possible test cases which can cover a maximum test area of the software.

The main advantage of cause-effect graph testing is, it reduces the time of test execution and cost.

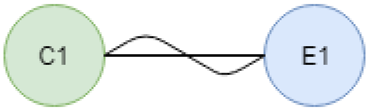
AND - E1 is an effect and C1 and C2 are the causes. If both C1 and C2 are true, then effect E1 will be true.



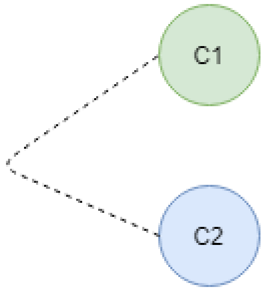
OR - If any cause from C1 and C2 is true, then effect E1 will be true.



NOT - If cause C1 is false, then effect E1 will be true.



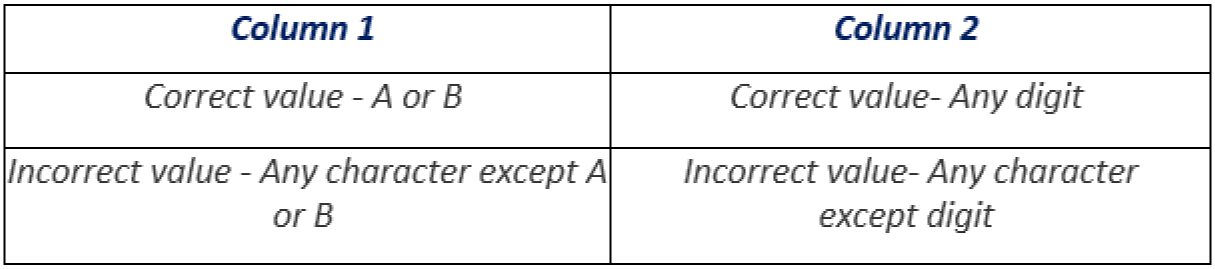
Mutually Exclusive - When only one cause is true.



Ex:

The character in column 1 should be either A or B and in the column 2 should be a digit. If both columns contain appropriate values then update is made. If the input of column 1 is incorrect, i.e. neither A nor B, then message X will be displayed. If the input in column 2 is incorrect, i.e. input is not a digit, then message Y will be displayed.

- A file must be updated, if the character in the first column is either "A" or "B" and in the second column it should be a digit.
- If the value in the first column is incorrect (the character is neither A nor B) then massage X will be displayed.
- If the value in the second column is incorrect (the character is not a digit) then massage Y will be displayed.



Column 1	***Column 2***
Correct value - A or B	*Correct value- Any digit*
Incorrect value - Any character except A or B	*Incorrect value- Any character except digit*

Now, we are going to make a Cause-Effect graph for the above situation:

Causes are:

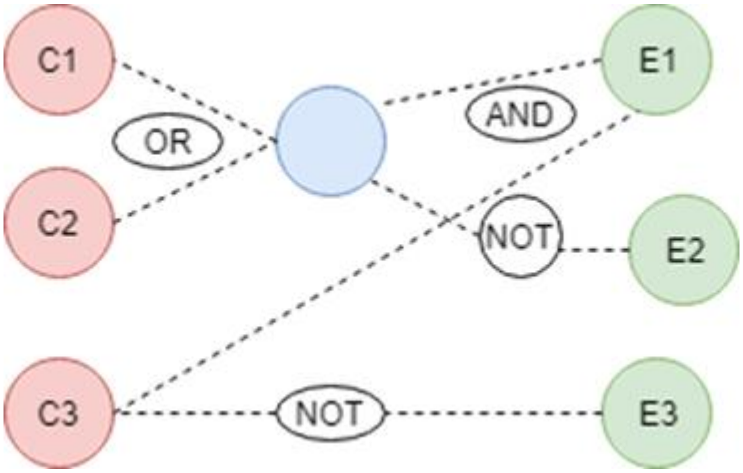
- C1 - Character in column 1 is A
- C2 - Character in column 1 is B

- C3 - Character in column 2 is digit!

Effects:

- E1 - Update made (C1 OR C2) AND C3
- E2 - Displays Massage X (NOT C1 AND NOT C2)
- E3 - Displays Massage Y (NOT C3)

Where AND, OR, NOT are the logical gates.

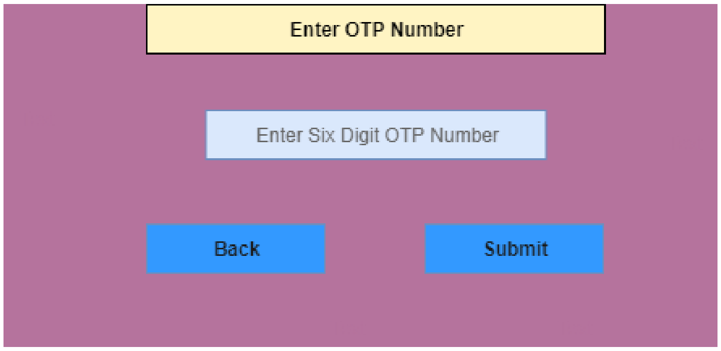


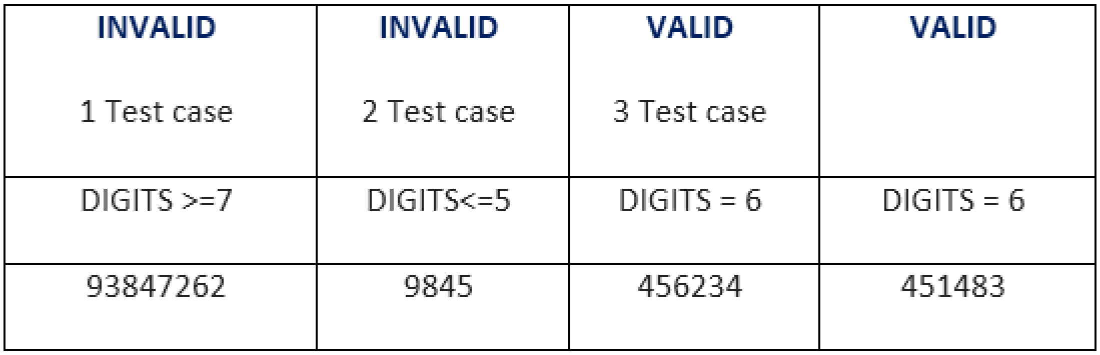
So, it is the cause-effect graph for the given situation. A tester needs to convert causes and effects into logical statements and then design cause-effect graph. If function gives output (effect) according to the input (cause) so, it is considered as defect free, and if not doing so, then it is sent to the development team for the correction.

