SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – I – Software Engineering – SCS1305

I INTRODUCTION

S/W engineering paradigm - Life cycle models - Water fall - Incremental - Spiral - Evolutionary - Prototyping - Object oriented system engineering - Computer based system - Verification - Validation - Life cycle process - Development process - System engineering hierarchy - Introduction to CMM - Levels of CMM.

Evolving Role of Software:

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called software product.

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods.

Software engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

Fritz Bauer, a German computer scientist, defines software engineering as:

Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and work efficiently on real machines.

Software Evolution

The process of developing a software product using software engineering principles and methods is referred to as software evolution. This includes the initial development of software and its maintenance and updates, till desired software product is developed, which satisfies the expected requirements,

Fig 1.1: Software Evolution

Evolution starts from the requirement gathering process. After which developers create a prototype of the intended software and show it to the users to get their feedback at the early stage of software product development. The users suggest changes, on which several consecutive updates and maintenance keep on changing too. This process changes to the original software, till the desired software is accomplished.

Even after the user has desired software in hand, the advancing technology and the changing requirements force the software product to change accordingly. Re-creating software from scratch and to go one-on-one with requirement is not feasible. The only feasible and economical solution is to update the existing software so that it matches the latest requirements.

Software Evolution Laws

Lehman has given laws for software evolution. He divided the software into three different categories:

- **S-type (static-type) -** This is a software, which works strictly according to defined specifications and solutions. The solution and the method to achieve it, both are immediately understood before coding. The s-type software is least subjected to changes hence this is the simplest of all. For example, calculator program for mathematical computation.
- **P-type (practical-type) -** This is a software with a collection of procedures. This is defined by exactly what procedures can do. In this software, the specifications can be described but the solution is not obvious instantly. For example, gaming software.
- **E-type (embedded-type) -** This software works closely as the requirement of real-world environment. This software has a high degree of evolution as

there are various changes in laws, taxes etc. in the real-world situations. For example, Online trading software.

E-Type software evolution

Lehman has given eight laws for E-Type software evolution -

- **Continuing change -** An E-type software system must continue to adapt to the real-world changes, else it becomes progressively less useful.
- **Increasing complexity -** As an E-type software system evolves, its complexity tends to increase unless work is done to maintain or reduce it.
- **Conservation of familiarity -** The familiarity with the software or the knowledge about how it was developed, why was it developed in that particular manner etc. must be retained at any cost, to implement the changes in the system.
- **Continuing growth-** In order for an E-type system intended to resolve some business problem, its size of implementing the changes grows according to the lifestyle changes of the business.
- **Reducing quality -** An E-type software system declines in quality unless rigorously maintained and adapted to a changing operational environment.
- **Feedback systems-** The E-type software systems constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved.
- **Self-regulation -** E-type system evolution processes are self-regulating with the distribution of product and process measures close to normal.
- **Organizational stability -** The average effective global activity rate in an evolving E-type system is invariant over the lifetime of the product.

Software Paradigms

Software paradigms refer to the methods and steps, which are taken while designing the software. There are many methods proposed and are in work today, but we need to see where in the software engineering these paradigms stand. These can be combined into various categories, though each of them is contained in one another:

Programming paradigm is a subset of Software design paradigm which is further a subset of Software development paradigm.

Software Development Paradigm:

This Paradigm is known as software engineering paradigms where all the engineering concepts pertaining to the development of software are applied. It includes various researches and requirement gathering which helps the software product to build. It consists of –

- Requirement gathering
- Software design
- Programming

Software Design Paradigm

This paradigm is a part of Software Development and includes –

- Design
- Maintenance
- Programming

Programming Paradigm

This paradigm is related closely to programming aspect of software development. This includes –

- Coding
- Testing
- Integration

Need of Software Engineering

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

- **Large software -** It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.

- **Scalability-** If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.
- **Cost-** As hardware industry has shown its skills and huge manufacturing has lower down,the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.
- **Dynamic Nature-** The always growing and adapting nature of software hugely depends upon the environment in which user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.
- **Quality Management-** Better process of software development provides better and quality software product.

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:

- Operational
- Transitional
- Maintenance

Well-engineered and crafted software is expected to have the following characteristics:

Operational

This tells us how well software works in operations. It can be measured on:

- Budget
- Usability
- Efficiency
- Correctness
- Functionality
- Dependability
- Security
- Safety

Transitional

This aspect is important when the software is moved from one platform to another:

- Portability
- Interoperability
- Reusability
- Adaptability

Maintenance

This aspect briefs about how well a software has the capabilities to maintain itself in the ever-changing environment:

- Modularity
- Maintainability
- Flexibility
- Scalability

In short, Software engineering is a branch of computer science, which uses well-defined engineering concepts required to produce efficient, durable, scalable, in-budget and on-time software products.

Software Development Lifecycle (SDLC)

It is a systematic process for building software that ensures the quality and correctness of the software built. SDLC process aims to produce high-quality software that meets customer expectations. The system development should be complete in the pre-defined time frame and cost. SDLC consists of a detailed plan which explains how to plan, build, and maintain specific software. Every phase of the SDLC life cycle has its own process and deliverables that feed into the next phase. SDLC stands for Software Development Lifecycle.

Here, are prime reasons why SDLC is important for developing a software system.

- It offers a basis for project planning, scheduling, and estimating
- Provides a framework for a standard set of activities and deliverables
- It is a mechanism for project tracking and control

- Increases visibility of project planning to all involved stakeholders of the development process
- Increased and enhance development speed
- Improved client relations
- Helps you to decrease project risk and project management plan overhead

SDLC Phases

The entire SDLC process divided into the following stages:

Fig 1.2: Software Development Life Cycle Phases

- Phase 1: Requirement collection and analysis
- Phase 2: Feasibility study:
- Phase 3: Design:
- Phase 4: Coding:
- Phase 5: Testing:
- Phase 6: Installation/Deployment:
- Phase 7: Maintenance:

In this tutorial, I have explained all these phases

Phase 1: Requirement collection and analysis:

The requirement is the first stage in the SDLC process. It is conducted by the senior team members with inputs from all the stakeholders and domain experts in the industry. Planning for the quality assurance requirements and recognition of the risks involved is also done at this stage.

This stage gives a clearer picture of the scope of the entire project and the anticipated issues, opportunities, and directives which triggered the project.

Requirements Gathering stage need teams to get detailed and precise requirements. This helps companies to finalize the necessary timeline to finish the work of that system.

Phase 2: Feasibility study:

Once the requirement analysis phase is completed the next step is to define and document software needs. This process conducted with the help of 'Software Requirement Specification' document also known as 'SRS' document. It includes everything which should be designed and developed during the project life cycle.

There are mainly five types of feasibilities checks:

- **Economic:** Can we complete the project within the budget or not?
- **Legal:** Can we handle this project as cyber law and other regulatory framework/compliances.
- **Operation feasibility:** Can we create operations which is expected by the client?
- **Technical:** Need to check whether the current computer system can support the software
- **Schedule:** Decide that the project can be completed within the given schedule or not.

Phase 3: Design:

In this third phase, the system and software design documents are prepared as per the requirement specification document. This helps define overall system architecture.

This design phase serves as input for the next phase of the model.

There are two kinds of design documents developed in this phase:

High-Level Design (HLD)

- Brief description and name of each module
- An outline about the functionality of every module
- Interface relationship and dependencies between modules
- Database tables identified along with their key elements
- Complete architecture diagrams along with technology details

Low-Level Design (LLD)

- Functional logic of the modules
- Database tables, which include type and size
- Complete detail of the interface
- Addresses all types of dependency issues

- Listing of error messages
- Complete input and outputs for every module

Phase 4: Coding:

Once the system design phase is over, the next phase is coding. In this phase, developers start build the entire system by writing code using the chosen programming language. In the coding phase, tasks are divided into units or modules and assigned to the various developers. It is the longest phase of the Software Development Life Cycle process.

In this phase, Developer needs to follow certain predefined coding guidelines. They also need to use programming tools like compiler, interpreters, debugger to generate and implement the code.

Phase 5: Testing:

Once the software is complete, and it is deployed in the testing environment. The testing team starts testing the functionality of the entire system. This is done to verify that the entire application works according to the customer requirement.

During this phase, QA and testing team may find some bugs/defects which they communicate to developers. The development team fixes the bug and send back to QA for a re-test. This process continues until the software is bug-free, stable, and working according to the business needs of that system.

Phase 6: Installation/Deployment:

Once the software testing phase is over and no bugs or errors left in the system then the final deployment process starts. Based on the feedback given by the project manager, the final software is released and checked for deployment issues if any.

Phase 7: Maintenance:

Once the system is deployed, and customers start using the developed system, following 3 activities occur

- Bug fixing - bugs are reported because of some scenarios which are not tested at all
- Upgrade - Upgrading the application to the newer versions of the Software

- Enhancement - Adding some new features into the existing software

The main focus of this SDLC phase is to ensure that needs continue to be met and that the system continues to perform as per the specification mentioned in the first phase.

Waterfall model

The waterfall is a widely accepted SDLC model. In this approach, the whole process of the software development is divided into various phases. In this SDLC model, the outcome of one phase acts as the input for the next phase.

This SDLC model is documentation-intensive, with earlier phases documenting what need be performed in the subsequent phases.