6. <u>Equivalence Partitioning Technique</u>

Equivalence partitioning is a technique of software testing in which input data divided into partitions of valid and invalid values, and it is mandatory that all partitions must exhibit the same behavior.
Ex:
An OTP number which contains only six digits, less or more than six digits will not be accepted, and the application will redirect the user to the error page.

OTP Number = 6 digits





INVALID 1 Test case	INVALID 2 Test case	VALID 3 Test case	VALID
DIGITS >=7	DIGITS<=5	DIGITS = 6	DIGITS = 6
93847262	9845	456234	451483

Advantages and disadvantages of Equivalence Partitioning technique

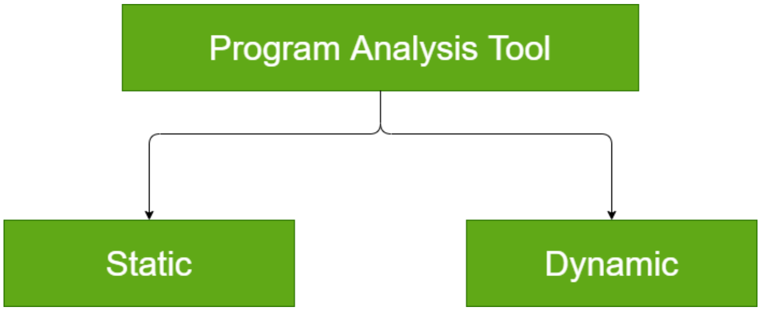
Advantages	**disadvantages**
It is process-oriented	All necessary inputs may not cover.
We can achieve the Minimum test coverage	This technique will not consider the condition for boundary value analysis.
It helps to decrease the general test execution time and also reduce the set of test data.	The test engineer might assume that the output for all data set is right, which leads to the problem during the testing process.

Programming Analysis Tool:

Program Analysis Tool is an automated tool whose input is the source code or the executable code of a program and the output is the observation of characteristics of the program. It gives various characteristics of the program such as its size, complexity, adequacy of commenting, adherence to programming standards and many other characteristics.

Classification of Program Analysis Tools:

Program Analysis Tools are classified into two categories:



- **Static Program Analysis Tools:**
  Static Program Analysis Tool is such a program analysis tool that evaluates and computes various characteristics of a software product without executing it. Normally, static program analysis tools analyze some structural representation of a program to reach a certain analytical conclusion. Basically some structural properties are analyzed using static program analysis tools.

The structural properties that are usually analyzed are:

1. Whether the coding standards have been fulfilled or not.
2. Some programming errors such as uninitialized variables.
3. Mismatch between actual and formal parameters.
4. Variables that are declared but never used.

Code walkthroughs and code inspections are considered as static analysis methods but static program analysis tool is used to designate automated analysis tools. Hence, a compiler can be considered as a static program analysis tool.

- **Dynamic Program Analysis Tools:**

Dynamic Program Analysis Tool is such type of program analysis tool that require the program to be executed and its actual behavior to be observed. A dynamic program analyzer basically implements the code. It adds additional statements in the source code to collect the traces of program execution. When the code is executed, it allows us to observe the behavior of the software for different test cases. Once the software is tested and its behavior is observed, the dynamic program analysis tool performs a post execution analysis and produces reports which describe the structural coverage that has been achieved by the complete testing process for the program.

For example, the post execution dynamic analysis report may provide data on extent statement, branch and path coverage achieved.

The results of dynamic program analysis tools are in the form of a histogram or a pie chart. It describes the structural coverage obtained for different modules of the program. The output of a dynamic program analysis tool can be stored and printed easily and provides evidence that complete testing has been done. The result of dynamic analysis is the extent of testing performed as white box testing. If the testing result is not satisfactory then more test cases are designed and added to the test scenario. Also dynamic analysis helps in elimination of redundant test cases.

Integration Testing:

Integration testing is the process of testing the interface between two software units or modules. It focuses on determining the correctness of the interface. The purpose of integration testing is to expose faults in the interaction between integrated units. Once all the modules have been unit tested, integration testing is performed.

Integration testing is a software testing technique that focuses on verifying the interactions and data exchange between different components or modules of a software application. The goal of integration testing is to identify any problems or bugs that arise when different components are combined and interact with each other. Integration testing is typically performed after unit testing and before system testing. It helps to identify and resolve integration issues early in the development cycle, reducing the risk of more severe and costly problems later on.

Integration testing can be done by picking module by module. This can be done so that there should be a proper sequence to be followed. And also if you don't want to miss out on any integration scenarios then you have to follow the proper sequence. Exposing the defects is the major focus of the integration testing and the time of interaction between the integrated units.
Integration test approaches – There are four types of integration testing approaches. Those approaches are the following:

1. Big-Bang Integration Testing – It is the simplest integration testing approach, where all the modules are combined and the functionality is verified after the completion of individual module testing. In simple words, all the modules of the system are simply put together and tested. This approach is practicable only for very small systems. If an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. So, debugging errors reported during big bang integration testing is very expensive to fix.

Big-Bang integration testing is a software testing approach in which all components or modules of a software application are combined and tested at once. This approach is typically used when the software components have a low degree of interdependence or when there are constraints in the development environment that prevent testing individual components. The goal of big-bang integration testing is to verify the overall functionality of the system and to identify any integration problems that arise when the components are combined. While big-bang integration testing can be useful in some situations, it can also be a high-risk approach, as the complexity of the system and the number of interactions between components can make it difficult to identify and diagnose problems.

Advantages:

1. It is convenient for small systems.

2. Simple and straightforward approach.
3. Can be completed quickly.
4. Does not require a lot of planning or coordination.
5. May be suitable for small systems or projects with a low degree of interdependence between components.

Disadvantages:

1. There will be quite a lot of delay because you would have to wait for all the modules to be integrated.
2. High-risk critical modules are not isolated and tested on priority since all modules are tested at once.
3. Not Good for long projects.
4. High risk of integration problems that are difficult to identify and diagnose.
5. This can result in long and complex debugging and troubleshooting efforts.
6. This can lead to system downtime and increased development costs.
7. May not provide enough visibility into the interactions and data exchange between components.
8. This can result in a lack of confidence in the system's stability and reliability.
9. This can lead to decreased efficiency and productivity.
10. This may result in a lack of confidence in the development team.
11. This can lead to system failure and decreased user satisfaction.