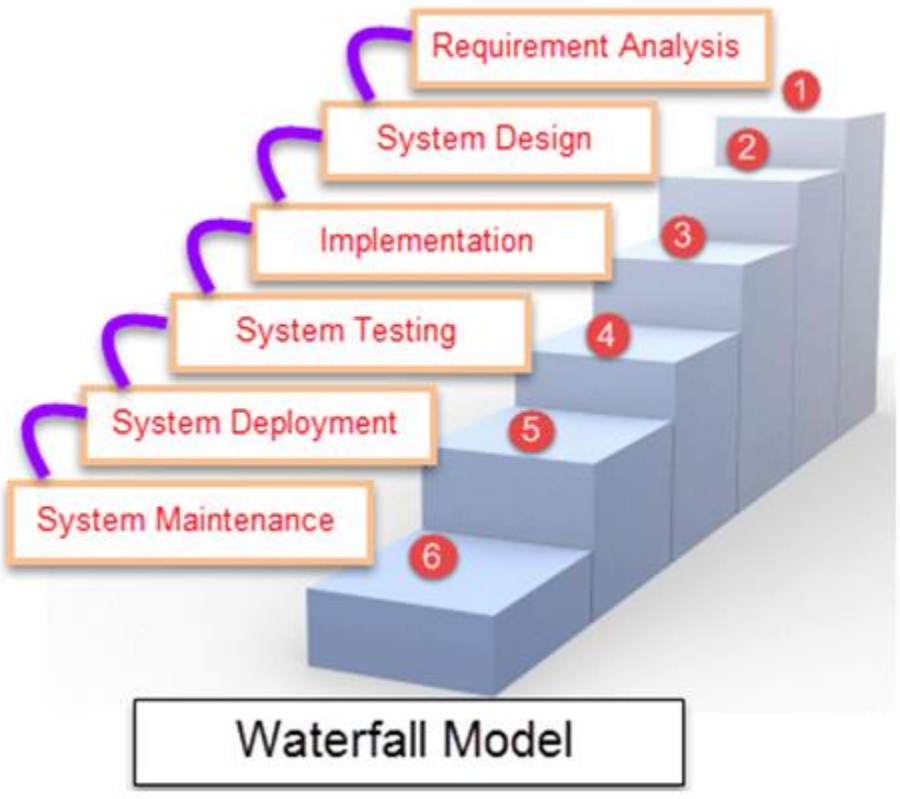


Fig 1.3: Waterfall Model

Different Phases of Waterfall Model in Software Engineering	
Different phases	**Activities performed in each stage**
Requirement Gathering stage	• During this phase, detailed requirements of the software system to be developed are gathered from client
Design Stage	• Plan the programming language, for Example Java, PHP, .net • or database like Oracle, MySQL, etc. • Or other high-level technical details of the project
Built Stage	• After design stage, it is built stage, that is nothing but coding the software
Test Stage	• In this phase, you test the software to verify that it is built as per the specifications given by the client.
Deployment stage	• Deploy the application in the respective environment
Maintenance stage	• Once your system is ready to use, you may later require change the code as per customer request

Table 1.1: Waterfall Model Stages

Waterfall model can be used when

- Requirements are not changing frequently
- Application is not complicated and big
- Project is short
- Requirement is clear
- Environment is stable
- Technology and tools used are not dynamic and is stable
- Resources are available and trained

Advantages and Disadvantages of Waterfall-Model

Advantages	Dis-Advantages
• Before the next phase of development, each phase must be completed	• Error can be fixed only during the phase
• Suited for smaller projects where requirements are well defined	• It is not desirable for complex project where requirement changes frequently
• They should perform quality assurance test (Verification and Validation) before completing each stage	• Testing period comes quite late in the developmental process

Table 1.1.1.: Advantages and Disadvantages of Waterfall Model

Incremental Approach

The incremental model is not a separate model. It is essentially a series of waterfall cycles. The requirements are divided into groups at the start of the project. For each group, the SDLC model is followed to develop software. The SDLC process is repeated, with each release adding more functionality until all requirements are met. In this method, every cycle act as the maintenance phase for the previous software release. Modification to the incremental model allows development cycles to overlap. After that subsequent cycle may begin before the previous cycle is complete.

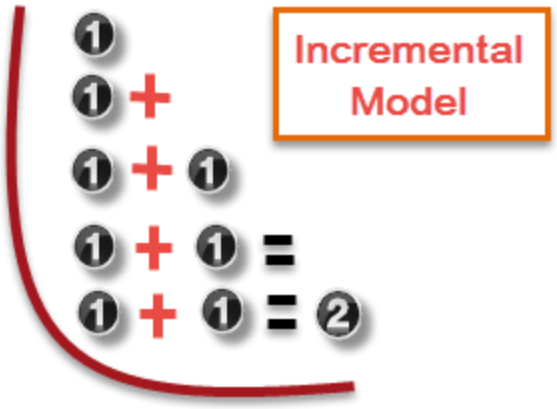


Fig 1.4:Incremental Model

Each iteration passes through the requirements, design, coding and testing phases. And each subsequent release of the system adds function to the previous release until all designed functionality has been implemented.

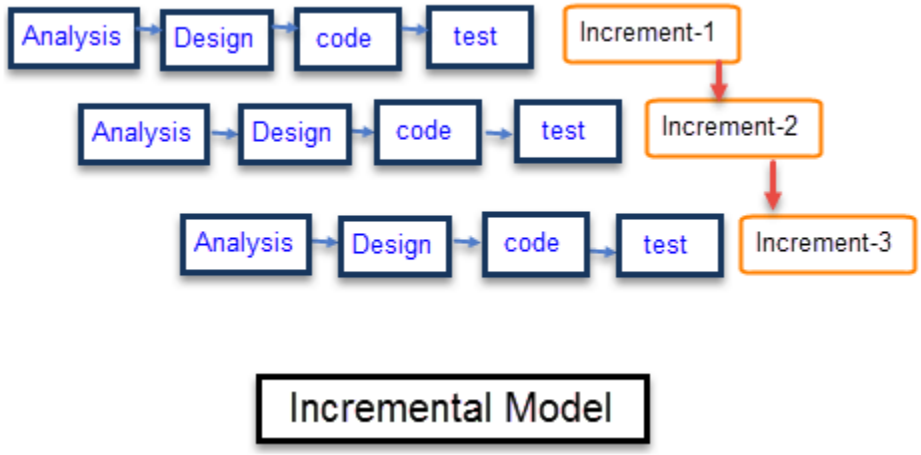


Fig 1.4.1: Incremental levels

The system is put into production when the first increment is delivered. The first increment is often a core product where the basic requirements are addressed, and supplementary features are added in the next increments. Once the core product is analyzed by the client, there is plan development for the next increment.

Characteristics of an Incremental module includes

- System development is broken down into many mini development projects

- Partial systems are successively built to produce a final total system
- Highest priority requirement is tackled first
- Once the requirement is developed, requirement for that increment are frozen

Incremental Phases	Activities performed in incremental phases
Requirement Analysis	• Requirement and specification of the software are collected
Design	• Some high-end function are designed during this stage
Code	• Coding of software is done during this stage
Test	• Once the system is deployed, it goes through the testing phase

Table 1.2: Stages of Incremental Model

When to use 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 and Disadvantages of Incremental Model

Advantages	Disadvantages
• The software will be generated quickly during the software life cycle	• It requires a good planning designing
• It is flexible and less expensive to change requirements and scope	• Problems might cause due to system architecture as such not all requirements collected up front for the entire software lifecycle

Table 1.3: Advantages and Disadvantages of Incremental Model

Spiral Model

The spiral model is a risk-driven process model. This SDLC model helps the team to adopt elements of one or more process models like a waterfall, incremental, waterfall, etc.

This model adopts the best features of the prototyping model and the waterfall model. The spiral methodology is a combination of rapid prototyping and concurrency in design and development activities.

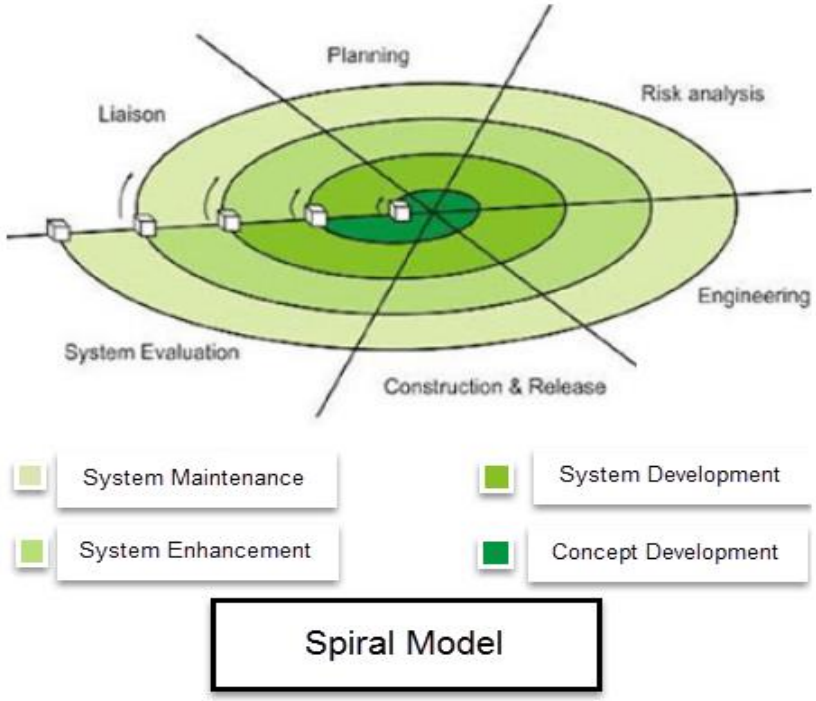


Fig 1.5: Spiral Model

Spiral Model Phases

Spiral Model Phases	Activities performed during phase
Planning	• It includes estimating the cost, schedule and resources for the iteration. It also involves understanding the system requirements for continuous communication between the system analyst and the customer
Risk Analysis	• Identification of potential risk is done while risk mitigation strategy is planned and finalized
Engineering	• It includes testing, coding and deploying software at the customer site
Evaluation	• Evaluation of software by the customer. Also, includes identifying and monitoring risks such as schedule slippage and cost overrun

Table 1.4: Spiral Model Phases

Useof Spiral Methodology:

- When project is large
- When releases are required to be frequent
- When creation of a prototype is applicable
- When risk and costs evaluation is important
- For medium to high-risk projects
- When requirements are unclear and complex
- When changes may require at any time
- When long term project commitment is not feasible due to changes in economic priorities

Advantages and Disadvantages of Spiral Model

Advantages	Disadvantages
• Additional functionality or changes can be done at a later stage	• Risk of not meeting the schedule or budget
• Cost estimation becomes easy as the prototype building is done in small fragments	• It works best for large projects only also demands risk assessment expertise
• Continuous or repeated development helps in risk management	• For its smooth operation spiral model protocol needs to be followed strictly
• Development is fast and features are added in a systematic way	• Documentation is more as it has intermediate phases

Table1.5: Advantages and Disadvantages of Spiral Model

Software Prototyping Model

Prototype methodology is defined as a Software Development model in which a prototype is built, test, and then reworked when needed until an acceptable prototype is achieved. It also creates a base to produce the final system.

Software prototyping model works best in scenarios where the project's requirement are not known. It is an iterative, trial, and error method which take place between the developer and the client.

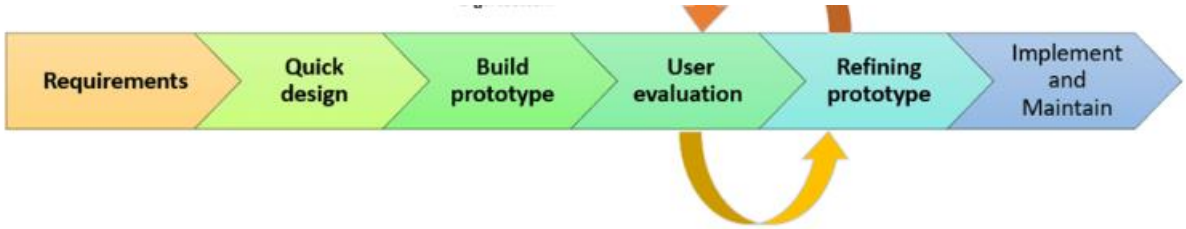


Fig 1.6: Prototyping Model

Prototyping Model has following six SDLC phases as follow:

Step 1: Requirements gathering and analysis

A prototyping model starts with requirement analysis. In this phase, the requirements of the system are defined in detail. During the process, the users of the system are interviewed to know what is their expectation from the system.

Step 2: Quick design

The second phase is a preliminary design or a quick design. In this stage, a simple design of the system is created. However, it is not a complete design. It gives a brief idea of the system to the user. The quick design helps in developing the prototype.

Step 3: Build a Prototype

In this phase, an actual prototype is designed based on the information gathered from quick design. It is a small working model of the required system.

Step 4: Initial user evaluation

In this stage, the proposed system is presented to the client for an initial evaluation. It helps to find out the strength and weakness of the working model. Comment and suggestion are collected from the customer and provided to the developer.

Step 5: Refining prototype

If the user is not happy with the current prototype, you need to refine the prototype according to the user's feedback and suggestions.

This phase will not over until all the requirements specified by the user are met. Once the user is satisfied with the developed prototype, a final system is developed based on the approved final prototype.

Step 6: Implement Product and Maintain

Once the final system is developed based on the final prototype, it is thoroughly tested and deployed to production. The system undergoes routine maintenance for minimizing downtime and prevent large-scale failures.

Types of Prototyping Models

Four types of Prototyping models are:

1. Rapid Throwaway prototypes
2. Evolutionary prototype
3. Incremental prototype
4. Extreme prototype

Rapid Throwaway Prototype

Rapid throwaway is based on the preliminary requirement. It is quickly developed to show how the requirement will look visually. The customer's feedback helps drives changes to the requirement, and the prototype is again created until the requirement is baselined.

In this method, a developed prototype will be discarded and will not be a part of the ultimately accepted prototype. This technique is useful for exploring ideas and getting instant feedback for customer requirements.

Evolutionary Prototyping

Here, the prototype developed is incrementally refined based on customer's feedback until it is finally accepted. It helps you to save time as well as effort. That's because developing a prototype from scratch for every interaction of the process can sometimes be very frustrating.

This model is helpful for a project which uses a new technology that is not well understood. It is also used for a complex project where every functionality must be checked once. It is helpful when the requirement is not stable or not understood clearly at the initial stage.

Incremental Prototyping

In incremental Prototyping, the final product is decimated into different small prototypes and developed individually. Eventually, the different prototypes are merged into a single product. This method is helpful to reduce the feedback time between the user and the application development team.

Extreme Prototyping:

Extreme prototyping method is mostly used for web development. It is consists of three sequential phases.

1. Basic prototype with all the existing page is present in the HTML format.
2. You can simulate data process using a prototype services layer.
3. The services are implemented and integrated into the final prototype.

Best practices of Prototyping

Here, are a few things which you should watch for during the prototyping process:

- You should use Prototyping when the requirements are unclear
- It is important to perform planned and controlled Prototyping.
- Regular meetings are vital to keep the project on time and avoid costly delays.
- The users and the designers should be aware of the prototyping issues and pitfalls.
- At a very early stage, you need to approve a prototype and only then allow the team to move to the next step.
- In software prototyping method, you should never be afraid to change earlier decisions if new ideas need to be deployed.
- You should select the appropriate step size for each version.

- Implement important features early on so that if you run out of the time, you still have a worthwhile system

Advantages of the Prototyping Model

Here, are important pros/benefits of using Prototyping models:

- Users are actively involved in development. Therefore, errors can be detected in the initial stage of the software development process.
- Missing functionality can be identified, which helps to reduce the risk of failure as Prototyping is also considered as a risk reduction activity.
- Helps team member to communicate effectively
- Customer satisfaction exists because the customer can feel the product at a very early stage.
- There will be hardly any chance of software rejection.
- Quicker user feedback helps you to achieve better software development solutions.
- Allows the client to compare if the software code matches the software specification.
- It helps you to find out the missing functionality in the system.
- It also identifies the complex or difficult functions.
- Encourages innovation and flexible designing.
- It is a straightforward model, so it is easy to understand.
- No need for specialized experts to build the model
- The prototype serves as a basis for deriving a system specification.
- The prototype helps to gain a better understanding of the customer's needs.
- Prototypes can be changed and even discarded.
- A prototype also serves as the basis for operational specifications.
- Prototypes may offer early training for future users of the software system.

Disadvantages of the Prototyping Model

Here, are important cons/drawbacks of prototyping model:

- Prototyping is a slow and time taking process.
- The cost of developing a prototype is a total waste as the prototype is ultimately thrown away.
- Prototyping may encourage excessive change requests.
- Sometimes customers may not be willing to participate in the iteration cycle for the longer time duration.

- There may be far too many variations in software requirements when each time the prototype is evaluated by the customer.
- Poor documentation because the requirements of the customers are changing.
- It is very difficult for software developers to accommodate all the changes demanded by the clients.
- After seeing an early prototype model, the customers may think that the actual product will be delivered to him soon.
- The client may lose interest in the final product when he or she is not happy with the initial prototype.
- Developers who want to build prototypes quickly may end up building sub-standard development solutions.

Concurrent Development:

The concurrent development model, sometimes called concurrent engineering, has been described in the following manner by Davis and Sitaram:

Project managers who track project status in terms of the major phases [of the classic life cycle] have no idea of the status of their projects. These are examples of trying to track extremely complex sets of activities using overly simple models. Note that although . . . [a large] project is in the coding phase, there are personnel on the project involved in activities typically associated with many phases of development simultaneously. For example, personnel are writing requirements, designing, coding, testing, and integration testing [all at the same time]. Software engineering process models by Humphrey and Kellner have shown the concurrency that exists for activities occurring during any one phase. Kellner's more recent work uses state charts [a notation that represents the states of a process] to represent the concurrent relationship existent among activities associated with a specific event (e.g., a requirements change during late development), but fails to capture the richness of concurrency that exists across all software development and management activities in the project. . . . Most software development process models are driven by time; the later it is, the later in the development process you are. [A concurrent process model] is driven by user needs, management decisions, and review results.

The concurrent process model can be represented schematically as a series of major technical activities, tasks, and their associated states. For example, the engineering activity defined for the spiral model is accomplished by invoking the

following tasks: prototyping and/or analysis modeling, requirements specification, and design.

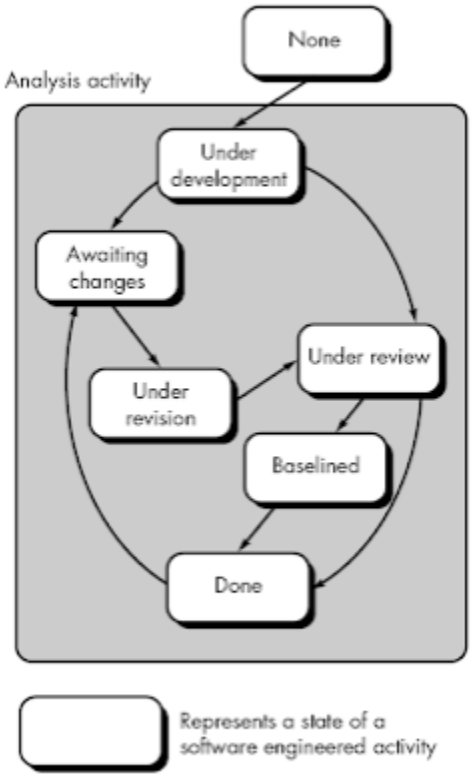


Fig 1.8: Incremental Model

The activity—analysis—may be in any one of the states noted at any given time. Similarly, other activities (e.g., design or customer communication) can be represented in an analogous manner. All activities exist concurrently but reside in different states. For example, early in a project the customer communication activity (not shown in the figure) has completed its first iteration and exists in the awaiting changes state. The analysis activity (which existed in the none state while initial customer communication was completed) now makes a transition into the under-development state. If, however, the customer indicates that changes in requirements must be made, the analysis activity moves from the under-development state into the awaiting changes state.