2. Bottom-Up Integration Testing – In bottom-up testing, each module at lower levels are tested with higher modules until all modules are tested. The primary purpose of this integration testing is that each subsystem tests the interfaces among various modules making up the subsystem. This integration testing uses test drivers to drive and pass appropriate data to the lower-level modules.

Advantages:

- In bottom-up testing, no stubs are required.
- A principal advantage of this integration testing is that several disjoint subsystems can be tested simultaneously.
- It is easy to create the test conditions.
- Best for applications that uses bottom up design approach.
- It is Easy to observe the test results.

Disadvantages:

- Driver modules must be produced.
- In this testing, the complexity that occurs when the system is made up of a large number of small subsystems.
- As Far modules have been created, there is no working model can be represented.

3. Top-Down Integration Testing – Top-down integration testing technique is used in order to simulate the behaviour of the lower-level modules that are not yet integrated. In this integration testing, testing takes place from top to bottom. First, high-level modules are tested and then low-level modules and finally integrating the low-level modules to a high level to ensure the system is working as intended.

Advantages:

- Separately debugged module.
- Few or no drivers needed.
- It is more stable and accurate at the aggregate level.
- Easier isolation of interface errors.
- In this, design defects can be found in the early stages.

Disadvantages:

- Needs many Stubs.
- Modules at lower level are tested inadequately.
- It is difficult to observe the test output.
- It is difficult to stub design.

4. Mixed Integration Testing – A mixed integration testing is also called sandwiched integration testing. A mixed integration testing follows a combination of top down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. In bottom-up approach, testing can start only after the bottom level modules are ready. This sandwich or mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. It is also called the hybrid integration testing. also, stubs and drivers are used in mixed integration testing.

Advantages:

- Mixed approach is useful for very large projects having several sub projects.
- This Sandwich approach overcomes this shortcoming of the top-down and bottom-up approaches.
- Parallel test can be performed in top and bottom layer tests.

Disadvantages:

- For mixed integration testing, it requires very high cost because one part has a Top-down approach while another part has a bottom-up approach.
- This integration testing cannot be used for smaller systems with huge interdependence between different modules.

Applications:

1. **Identify the components:** Identify the individual components of your application that need to be integrated. This could include the frontend, backend, database, and any third-party services.

2. **Create a test plan:** Develop a test plan that outlines the scenarios and test cases that need to be executed to validate the integration points between the different components. This could include testing data flow, communication protocols, and error handling.

3. **Set up test environment:** Set up a test environment that mirrors the production environment as closely as possible. This will help ensure that the results of your integration tests are accurate and reliable.

4. **Execute the tests:** Execute the tests outlined in your test plan, starting with the most critical and complex scenarios. Be sure to log any defects or issues that you encounter during testing.

5. **Analyze the results:** Analyze the results of your integration tests to identify any defects or issues that need to be addressed. This may involve working with developers to fix bugs or make changes to the application architecture.

6. **Repeat testing:** Once defects have been fixed, repeat the integration testing process to ensure that the changes have been successful and that the application still works as expected.

Testing Object oriented programs:

Software typically undergoes many levels of testing, from unit testing to system or acceptance testing. Typically, in-unit testing, small “units”, or modules of the software, are tested separately with focus on testing the code of that module. In higher, order testing (e.g, acceptance testing), the entire system (or a subsystem) is tested with the focus on testing the functionality or external behavior of the system.

As information systems are becoming more complex, the object-oriented paradigm is gaining popularity because of its benefits in analysis, design, and coding. Conventional testing methods cannot be applied for testing classes because of problems involved in testing classes, abstract classes, inheritance, dynamic binding, message, passing, polymorphism, concurrency, etc. Testing classes is a fundamentally different problem than testing functions. A function (or a procedure) has a clearly defined input-output behavior, while a class

does not have an input-output behavior specification. We can test a method of a class using approaches for testing functions, but we cannot test the class using these
approaches.

According to Davis the dependencies occurring in conventional systems are:

- Data dependencies between variables
- Calling dependencies between modules
- Functional dependencies between a module and the variable it computes
- Definitional dependencies between a variable and its types.

But in Object-Oriented systems there are following additional dependencies:

- Class to class dependencies
- Class to method dependencies
- Class to message dependencies
- Class to variable dependencies
- Method to variable dependencies
- Method to message dependencies
- Method to method dependencies

Issues in Testing Classes:

Additional testing techniques are, therefore, required to test these dependencies. Another issue of interest is that it is not possible to test the class dynamically, only its instances i.e, objects can be tested. Similarly, the concept of inheritance opens various issues e.g., if changes are made to a parent class or superclass, in a larger system of a class it will be difficult to test subclasses individually and isolate the error to one class.

In object-oriented programs, control flow is characterized by message passing among objects, and the control flow switches from one object to another by inter-object communication. Consequently, there is no control flow within a class like functions. This lack of sequential control flow within a class requires different approaches for testing. Furthermore, in a function, arguments passed to the function with global data determine the path of execution within the procedure. But, in an object, the state associated with the object also influences the path of execution, and methods of a class can communicate among themselves through this state because this state is persistent across invocations of methods. Hence, for testing objects, the state of an object has to play an important role.

Techniques of object-oriented testing are as follows:

1. **Fault Based Testing:**
   This type of checking permits for coming up with test cases supported the consumer specification or the code or both. It tries to identify possible faults (areas of design or code that may lead to errors.). For all of these faults, a test case is developed to “flush” the errors out. These tests also force each time of code to be executed.

This method of testing does not find all types of errors. However, incorrect specification and interface errors can be missed. These types of errors can be uncovered by function testing in the traditional testing model. In the object-oriented model, interaction errors can be uncovered by scenario-based testing. This form of Object oriented-testing can only test against the client’s specifications, so interface errors are still missed.

2. **Class Testing Based on Method Testing:**
   This approach is the simplest approach to test classes. Each method of the class performs a well defined cohesive function and can, therefore, be related to unit testing of the traditional testing techniques. Therefore all the methods of a class can be involved at least once to test the class.