The concurrent process model defines a series of events that will trigger transitions from state to state for each of the software engineering activities. For example, during early stages of design, an inconsistency in the analysis model is uncovered. This generates the event analysis model correction which will trigger the analysis activity from the done state into the awaiting changes state.

The concurrent process model is often used as the paradigm for the development of client/server applications. A client/server system is composed of a set of functional components. When applied to client/server, the concurrent process model defines activities in two dimensions: a system dimension and a component dimension. System level issues are addressed using three activities: design, assembly, and use. The component dimension is addressed with two activities: design and realization.

Concurrency is achieved in two ways:

(1) system and component activities occur simultaneously and can be modeled using the state-oriented approach described previously;

(2) a typical client/server application is implemented with many components, each of which can be designed and realized concurrently.

In reality, the concurrent process model is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities to a sequence of events, it defines a network of activities. Each activity on the network exists simultaneously with other activities. Events generated within a given activity or at some other place in the activity network trigger transitions among the states of an activity.

Specialized Process Models.

Special process models take on many of the characteristics of one or more of the conventional models. However, specialized models tend to be applied when a narrowly defined software engineering approach is chosen.

Component-Based Development.

Commercial off-the-shelf (COST) software components, developed by vendors who offer them as products, can be used when software is to be built. These components provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software. 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 composes applications from prepackaged software components. Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

• Available component-based products are researched and evaluated for the application domain in question. http://wikistudent.ws/Unisa

• 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 leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Based on studies of reusability component-based development can lead to reduction in development cycle time, reduction in project cost and increase in productivity. Although these results are a function of the robustness of the component library, there is little question that the component-based development model provides significant advantages for software engineers.

The Formal Methods Model.

The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software. Formal methods enable a software engineer to specify, develop, and verify a computer-based system by applying rigorous, mathematical notation. A variation on this approach is called clean-room software engineering. When formal methods are used during development, they provide a mechanism for eliminating many of the problems that are difficult to overcome using other software engineering paradigms. Ambiguity, incompleteness, and inconsistency can be discovered and corrected more easily, not through ad hoc review, but through the application of mathematical analysis. When formal methods are used during design, they serve as a basis for program verification and therefore enable the software engineer to discover and correct errors that might otherwise go undetected. Although not a mainstream approach,

the formal methods model offers the promise of defect-free software. Yet, concern about its applicability in a business environment has been voiced:

• 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 model as a communication mechanism for technically unsophisticated customers.

Aspect-Oriented Software Development:

Regardless of the software process that is chosen, the builders of complex software invariably implement a set of localized features, functions and information content. These localized software characteristics are modeled as components and then constructed within the context of a system architecture. As modern computer-based systems become more sophisticated and complex, certain concerns, customer required properties or areas of technical interest, span the entire architecture. Some concerns are high-level properties of a system; others affect functions or are systemic. When concerns cut across multiple system functions, features, and information they are often referred to as crosscutting concerns. Aspectual requirements define those crosscutting concerns that have impact across the software architecture. Aspects are mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concerns. Aspect-oriented software development (AOSD), often referred to as aspect-oriented programming (AOP), is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing aspects. http://wikistudent.ws/Unisa A distinct aspect-oriented process has not yet matured. However, it is likely that such a process will adopt characteristics of both the spiral and concurrent process models. The evolutionary nature of the spiral is appropriate as aspects are identified and then constructed. The parallel nature of concurrent development is essential because aspects are engineered independently of localized software components and yet, aspects have a direct impact on these components.

The Unified Process.

In some ways the unified process (UP) is an attempt to draw on the best features and characteristics of conventional software process models, but characterize them

in a way that implements many of the best principles of agile software development. The unified process recognizes the importance of customer communication and streamlined methods for describing the customer's view of a system. It emphasizes the important role of software architecture and helps the architect focus on the right goals, such as understandability, reliance to future changes, and reuse. It suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development.

Prescriptive Models or Conventional Models.

Every software engineering organization should describe a unique set of framework activities for the software processes it adopts. It should populate each framework activity with a set of software engineering actions, and define each action in terms of a task set that identifies the work (and work products) to be accomplished to meet the development goals. It should then adapt the resultant process model to accommodate the specific nature of each project, the people who will do the work and the environment in which the work will be conducted. Regardless of the process model that is selected, software engineers have traditionally chosen a generic process framework that encompasses the following framework activities:

• **Communication** – This framework activity involves heavy communication and collaboration with the customer (and other stakeholders) and encompasses requirements gathering and related activities. • Planning – This activity establishes a plan for the software engineering work that follows. It describes the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

• **Modeling** – this activity encompasses the creation of models that allow the developer and the customer to better understand software requirements and the design that will achieve those requirements.

• **Construction** – This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

• **Deployment** – The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation. We call these models prescriptive because they prescribe a set of process elements namely framework activities,

software engineering actions, tasks, work products, quality assurance and change control mechanisms for each project. Each process model also prescribes a workflow that is, the manner in which the process elements are interrelated to one another. All software process models can accommodate the generic framework activities, but each applies a different emphasis to these activities and defines a workflow that invokes each framework activity (as well as software engineering actions and tasks) in a different manner.

Verification And Validation:

Verification:

The verifying process includes checking documents, design, code, and program.

Validation:

Validation is a dynamic mechanism of Software testing and validates the actual product.

Verification	Validation
• The verifying process includes checking documents, design, code, and program	• It is a dynamic mechanism of testing and validating the actual product
• It does ***not*** involve executing the code	• It always involves executing the code
• Verification uses methods like reviews, walkthroughs, inspections, and desk-checking etc.	• It uses methods like Black Box Testing, White Box Testing, and non-functional testing
• Whether the software conforms to specification is checked	• It checks whether the software meets the requirements and expectations of a customer
• It finds bugs early in the development cycle	• It can find bugs that the verification process can not catch
• Target is application and software architecture, specification, complete design, high level, and database design etc.	• Target is an actual product
• QA team does verification and make sure that the software is as per the requirement in the SRS document.	• With the involvement of testing team validation is executed on software code.
• It comes before validation	• It comes after verification

Table 1.7: Difference of Verification and Validation

Example of verification and validation

- In Software Engineering, consider the following specification

A clickable button with name Submet

- Verification would check the design doc and correcting the spelling mistake.
- Otherwise, the development team will create a button like

  - So new specification is

A clickable button with name Submit

- Once the code is ready, Validation is done. A Validation test found –

- Owing to Validation testing, the development team will make the submit button clickable.

V-Model is also known as Verification and Validation Model. In this model Verification & Validation goes hand in hand i.e. development and testing goes parallel. V model and waterfall model are the same except that the test planning and testing start at an early stage in V-Model.

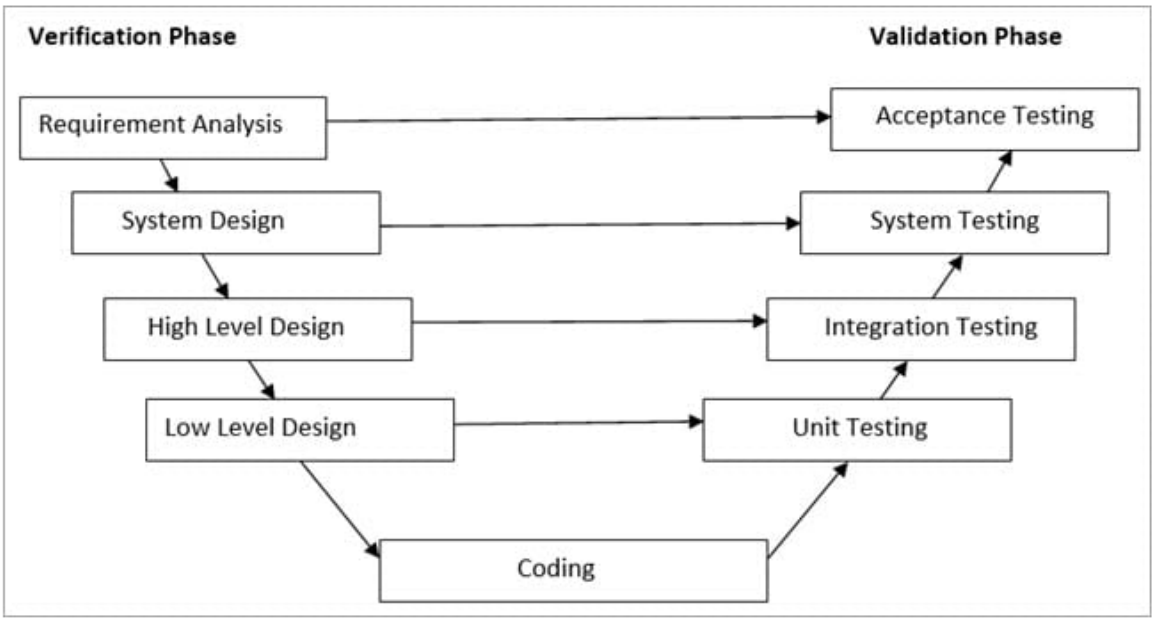


Fig 1.9: V -Model

a) Verification Phase:

(i) Requirement Analysis:

In this phase, all the required information is gathered & analyzed. Verification activities include reviewing the requirements.

(ii) System Design:

Once the requirement is clear, a system is designed i.e. architecture, components of the product are created and documented in a design document.

(iii) High-Level Design:

High-level design defines the architecture/design of modules. It defines the functionality between the two modules.

(iv) Low-Level Design:

Low-level Design defines the architecture/design of individual components.

(v) Coding:

Code development is done in this phase.

b) Validation Phase:

(i) Unit Testing:

Unit testing is performed using the unit test cases that are designed and is done in the Low-level design phase. Unit testing is performed by the developer itself. It is performed on individual components which lead to early defect detection.

(ii) Integration Testing:

Integration testing is performed using integration test cases in High-level Design phase. Integration testing is the testing that is done on integrated modules. It is performed by testers.

(iii) System Testing:

System testing is performed in the System Design phase. In this phase, the complete system is tested i.e. the entire system functionality is tested.

(iv) Acceptance Testing:

Acceptance testing is associated with the Requirement Analysis phase and is done in the customer's environment.

Advantages of V – Model:

- It is a simple and easily understandable model.
- V –model approach is good for smaller projects wherein the requirement is defined and it freezes in the early stage.
- It is a systematic and disciplined model which results in a high-quality product.

Disadvantages of V-Model:

- V-shaped model is not good for ongoing projects.
- Requirement change at the later stage would cost too high.

The System Engineering Hierarchy - System Modeling

Good system engineering begins with a clear understanding of context - the world view - and then progressively narrows focus until technical details are understood. System engineering encompasses a collection of top-down and bottom-up methods to navigate the hierarchy.

System engineering process begins with a world of view which is refined to focus more fully on a specific domain of interest. Within a specific domain, the need for targeted system elements is analyzed. Finally, the analysis, design, and construction of targeted system element is initiated. Broad context is established at the top of the hierarchy and at the bottom, detailed technical activities are conducted. It is important for a system engineer narrows the focus of work as one moves downward in the hierarchy.

System modeling is an important element of system engineering process. System engineering model accomplishes the following:

- define processes.

- represent behavior of the process.

- define both exogenous and endogenous input to model.

- represent all linkages.

Some restraining factors that are considered to construct a system model are:

- Assumptions that reduce number of possible permutations and variations thus enabling a model to reflect the problem in a reasonable manner.

- Simplifications that enable the model to be created in a timely manner.

- Limitations that help to bound the system.

- Constraints that will guide the manner in which the model is created and the approach taken when the model is implemented.

- Preferences that indicate the preferred architecture for all data, functions, and technology.

The resultant system model may call for a completely automated or semi-automated or a non-automated solution.

Software engineering lifecycle process:

Software Development Lifecycle (SDLC)

It is a systematic process for building software that ensures the quality and correctness of the software built. SDLC process aims to produce high-quality software that meets customer expectations. The system development should be complete in the pre-defined time frame and cost. SDLC consists of a detailed plan which explains how to plan, build, and maintain specific software. Every phase of the SDLC life cycle has its own process and deliverables that feed into the next phase. SDLC stands for Software Development Lifecycle.

Here, are prime reasons why SDLC is important for developing a software system.

- It offers a basis for project planning, scheduling, and estimating
- Provides a framework for a standard set of activities and deliverables
- It is a mechanism for project tracking and control
- Increases visibility of project planning to all involved stakeholders of the development process
- Increased and enhance development speed
- Improved client relations
- Helps you to decrease project risk and project management plan overhead

SDLC Phases

The entire SDLC process divided into the following stages:

- Phase 1: Requirement collection and analysis
- Phase 2: Feasibility study:
- Phase 3: Design:
- Phase 4: Coding:
- Phase 5: Testing:
- Phase 6: Installation/Deployment:
- Phase 7: Maintenance:

Phase 1: Requirement collection and analysis:

The requirement is the first stage in the SDLC process. It is conducted by the senior team members with inputs from all the stakeholders and domain experts in the industry. Planning for the quality assurance requirements and recognition of the risks involved is also done at this stage.

This stage gives a clearer picture of the scope of the entire project and the anticipated issues, opportunities, and directives which triggered the project.

Requirements Gathering stage need teams to get detailed and precise requirements. This helps companies to finalize the necessary timeline to finish the work of that system.

Phase 2: Feasibility study:

Once the requirement analysis phase is completed the next step is to define and document software needs. This process conducted with the help of 'Software Requirement Specification' document also known as 'SRS' document. It includes everything which should be designed and developed during the project life cycle.

There are mainly five types of feasibilities checks:

- **Economic:** Can we complete the project within the budget or not?
- **Legal:** Can we handle this project as cyber law and other regulatory framework/compliances.
- **Operation feasibility:** Can we create operations which is expected by the client?
- **Technical:** Need to check whether the current computer system can support the software
- **Schedule:** Decide that the project can be completed within the given schedule or not.

Phase 3: Design:

In this third phase, the system and software design documents are prepared as per the requirement specification document. This helps define overall system architecture.

This design phase serves as input for the next phase of the model.

There are two kinds of design documents developed in this phase:

High-Level Design (HLD)

- Brief description and name of each module
- An outline about the functionality of every module
- Interface relationship and dependencies between modules
- Database tables identified along with their key elements
- Complete architecture diagrams along with technology details

Low-Level Design (LLD)

- Functional logic of the modules
- Database tables, which include type and size
- Complete detail of the interface
- Addresses all types of dependency issues

- Listing of error messages
- Complete input and outputs for every module

Phase 4: Coding:

Once the system design phase is over, the next phase is coding. In this phase, developers start build the entire system by writing code using the chosen programming language. In the coding phase, tasks are divided into units or modules and assigned to the various developers. It is the longest phase of the Software Development Life Cycle process.

In this phase, Developer needs to follow certain predefined coding guidelines. They also need to use programming tools like compiler, interpreters, debugger to generate and implement the code.

Phase 5: Testing:

Once the software is complete, and it is deployed in the testing environment. The testing team starts testing the functionality of the entire system. This is done to verify that the entire application works according to the customer requirement.

During this phase, QA and testing team may find some bugs/defects which they communicate to developers. The development team fixes the bug and send back to QA for a re-test. This process continues until the software is bug-free, stable, and working according to the business needs of that system.

Phase 6: Installation/Deployment:

Once the software testing phase is over and no bugs or errors left in the system then the final deployment process starts. Based on the feedback given by the project manager, the final software is released and checked for deployment issues if any.

Phase 7: Maintenance:

Once the system is deployed, and customers start using the developed system, following 3 activities occur

- Bug fixing - bugs are reported because of some scenarios which are not tested at all
- Upgrade - Upgrading the application to the newer versions of the Software

- Enhancement - Adding some new features into the existing software

The main focus of this SDLC phase is to ensure that needs continue to be met and that the system continues to perform as per the specification mentioned in the first phase.

Capability Maturity Model:

Capability Maturity Model is used as a benchmark to measure the maturity of an organization's software process.

CMM was developed at the Software engineering institute in the late 80's. It was developed as a result of a study financed by the U.S Air Force as a way to evaluate the work of subcontractors. Later based on the CMM-SW model created in 1991 to assess the maturity of software development, multiple other models are integrated with CMM-I they are

Capability Maturity Model (CMM) Levels:

1. Initial
2. Repeatable/Managed
3. Defined
4. Quantitatively Managed
5. Optimizing

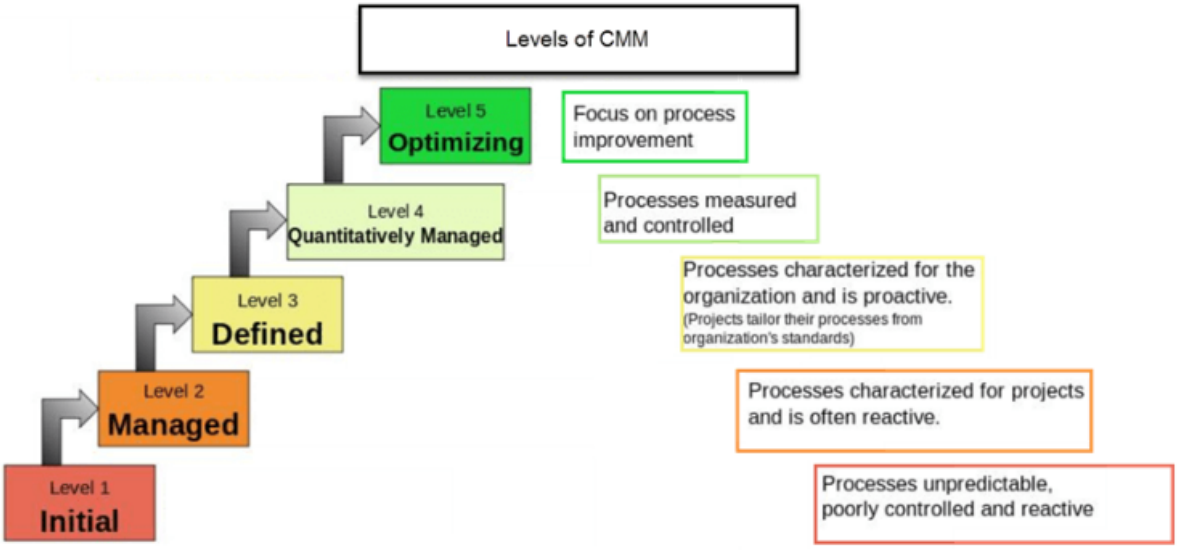


Fig1.10: CMM Levels

Levels	Activities	Benefits
Level 1 Initial	• At level 1, the process is usually chaotic and ad hoc • A capability is characterized on the basis of the individuals and not of the organization • Progress not measured • Products developed are often schedule and over budget • Wide variations in the schedule, cost, functionality, and quality targets	None. A project is Total Chaos

Level 2 Managed	• Requirement Management • Estimate project parameters like cost, schedule, and functionality • Measure actual progress • Develop plans and process • Software project standards are defined • Identify and control products, problem reports changes, etc. • Processes may differ between projects	• Processes become easier to comprehend • Managers and team members spend less time in explaining how things are done and more time in executing it • Projects are better estimated, better planned and more flexible • Quality is integrated into projects • Costing might be high initially but goes down overtime • Ask more paperwork and documentation
Level-3 Defined	• Clarify customer requirements • Solve design requirements, develop an implementation process • Makes sure that product meets the requirements and intended use • Analyze decisions systematically • Rectify and control potential problems	• Process Improvement becomes the standard • Solution progresses from being "coded" to being "engineered" • Quality gates appear throughout the project effort with the entire team involved in the process • Risks are mitigated and don't take the team by surprise
Level-4 Quantitatively Managed	• Manages the project's processes and sub-processes statistically • Understand process performance, quantitatively manage the organization's project	• Optimizes Process Performance across the organization • Fosters Quantitative Project Management in an organization.
Level-5 Optimizing	• Detect and remove the cause of defects early • Identify and deploy new tools and process improvements to meet needs and business objectives	• Fosters Organizational Innovation and Deployment • Gives impetus to Causal Analysis and Resolution

Table 1.8: Description of Levels of CMM

Limitations of CMM Models

- CMM determines what a process should address instead of how it should be implemented
- It does not explain every possibility of software process improvement
- It concentrates on software issues but does not consider strategic business planning, adopting technologies, establishing product line and managing human resources
- It does not tell on what kind of business an organization should be in
- CMM will not be useful in the project having a crisis right now

Use of CMM:

Today CMM act as a "seal of approval" in the software industry. It helps in various ways to improve the software quality.