3. **Random Testing:**
   It is supported by developing a random test sequence that tries the minimum variety of operations typical to the behavior of the categories

4. **Partition Testing:**
   This methodology categorizes the inputs and outputs of a category so as to

check them severely. This minimizes the number of cases that have to be designed.

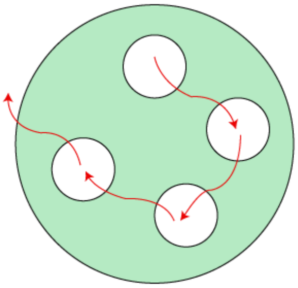
5. **Scenario-based Testing:**
   It primarily involves capturing the user actions then stimulating them to similar actions throughout the test.
   These tests tend to search out interaction form of error.

System Testing:

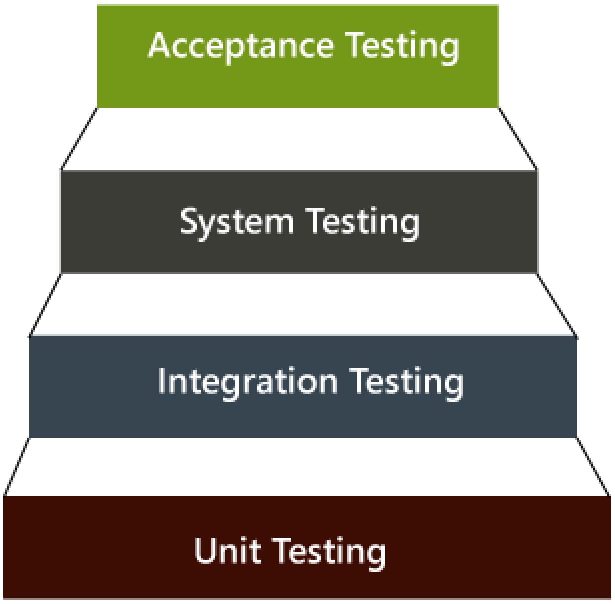
System Testing includes testing of a fully integrated software system. Generally, a computer system is made with the integration of software (any software is only a single element of a computer system). The software is developed in units and then interfaced with other software and hardware to create 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.

To check the end-to-end flow of an application or the software as a user is known as **System testing**. In this, we navigate (go through) all the necessary modules of an application and check if the end features or the end business works fine, and test the product as a whole system.

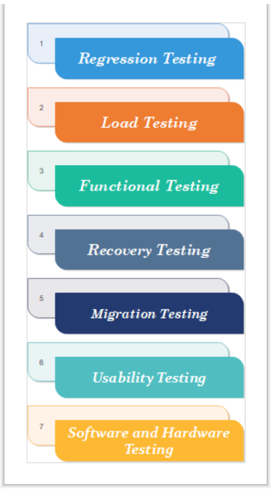
It is **end-to-end testing** where the testing environment is similar to the production environment.



There are four levels of software testing: unit testing, integration testing, system testing and acceptance testing, all are used for the testing purpose. Unit Testing used to test single software; Integration Testing used to test a group of units of software, System Testing used to test a whole system and Acceptance Testing used to test the acceptability of business requirements. Here we are discussing system testing which is the third level of testing levels.



Types of System Testing.



Regression Testing

Regression testing is performed under system testing to confirm and identify that if there's any defect in the system due to modification in any other part of the system. It makes sure, any changes done during the development process have not introduced a new defect and also gives assurance; old defects will not exist on the addition of new software over the time.

Load Testing

Load testing is performed under system testing to clarify whether the system can work under real-time loads or not.

Functional Testing

Functional testing of a system is performed to find if there's any missing function in the system. Tester makes a list of vital functions that should be in the system and can be added during functional testing and should improve quality of the system.

Recovery Testing

Recovery testing of a system is performed under system testing to confirm reliability, trustworthiness, accountability of the system and all are lying on recouping skills of the system. It should be able to recover from all the possible system crashes successfully.

In this testing, we will test the application to check how well it recovers from the crashes or disasters.

Migration Testing

Migration testing is performed to ensure that if the system needs to be modified in new infrastructure so it should be modified without any issue.

Usability Testing

The purpose of this testing to make sure that the system is well familiar with the user and it meets its objective for what it supposed to do.

Software Reliability and Quality Management: Software Reliability, Statistical Testing, Software Quality, Software Quality Management System, ISO 9000, SEI Capability Maturity Model.**Software Maintenance:** Software maintenance, Maintenance Process Models, Maintenance Cost, Software Configuration Management.

SOFTWARE RELIABILITY

The reliability of a software product essentially denotes its trustworthiness or dependability

The reliability of a software product can also be defined as the probability of the product working "correctly" over a given period of time.

The reliability improvement due to fixing a single bug depends on where the bug is located in the code.

The perceived reliability of a software product is observer-dependent.

The reliability of a product keeps changing as errors are detected and fixed.

Hardware versus Software Reliability
Reliability Metrics of Software Products
Reliability Growth Modelling

Hardware versus Software Reliability

Hardware components fail due to very different reasons as compared to software components.

Hardware components fail mostly due to wear and tear, whereas software components fail due to bugs.

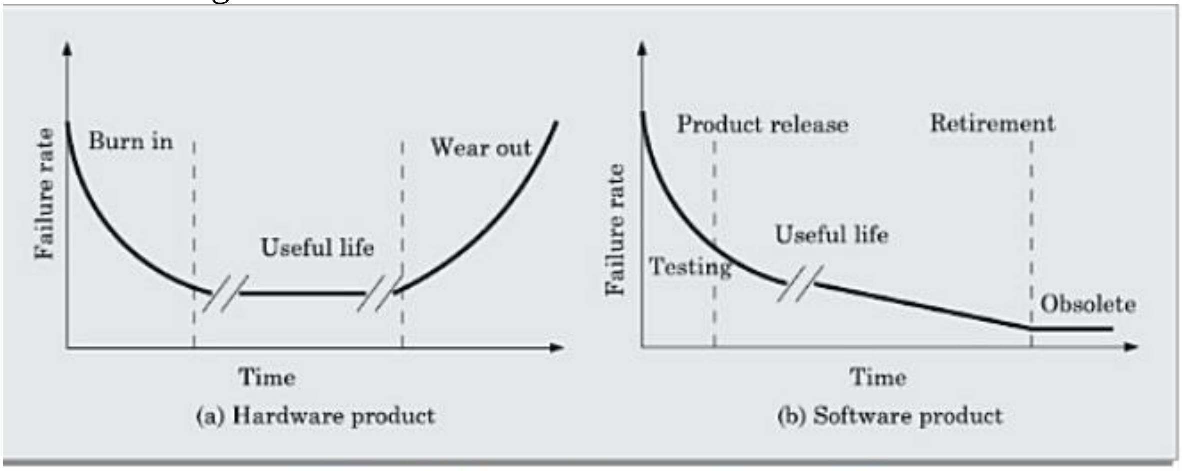


Figure 11.1: Change in failure rate of a product.