- It guides towards repeatable standard process and hence reduce the learning time on how to get things done
- Practicing CMM means practicing standard protocol for development, which means it not only helps the team to save time but also gives a clear view of what to do and what to expect
- The quality activities gel well with the project rather than thought of as a separate event
- It acts as a commuter between the project and the team
- CMM efforts are always towards the improvement of the process

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – II – Software Engineering – SCS1305

SOFTWARE ENGINEERING PROCESS

Functional And Non-Functional - User - System - Requirement Engineering Process - Feasibility Studies - Requirements - Elicitation - Validation and management - Fundamental of requirement analysis - Analysis principles- Software prototyping - Prototyping in the Software Process - Rapid Prototyping Techniques - User Interface Prototyping - Software Document Analysis and Modeling - Data - Functional and Behavioral Models - Structured Analysis and Data Dictionary.

Functional Requirement:

In software engineering, a functional requirement defines a system or its component. It describes the functions a software must perform. A function is nothing but inputs, its behavior, and outputs. It can be a calculation, data manipulation, business process, user interaction, or any other specific functionality which defines what function a system is likely to perform.

Functional software requirements help you to capture the intended behavior of the system. This behavior may be expressed as functions, services or tasks or which system is required to perform.

Example of Functional Requirements

- The software automatically validates customers against the ABC Contact Management System
- The Sales system should allow users to record customers sales
- The background color for all windows in the application will be blue and have a hexadecimal RGB color value of 0x0000FF.
- Only Managerial level employees have the right to view revenue data.

Non-Functional Requirement:

A non-functional requirement defines the quality attribute of a software system. They represent a set of standards used to judge the specific operation of a system. Example, how fast does the website load?

A non-functional requirement is essential to ensure the usability and effectiveness of the entire software system. Failing to meet non-functional requirements can result in systems that fail to satisfy user needs.

Non-functional Requirements allows you to impose constraints or restrictions on the design of the system across the various agile backlogs. Example, the site should load in 3 seconds when the number of simultaneous users are> 10000. Description of non-functional requirements is just as critical as a functional requirement.

- The software system should be integrated with banking API
- The software system should pass Section 508 accessibility requirement.

Examples of Non-functional requirements

Here, are some examples of non-functional requirement:

1. Users must change the initially assigned login password immediately after the first successful login. Moreover, the initial should never be reused.
2. Employees never allowed to update their salary information. Such attempt should be reported to the security administrator.
3. Every unsuccessful attempt by a user to access an item of data shall be recorded on an audit trail.
4. A website should be capable enough to handle 20 million users with affecting its performance
5. The software should be portable. So moving from one OS to other OS does not create any problem.
6. Privacy of information, the export of restricted technologies, intellectual property rights, etc. should be audited.

Functional vs Non Functional Requirements

Parameters	Functional Requirement	Non-Functional Requirement
What it is	Verb	Attributes
Requirement	It is mandatory	It is non-mandatory
Capturing type	It is captured in use case.	It is captured as a quality attribute.
End-result	Product feature	Product properties
Capturing	Easy to capture	Hard to capture
Objective	Helps you verify the functionality of the software.	Helps you to verify the performance of the software.
Area of focus	Focus on user requirement	Concentrates on the user's expectation.
Documentation	Describe what the product does	Describes how the product works
Type of Testing	Functional Testing like System, Integration, End to End, API testing,	Non-Functional Testing like Performance, Stress, Usability, Security testing, etc.

Test Execution	Test Execution is done before non-functional testing.	After the functional testing
Product Info	Product Features	Product Properties

Table 2.1 Functional vs Non Functional Requirements

Advantages of Functional Requirement

Here, are the pros/advantages of creating a typical functional requirement document-

- Helps you to check whether the application is providing all the functionalities that were mentioned in the functional requirement of that application
- A functional requirement document helps you to define the functionality of a system or one of its subsystems.
- Functional requirements along with requirement analysis help identify missing requirements. They help clearly define the expected system service and behavior.
- Errors caught in the Functional requirement gathering stage are the cheapest to fix.
- Support user goals, tasks, or activities for easy project management
- Functional requirement can be expressed in Use Case form or user story as they exhibit externally visible functional behavior.

Advantages of Non-Functional Requirement:

Benefits/pros of Non-functional testing are:

- The nonfunctional requirements ensure the software system follow legal and compliance rules.
- They ensure the reliability, availability, and performance of the software system
- They ensure good user experience and ease of operating the software.
- They help in formulating security policy of the software system.

User requirements:

User requirements, often referred to as user needs, describe what the user does with the system, such as what activities that users must be able to perform. User requirements are generally documented in a User Requirements Document (URD) using narrative text. User requirements are generally signed off by the user and used as the primary input for creating system requirements.

An important and difficult step of designing a software product is determining what the user actually wants it to do. This is because the user is often not able to communicate the entirety of their needs and wants, and the information they provide may also be incomplete, inaccurate and

self-conflicting. The responsibility of completely understanding what the customer wants falls on the business analyst. This is why user requirements are generally considered separately from system requirements. The business analyst carefully analyzes user requirements and carefully constructs and documents a set of high quality system requirements ensuring that that the requirements meet certain quality characteristics.

A system is said be good if it provides means to use it efficiently. User interface requirements are briefly mentioned below -

- Content presentation
- Easy Navigation
- Simple interface
- Responsive
- Consistent UI elements
- Feedback mechanism
- Default settings
- Purposeful layout
- Strategical use of color and texture.
- Provide help information
- User centric approach
- Group based view settings.

System Requirements:

System requirements are the building blocks developers use to build the system. These are the traditional "shall" statements that describe what the system "shall do." System requirements are classified as either functional or supplemental requirements. A functional requirement specifies something that a user needs to perform their work. For example, a system may be required to enter and print cost estimates; this is a functional requirement. Supplemental or non-functional requirements specify all the remaining requirements not covered by the functional requirements. I prefer to use the term supplemental requirements instead of non-functional requirements; who wants to be termed non-functional. Supplemental requirements are sometimes called quality of service requirements. The plan for implementing functional requirements is detailed in the system design. The plan for implementing supplemental requirements is detailed in the system architecture. The list below shows various types of supplemental requirements.

- Accessibility
- Accuracy
- Audit, control, and reporting
- Availability
- Backup and restore

- Capacity, current and forecast
- Certification
- Compliance
- Compatibility of software, tools, standards, platform, database, and the like
- Concurrency
- Configuration management
- Dependency on other parties
- Deployment
- Documentation
- Disaster recovery
- Efficiency (resource consumption for given load)
- Effectiveness (resulting performance in relation to effort)
- Emotional factors (like fun or absorbing)
- Environmental protection
- Error handling
- Escrow
- Exploitability
- Extensibility (adding features, and carry-forward of customizations at next major version upgrade)
- Failure management
- Interoperability
- Legal and regulatory
- Licensing
- Localizability
- Maintainability
- Modifiability
- Network topology
- Open source
- Operability
- Performance/response time
- Price
- Privacy
- Portability

- Quality
- Recovery/recoverability
- Redundancy

A solution may contain only software components, or it could incorporate both software and hardware components. It may also include training and organizational change requirements that will be provided to instructional designers to develop a comprehensive training program. Software requirements represent the system's functional and supplemental requirements that define the software components of the system.

Traditionally, functional and supplemental requirements for software were documented in the software requirements specification (SRS). The SRS is the principal deliverable that analysts use to communicate detailed requirements information to developers, testers, and other project stakeholders. Many modern systems, such as the Enfocus Requirement Suite,™ place requirements into multiple requirement bundles to support iterative development and to break requirements into various components required by teams to develop the solution (e.g., software, hardware, organizational change, etc.) Requirement bundles provide more flexibility than URD and SRS. Requirement bundles may be developed iteratively and incrementally with both stakeholders and project team members being able to validate and review the requirements as they evolve. User needs are always mapped to system requirements so that the developer sees not only the requirement but the source from where it originated.

Requirement Engineering:

The process to gather the software requirements from client, analyze and document them is known as requirement engineering.

The goal of requirement engineering is to develop and maintain sophisticated and descriptive 'System Requirements Specification' document.

Requirements engineering (RE) refers to the process of defining, documenting, and maintaining requirements in the engineering design process. Requirement engineering provides the appropriate mechanism to understand what the customer desires, analyzing the need, and assessing feasibility, negotiating a reasonable solution, specifying the solution clearly, validating the specifications and managing the requirements as they are transformed into a working system. Thus, requirement engineering is the disciplined application of proven

principles, methods, tools, and notation to describe a proposed system's intended behavior and its associated constraints.