Reliability Metrics of Software Products

Rate of occurrence of failure (ROCOF):

ROCOF measures the frequency of occurrence of failures. ROCOF measure of a software product can be obtained by observing the behaviour of a software product in operation over a specified time interval and then calculating the ROCOF value as the ratio of the total number of failures observed and the duration of observation.

Mean time to failure (MTTF):

MTTF is the time between two successive failures, averaged over a large number of failures. To measure MTTF, we can record the failure data for n failures.

Mean time to repair (MTTR):

Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.

Mean time between failure (MTBF):

The MTTF and MTTR metrics can be combined to get the MTBF metric: MTBF=MTTF+MTTR.

Probability of failure on demand (POFOD):

Unlike the other metrics discussed, this metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made.

Availability:

Availability of a system is a measure of how likely would the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs.

Reliability Growth Modelling

A reliability growth model can be used to predict when (or if at all) a particular level of reliability is likely to be attained.

Reliability growth modelling can be used to determine when to stop testing to attain a given reliability level.

Jelinski and Moranda model

The simplest reliability growth model is a step function model where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired. Such a model is shown in Figure 11.2.

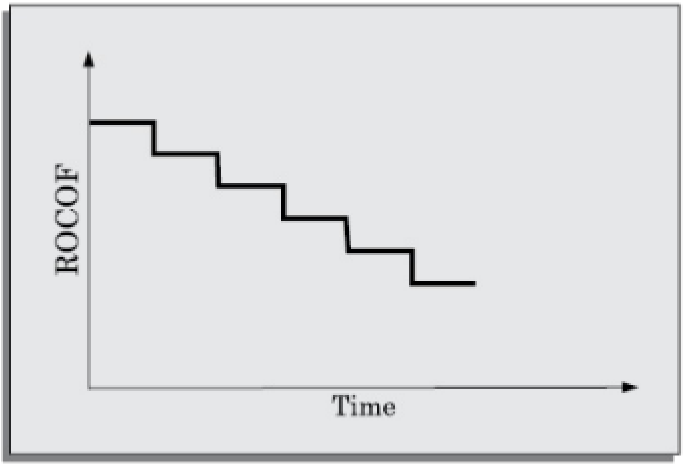


Figure 11.2: Step function model of reliability growth.

Littlewood and Verall's model

This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It also models the fact that as errors are repaired, the average improvement to the product reliability per repair decreases.

STATISTICAL TESTING

Statistical Testing makes use of statistical methods to determine the reliability of the program. Statistical testing focuses on how faulty programs can affect its operating conditions.

How to perform ST?

- Software is tested with the test data that statistically models the working environment.
- Failures are collated and analysed.
- From the computed data, an estimate of the program's failure rate is calculated.
- A Statistical method for testing the possible paths is computed by building an algebraic function.
- Statistical testing is a bootless activity as the intent is NOT to find defects.

Pros and cons of statistical testing

Statistical testing allows one to concentrate on testing parts of the system that are most likely to be used. Therefore, it results in a system that the users can find to be more reliable (than actually it is!).

However, it is not easy to perform the statistical testing satisfactorily due to the following two reasons. There is no simple and repeatable way of defining operation profiles. Also, the number of test cases with which the system is to be tested should be statistically significant.

Software Quality

The quality of a product is defined in terms of its fitness of purpose. That is, a good quality product does exactly what the users want it to do

Quality factors like

Portability: A software product is said to be portable, if it can be easily made to work in different hardware and operating system environments, and easily interface with external hardware devices and software products.

Usability: A software product has good usability, if different categories of users (i.e., both expert and novice users) can easily invoke the functions of the product.

Reusability: A software product has good reusability, if different modules of the product can easily be reused to develop new products.

Correctness: A software product is correct, if different requirements as specified in the SRS document have been correctly implemented.

Maintainability: A software product is maintainable, if errors can be easily corrected as and when they show up, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

SOFTWARE QUALITY MANAGEMENT SYSTEM

A quality management system (often referred to as quality system) is the principal methodology used by organisations to ensure that the products they develop have the desired quality

Managerial structure and individual responsibilities

A quality system is the responsibility of the organisation as a whole. However, every organisation has a separate quality department to perform several quality system activities. The quality system of an organisation should have the full support of the top management.

Quality system activities

- Auditing of projects to check if the processes are being followed.

- Collect process and product metrics and analyse them to check if quality goals are being met.
- Review of the quality system to make it more effective.
- Development of standards, procedures, and guidelines.
- Produce reports for the top management summarising the effectiveness of the quality system in the organisation.

Evolution of Quality Systems

Product Metrics versus Process Metrics

Evolution of Quality Systems

Quality control (QC) focuses not only on detecting the defective products and eliminating them, but also on determining the causes behind the defects, so that the product rejection rate can be reduced.

The basic premise of modern quality assurance is that if an organisation's processes are good and are followed rigorously, then the products are bound to be of good quality.

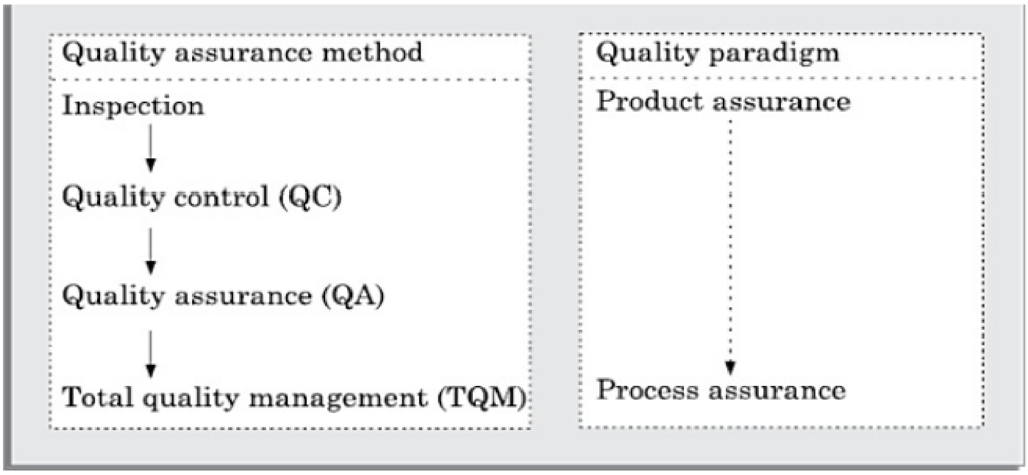


Figure 11.3: Evolution of quality system and corresponding shift in the quality paradigm.

Product Metrics versus Process Metrics

Product metrics help measure the characteristics of a product being developed, whereas process metrics help measure how a process is performing.

Examples of product metrics are LOC and function point to measure size, PM (person- month) to measure the effort required to develop it, months to measure the time required to develop the product, time complexity of the algorithms,

Examples of process metrics are review effectiveness, average number of defects found per hour of inspection, average defect correction time, productivity, average number of failures detected during testing per LOC, number of latent defects per line of code in the developed product.

ISO 9000

The basic premise of modern quality assurance is that if an organisation's processes are good and are followed rigorously, then the products are bound to be of good quality.

Quality system of an organisation means the various activities related to its products or services. Standard of ISO addresses both aspects i.e. operational and organisational aspects which includes responsibilities, reporting etc.

Features of ISO 9000

DocumentControl

All documents concerned with the development of a software product should be properly managed and controlled.

Planning

Proper plans should be prepared and monitored.

Review

For effectiveness and correctness all important documents across all phases should be independently checked and reviewed .

Testing

The product should be tested against specification.

OrganizationalAspects

Various organizational aspects should be addressed e.g., management reporting of the quality team.

Advantages of ISO 9000 Certification

- Business ISO-9000 certification forces a corporation to specialize in "how they are doing business".
- Employees morale is increased as they're asked to require control of their processes and document their work processes
- Better products and services result from continuous improvement process.
- Increased employee participation, involvement, awareness and systematic employee training are reduced problems.

Shortcomings of ISO 9000 Certification

- ISO 9000 does not give any guidelines for defining an appropriate process and does not give guarantee for high quality processes.
- ISO 9000 certification process has no international accreditation agency exists.

SEI Capability Maturity Model

CMM was developed by the Software Engineering Institute (SEI) at Carnegie Mellon University in 1987.

It is a framework that is used to analyze the approach and techniques followed by any organization to develop software products.

It also provides guidelines to further enhance the maturity of the process used to develop those software products.

It is based on profound feedback and development practices adopted by the most successful organizations worldwide.

This model describes a strategy for software process improvement that should be followed by moving through 5 different levels.

Each level of maturity shows a process capability level. All the levels except level-1 are further described by Key Process Areas (KPA's).

Shortcomings of SEI/CMM:

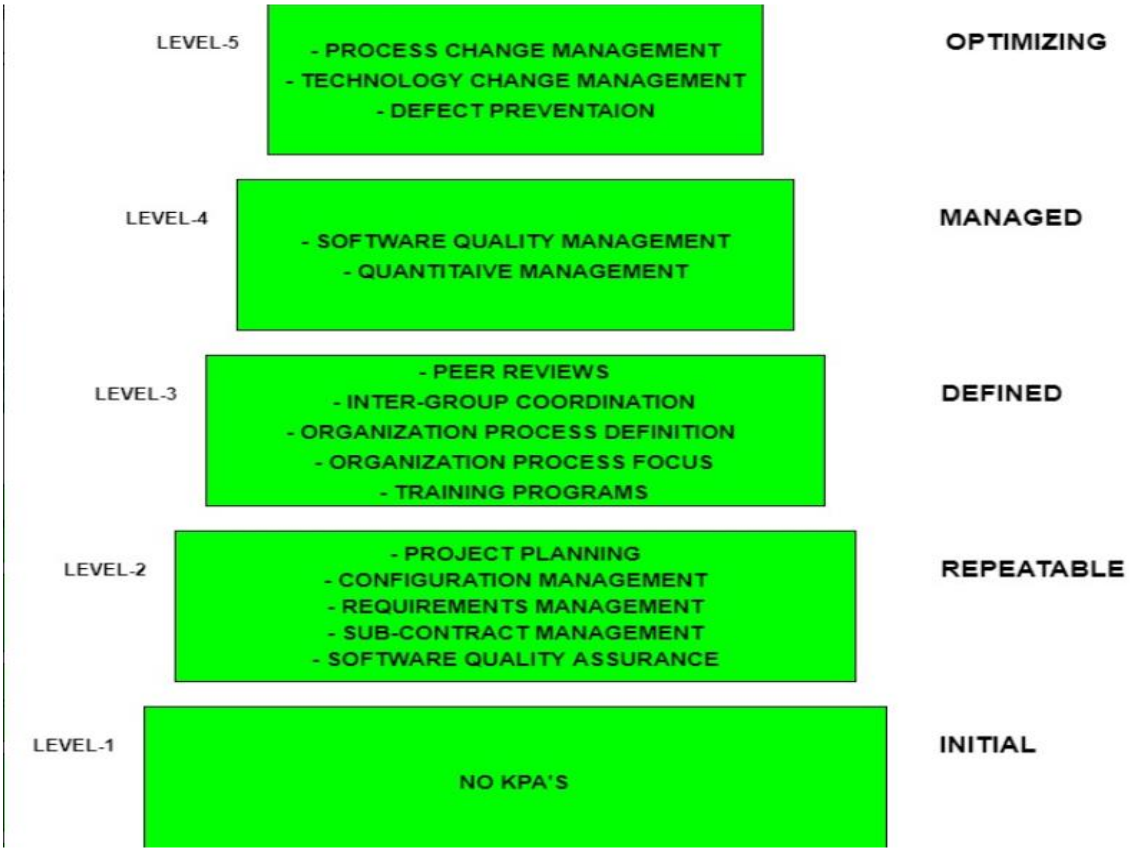
- It only helps if it is put into place early in the software development process.
- It has no formal theoretical basis and in fact is based on the experience of very knowledgeable people.

Key Process Areas (KPA's):

Each of these KPA's defines the basic requirements that should be met by a software process in order to satisfy the KPA and achieve that level of maturity.