Requirement Engineering Process

It is a four step process, which includes –

- Feasibility Study
- Requirement Gathering
- Software Requirement Specification
- Software Requirement Validation

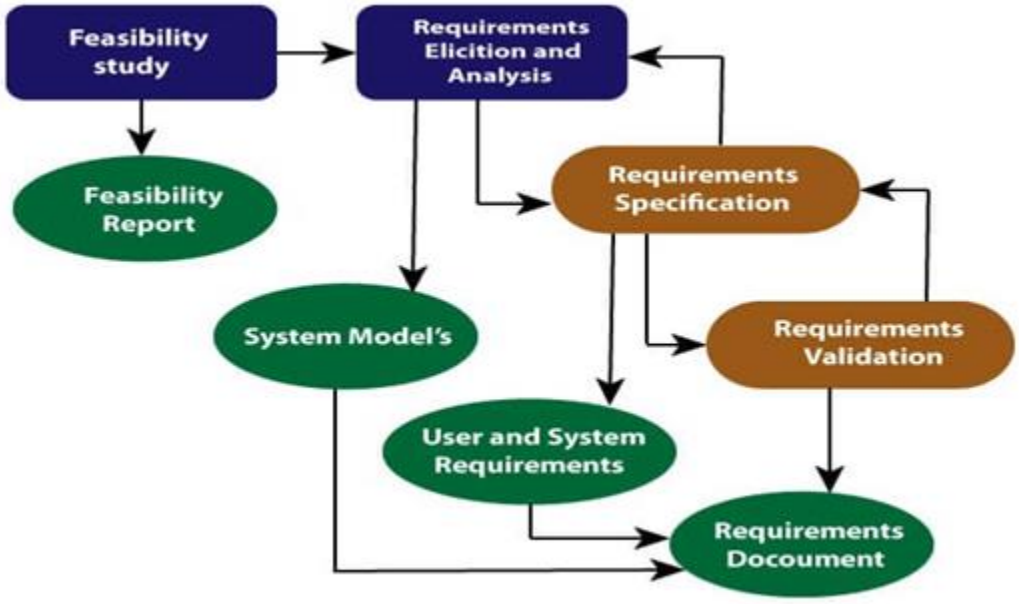


Fig 2.1- Requirement Engineering Process

Feasibility study:

The objective behind the feasibility study is to create the reasons for developing the software that is acceptable to users, flexible to change and conformable to established standards.

Types of Feasibility:

Technical Feasibility - Technical feasibility evaluates the current technologies, which are needed to accomplish customer requirements within the time and budget.

Operational Feasibility - Operational feasibility assesses the range in which the required software performs a series of levels to solve business problems and customer requirements.

Economic Feasibility - Economic feasibility decides whether the necessary software can generate financial profits for an organization.

When the client approaches the organization for getting the desired product developed, it comes up with rough idea about what all functions the software must perform and which all features are expected from the software.

Referencing to this information, the analysts does a detailed study about whether the desired system and its functionality are feasible to develop.

This feasibility study is focused towards goal of the organization. This study analyzes whether the software product can be practically materialized in terms of implementation, contribution of project to organization, cost constraints and as per values and objectives of the organization. It explores technical aspects of the project and product such as usability, maintainability, productivity and integration ability.

The output of this phase should be a feasibility study report that should contain adequate comments and recommendations for management about whether or not the project should be undertaken.

Requirement Gathering:

If the feasibility report is positive towards undertaking the project, next phase starts with gathering requirements from the user. Analysts and engineers communicate with the client and end-users to know their ideas on what the software should provide and which features they want the software to include.

Software Requirement Specification

SRS is a document created by system analyst after the requirements are collected from various stakeholders.

SRS defines how the intended software will interact with hardware, external interfaces, speed of operation, response time of system, portability of software across various platforms, maintainability, speed of recovery after crashing, Security, Quality, Limitations etc.

The requirements received from client are written in natural language. It is the responsibility of system analyst to document the requirements in technical language so that they can be comprehended and useful by the software development team.

SRS should come up with following features:

- User Requirements are expressed in natural language.
- Technical requirements are expressed in structured language, which is used inside the organization.
- Design description should be written in Pseudo code.
- Format of Forms and GUI screen prints.

- Conditional and mathematical notations for DFDs etc.

Software Requirement Validation

After requirement specifications are developed, the requirements mentioned in this document are validated. User might ask for illegal, impractical solution or experts may interpret the requirements incorrectly. This results in huge increase in cost if not nipped in the bud. Requirements can be checked against following conditions -

- If they can be practically implemented
- If they are valid and as per functionality and domain of software
- If there are any ambiguities
- If they are complete
- If they can be demonstrated

Requirement Elicitation Process

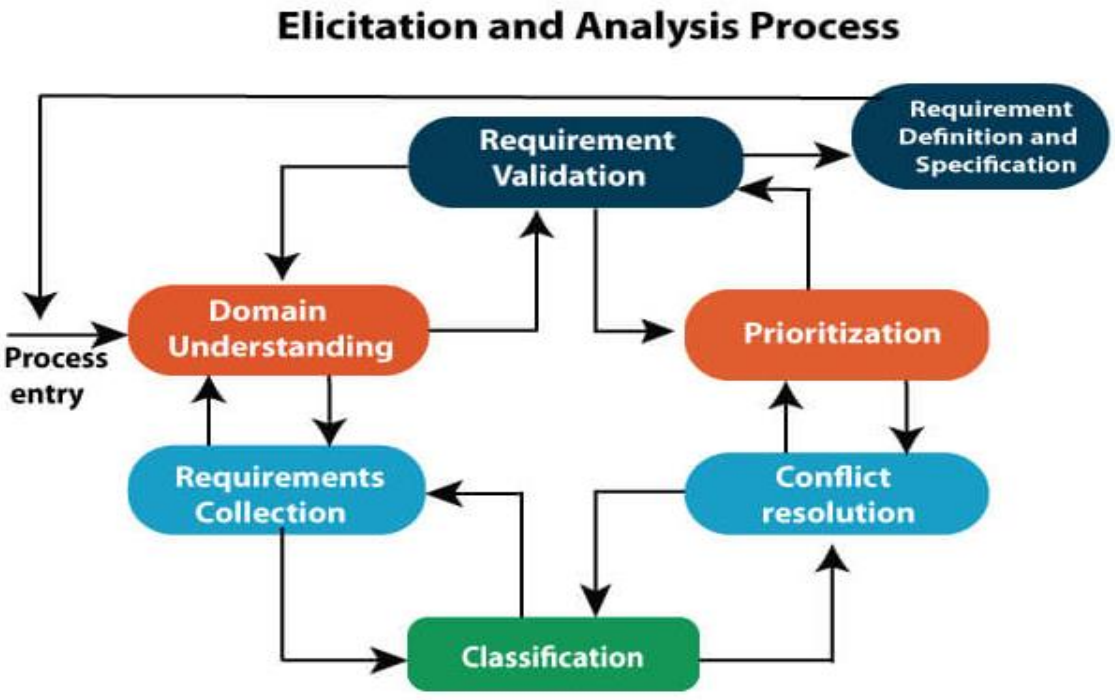


Fig 2.2 Elicitation and Analysis process

Requirement elicitation process can be depicted using the following diagram:

- **Requirements gathering -** The developers discuss with the client and end users and know their expectations from the software.
- **Organizing Requirements -** The developers prioritize and arrange the requirements in order of importance, urgency and convenience.
- **Negotiation & discussion -** If requirements are ambiguous or there are some conflicts in requirements of various stakeholders, if they are, it is then negotiated and discussed with stakeholders. Requirements may then be prioritized and reasonably compromised.

  The requirements come from various stakeholders. To remove the ambiguity and conflicts, they are discussed for clarity and correctness. Unrealistic requirements are compromised reasonably.

- **Documentation -** All formal & informal, functional and non-functional requirements are documented and made available for next phase processing.

Requirement Elicitation Techniques

Requirements Elicitation is the process to find out the requirements for an intended software system by communicating with client, end users, system users and others who have a stake in the software system development.

There are various ways to discover requirements

Interviews

Interviews are strong medium to collect requirements. Organization may conduct several types of interviews such as:

- Structured (closed) interviews, where every single information to gather is decided in advance, they follow pattern and matter of discussion firmly.
- Non-structured (open) interviews, where information to gather is not decided in advance, more flexible and less biased.
- Oral interviews
- Written interviews
- One-to-one interviews which are held between two persons across the table.
- Group interviews which are held between groups of participants. They help to uncover any missing requirement as numerous people are involved.

Surveys

Organization may conduct surveys among various stakeholders by querying about their expectation and requirements from the upcoming system.

Questionnaires

A document with pre-defined set of objective questions and respective options is handed over to all stakeholders to answer, which are collected and compiled.

A shortcoming of this technique is, if an option for some issue is not mentioned in the questionnaire, the issue might be left unattended.

Task analysis

Team of engineers and developers may analyze the operation for which the new system is required. If the client already has some software to perform certain operation, it is studied and requirements of proposed system are collected.

Domain Analysis

Every software falls into some domain category. The expert people in the domain can be a great help to analyze general and specific requirements.

Brainstorming

An informal debate is held among various stakeholders and all their inputs are recorded for further requirements analysis.

Prototyping

Prototyping is building user interface without adding detail functionality for user to interpret the features of intended software product. It helps giving better idea of requirements. If there is no software installed at client's end for developer's reference and the client is not aware of its own requirements, the developer creates a prototype based on initially mentioned requirements. The prototype is shown to the client and the feedback is noted. The client feedback serves as an input for requirement gathering.

Observation

Team of experts visit the client's organization or workplace. They observe the actual working of the existing installed systems. They observe the workflow at client's end and how execution problems are dealt. The team itself draws some conclusions which aid to form requirements expected from the software.

Software Requirements Characteristics

Gathering software requirements is the foundation of the entire software development project. Hence they must be clear, correct and well-defined.

A complete Software Requirement Specifications must be:

- Clear
- Correct
- Consistent

- Coherent
- Comprehensible
- Modifiable
- Verifiable
- Prioritized
- Unambiguous
- Traceable
- Credible source

Validation and Management:

Requirements validation is the process of checking that requirements defined for development, define the system that the customer really wants. To check issues related to requirements, we perform requirements validation. We usually use requirements validation to check error at the initial phase of development as the error may increase excessive rework when detected later in the development process.