Key process areas form the basis for management control of the software project and establish a context in which technical methods are applied, work products like

models, documents, data, reports, etc. are produced, milestones are established, quality is ensured and change is properly managed.
There are five levels in SEI/CMM.They are


LEVEL-5
- PROCESS CHANGE MANAGEMENT
- TECHNOLOGY CHANGE MANAGEMENT
- DEFECT PREVENTAION
OPTIMIZING
LEVEL-4
- SOFTWARE QUALITY MANAGEMENT
- QUANTITAIVE MANAGEMENT
MANAGED
LEVEL-3
- PEER REVIEWS
- INTER-GROUP COORDINATION
- ORGANIZATION PROCESS DEFINITION
- ORGANIZATION PROCESS FOCUS
- TRAINING PROGRAMS
DEFINED
LEVEL-2
- PROJECT PLANNING
- CONFIGURATION MANAGEMENT
- REQUIREMENTS MANAGEMENT
- SUB-CONTRACT MANAGEMENT
- SOFTWARE QUALITY ASSURANCE
REPEATABLE
LEVEL-1
NO KPA'S
INITIAL

Level-1: Initial –

- No KPA's defined.
- Processes followed are Adhoc and immature and are not well defined.
- Unstable environment for software development.
- No basis for predicting product quality, time for completion, etc.

Level-2: Repeatable –

- Focuses on establishing basic project management policies.
- Experience with earlier projects is used for managing new similar natured projects.
- **Project Planning-** It includes defining resources required, goals, constraints, etc. for the project. It presents a detailed plan to be followed systematically for the successful completion of good quality software.
- **Configuration Management-** The focus is on maintaining the performance of the software product, including all its components, for the entire lifecycle.
- **Requirements Management-** It includes the management of customer reviews and feedback which result in some changes in the requirement set. It also consists of accommodation of those modified requirements.
- **Subcontract Management-** It focuses on the effective management of qualified software contractors i.e. it manages the parts of the software which are developed by third parties.
- **Software Quality Assurance-** It guarantees a good quality software product by following certain rules and quality standard guidelines while developing.

Level-3: Defined –

- At this level, documentation of the standard guidelines and procedures takes place.
- It is a well-defined integrated set of project-specific software engineering and management processes.
- **Peer Reviews-** In this method, defects are removed by using a number of review methods like walkthroughs, inspections, buddy checks, etc.
- **Intergroup Coordination-** It consists of planned interactions between different development teams to ensure efficient and proper fulfillment of customer needs.
- **Organization Process Definition-** Its key focus is on the development and maintenance of the standard development processes.
- **Organization Process Focus-** It includes activities and practices that should be followed to improve the process capabilities of an organization.
- **Training Programs-** It focuses on the enhancement of knowledge and skills of the team members including the developers and ensuring an increase in work efficiency.

Level-4: Managed –

- At this stage, quantitative quality goals are set for the organization for software products as well as software processes.
- The measurements made help the organization to predict the product and process quality within some limits defined quantitatively.
- **Software Quality Management-** It includes the establishment of plans and strategies to develop quantitative analysis and understanding of the product's quality.
- **Quantitative Management-** It focuses on controlling the project performance in a quantitative manner.

Level-5: Optimizing –

- This is the highest level of process maturity in CMM and focuses on continuous process improvement in the organization using quantitative feedback.
- Use of new tools, techniques, and evaluation of software processes is done to prevent recurrence of known defects.
- **Process Change Management-** Its focus is on the continuous improvement of the organization's software processes to improve productivity, quality, and cycle time for the software product.
- **Technology Change Management-** It consists of the identification and use of new technologies to improve product quality and decrease product development time.
- **Defect Prevention-** It focuses on the identification of causes of defects and prevents them from recurring in future projects by improving project-defined processes.

Software Maintenance

Software Maintenance is the process of modifying a software product after it has been delivered to the customer.

The main purpose of software maintenance is to modify and update software applications after delivery to correct faults and to improve performance.

Need for Maintenance

- Correct faults.

- Improve the design.
- Implement enhancements.
- Interface with other systems.
- Accommodate programs so that different hardware, software, system features, and telecommunications facilities can be used.

Categories of Software Maintenance –

Maintenance can be divided into the following:

1. **Corrective maintenance:** Corrective maintenance of a software product may be essential either to rectify some bugs observed while the system is in use, or to enhance the performance of the system.
2. **Adaptive maintenance:** This includes modifications and updates when the customers need the product to run on new platforms, on new operating systems, or when they need the product to interface with new hardware and software.
3. **Perfective maintenance:** A software product needs maintenance to support the new features that the users want or to change different types of functionalities of the system according to the customer demands.
4. **Preventive maintenance:** This type of maintenance includes modifications and updations to prevent future problems of the software. It aims to attend to problems, which are not significant at this moment but may cause serious issues in future.

SOFTWARE MAINTENANCE PROCESS MODELS

The activities involved in a software maintenance project are not unique and depend on several factors such as:

(i) the extent of modification to the product required

(ii)the resources available to the maintenance team

(iii)the conditions of the existing product (e.g., how structured it is, how well documented it is)

(iv)the expected project risks, etc.

Software Maintenance process models are divided into two types .They are

First model

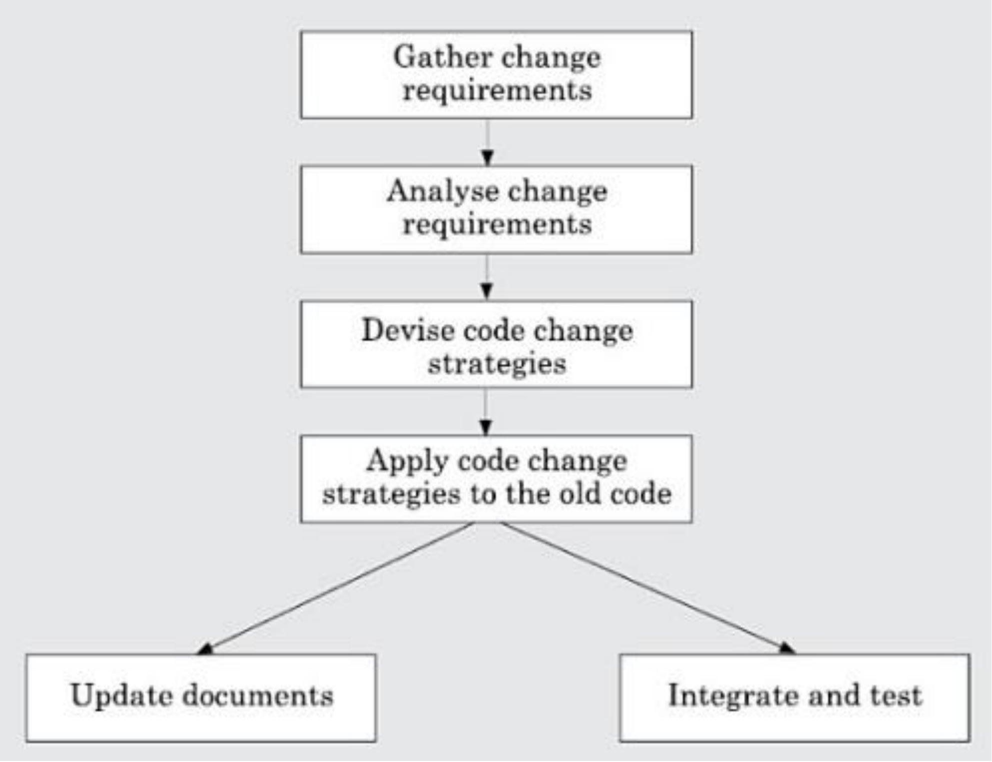
Second model

First Model

The first model is preferred for projects involving small reworks where the code is changed directly and the changes are reflected in the relevant documents later.

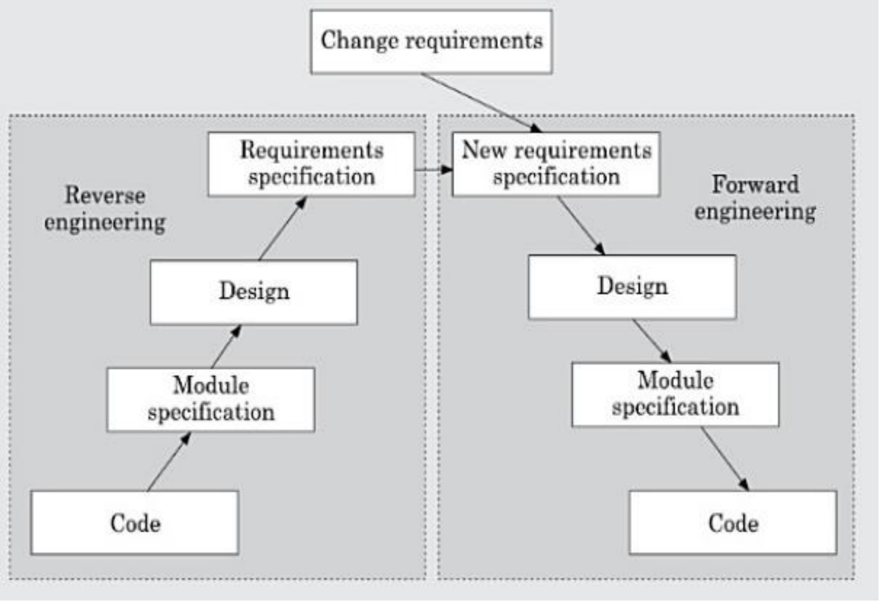
This maintenance process is graphically presented in below Figure .

In this approach, the project starts by gathering the requirements for changes. The requirements are next analysed to formulate the strategies to be adopted for code change. At this stage, the association of at least a few members of the original development team goes a long way in reducing the cycle time, especially for projects involving unstructured and inadequately documented code. The availability of a working old system to the maintenance engineers at the maintenance site greatly facilitates the task of the maintenance team as they get a good insight into the working of the old system and also can compare the working of their modified system with the old system. Also, debugging of the reengineered system becomes easier as the program traces of both the systems can be compared to localise the bugs.


Gather change requirements
Analyse change requirements
Devise code change strategies
Apply code change strategies to the old code
Update documents
Integrate and test

Second model

The second model is preferred for projects where the amount of rework required is significant. This approach can be represented by a reverse engineering cycle followed by a forward engineering cycle. Such an approach is also known as software re-engineering.



Re-engineering might be preferable for products which exhibit a high failure rate. Re-engineering might also be preferable for legacy products having poor design and code structure.

Maintenance Cost

The cost of system maintenance represents a large proportion of the budget of most organisations that use software system

Reasons for intangible cost

Customer dissatisfaction when requests for repair or modification cannot be addressed in a timely manner.

Reduction in overall software quality as a result of changes that introduce hidden errors in maintained software.

Software maintenance cost factors:

Non-Technical factors
Technical factors

Non-Technical factors:

1. Staff stability
2. Program lifetime
3. Dependence on External Environment
4. Hardware stability

Technical factors:

1. module independence
2. Programming language
3. Programming style
4. Program validation and testing
5. Documentation
6. Configuration management techniques

ESTIMATION OF MAINTENANCE COST

Boehm [1981] proposed a formula for estimating maintenance costs as part of his COCOMO cost estimation model.

Annual change traffic (ACT)

$$ACT = \frac{KLOC_{added} + KLOC_{deleted}}{KLOC_{total}}$$

KLOCadded is the total kilo lines of source code added during maintenance.
KLOCdeleted is the total KLOC deleted during maintenance.
Maintenance cost = ACT × Development cost

Software Configuration Management

It is just similar to an umbrella activity which is to be applied throughout the software process. It manages and tracks the emerging product and its versions also it identifies and controls the configuration of software, hardware and the tools that are used throughout the development cycle

It is an arrangement of exercises which controls change by recognizing the things for change, setting up connections between those things, making/characterizing instruments for overseeing diverse variants, controlling the changes being executed in the current framework, inspecting and revealing/reporting on the changes made.

Processes involved in SCM

Identification and Establishment:

Identifying the configuration items from products that compose baselines at given points in time.

Establishing relationship among items, creating a mechanism to manage multiple level of control and procedure for change management

Version control:

Creating versions/specifications of the existing product to build new products from the help of SCM system.

Change control:

Configuration auditing:

A software configuration audit complements the formal technical review of the process and product. It focuses on the technical correctness of the configuration object that has been modified.

Reporting :

A software configuration audit complements the formal technical review of the process and product. It focuses on the technical correctness of the configuration object that has been modified.