In the requirements validation process, we perform a different type of test to check the requirements mentioned in the Software Requirements Specification (SRS), these checks include:

- Completeness checks
- Consistency checks
- Validity checks
- Realism checks
- Ambiguity checks
- Verifiability

The output of requirements validation is the list of problems and agreed on actions of detected problems. The lists of problems indicate the problem detected during the process of requirement validation. The list of agreed action states the corrective action that should be taken to fix the detected problem.

There are several techniques which are used either individually or in conjunction with other techniques to check to check entire or part of the system:

1. **Test case generation:**
   Requirement mentioned in SRS document should be testable, the conducted tests reveal the error present in the requirement. It is generally believed that if the test is difficult or impossible to design than, this usually means that requirement will be difficult to implement and it should be reconsidered.
2. **Prototyping:**
   In this validation techniques the prototype of the system is presented before the end-user or customer, they experiment with the presented model and check if it meets their need. This type of model is generally used to collect feedback about the requirement of the user.
3. **Requirements Reviews:**
   In this approach, the SRS is carefully reviewed by a group of people including people from both the contractor organisations and the client side, the reviewer systematically analyses the document to check error and ambiguity.
4. **Automated Consistency Analysis:**
   This approach is used for automatic detection of an error, such as nondeterminism, missing cases, a type error, and circular definitions, in requirements specifications.
   First, the requirement is structured in formal notation then CASE tool is used to check in-consistency of the system, The report of all inconsistencies is identified and corrective actions are taken.
5. **Walk-through:**
   A walkthrough does not have a formally defined procedure and does not require a differentiated role assignment.
   - Checking early whether the idea is feasible or not.
   - Obtaining the opinions and suggestion of other people.
   - Checking the approval of others and reaching an agreement.

Requirements analysis process

Requirements Analysis is the process of defining the expectations of the users for an application that is to be built or modified. Requirements analysis involves all the tasks that are conducted to identify the needs of different stakeholders. Therefore requirements analysis means to analyze, document, validate and manage software or system requirements. High-quality requirements are documented, actionable, measurable, testable, traceable, helps to identify business opportunities, and are defined to a facilitate system design.

Requirements analysis process

The requirements analysis process involves the following steps:

Eliciting requirements.The process of gathering requirements by communicating with the customers is known as eliciting requirements.

Analyzing requirements

This step helps to determine the quality of the requirements. It involves identifying whether the requirements are unclear, incomplete, ambiguous, and contradictory. These issues resolved before moving to the next step.

Requirements modeling

In Requirements modeling, the requirements are usually documented in different formats such as use cases, user stories, natural-language documents, or process specification.

Review and retrospective

This step is conducted to reflect on the previous iterations of requirements gathering in a bid to make improvements in the process going forward.

Requirements Analysis Techniques

There are different techniques used for Requirements Analysis. Below is a list of different Requirements Analysis Techniques:

Business process modeling notation (BPMN)

This technique is similar to creating process flowcharts, although BPMN has its own symbols and elements. Business process modeling and notation is used to create graphs for the business process. These graphs simplify understanding the business process. BPMN is widely popular as a process improvement methodology.

UML (Unified Modeling Language)

UML consists of an integrated set of diagrams that are created to specify, visualize, construct and document the artifacts of a software system. UML is a useful technique while creating object-oriented software and working with the software development process. In UML, graphical notations are used to represent the design of a software project. UML also help in validating the architectural design of the software.

Flowchart technique

A flowchart depicts the sequential flow and control logic of a set of activities that are related. Flowcharts are in different formats such as linear, cross-functional, and top-down. The flowchart can represent system interactions, data flows, etc. Flow charts are easy to understand and can be used by both the technical and non-technical team members. Flowchart technique helps in showcasing the critical attributes of a process.

Data flow diagram

This technique is used to visually represent systems and processes that are complex and difficult to describe in text. Data flow diagrams represent the flow of information through a process or a system. It also includes the data inputs and outputs, data stores, and the various subprocess through which the data moves. DFD describes various entities and their relationships with the help of standardized notations and symbols. By visualizing all the elements of the system it is easier to identify any shortcomings. These shortcomings are then eliminated in a bid to create a robust solution.

Role Activity Diagrams (RAD)

Role-activity diagram (RAD) is a role-oriented process model that represents role-activity diagrams. Role activity diagrams are a high-level view that captures the dynamics and role structure of an organization. Roles are used to grouping together activities into units of responsibilities. Activities are the basic parts of a role. An activity may be either carried out in isolation or it may require coordination with other activities within the role.

Gantt Charts

Gantt charts used in project planning as they provide a visual representation of tasks that are scheduled along with the timelines. The Gantt charts help to know what is scheduled to be completed by which date. The start and end dates of all the tasks in the project can be seen in a single view.

IDEF (Integrated Definition for Function Modeling)

Integrated definition for function modeling (IDEFM) technique represents the functions of a process and their relationships to child and parent systems with the help of a box. It provides a blueprint to gain an understanding of an organization’s system.

Gap Analysis

Gap analysis is a technique which helps to analyze the gaps in performance of a software application to determine whether the business requirements are met or not. It also involves the steps that are to be taken to ensure that all the business requirements are met successfully. Gap denotes the difference between the present state and the target state. Gap analysis is also known as need analysis, need assessment or need-gap analysis.

Structured Analysis:

Structured Analysis is a development method that allows the analyst to understand the system and its activities in a logical way.

It is a systematic approach, which uses graphical tools that analyze and refine the objectives of an existing system and develop a new system specification which can be easily understandable by user.

It has following attributes −

- It is graphic which specifies the presentation of application.
- It divides the processes so that it gives a clear picture of system flow.
- It is logical rather than physical i.e., the elements of system do not depend on vendor or hardware.
- It is an approach that works from high-level overviews to lower-level details.

Structured Analysis Tools

During Structured Analysis, various tools and techniques are used for system development. They are −

- Data Flow Diagrams
- Data Dictionary
- Decision Trees
- Decision Tables
- Structured English
- Pseudocode

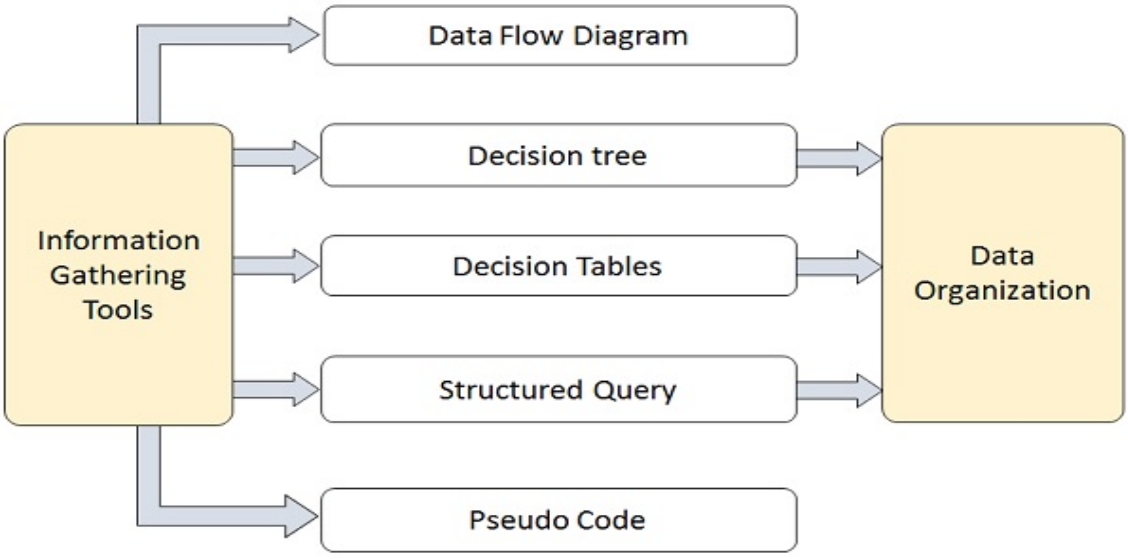


Fig 2.3 Structured Analysis Tools

Data Flow Diagrams (DFD) or Bubble Chart

It is a technique developed by Larry Constantine to express the requirements of system in a graphical form.

- It shows the flow of data between various functions of system and specifies how the current system is implemented.
- It is an initial stage of design phase that functionally divides the requirement specifications down to the lowest level of detail.
- Its graphical nature makes it a good communication tool between user and analyst or analyst and system designer.
- It gives an overview of what data a system processes, what transformations are performed, what data are stored, what results are produced and where they flow.

Basic Elements of DFD

DFD is easy to understand and quite effective when the required design is not clear and the user wants a notational language for communication. However, it requires a large number of iterations for obtaining the most accurate and complete solution.

The following table shows the symbols used in designing a DFD and their significance

Symbol Name	Symbol	Meaning
Square		Source or Destination of Data
Arrow		Data flow
Circle		Process transforming data flow
Open Rectangle		Data Store

Fig 2.4 DFD

Types of DFD

DFDs are of two types: Physical DFD and Logical DFD. The following table lists the points that differentiate a physical DFD from a logical DFD.

Physical DFD	Logical DFD
It is implementation dependent. It shows which functions are performed.	It is implementation independent. It focuses only on the flow of data between processes.
It provides low level details of hardware, software, files, and people.	It explains events of systems and data required by each event.
It depicts how the current system operates and how a system will be implemented.	It shows how business operates; not how the system can be implemented.

Table 2.2 Physical vs Logical DFD

Context Diagram

A context diagram helps in understanding the entire system by one DFD which gives the overview of a system. It starts with mentioning major processes with little details and then goes onto giving more details of the processes with the top-down approach.

The context diagram of mess management is shown below.

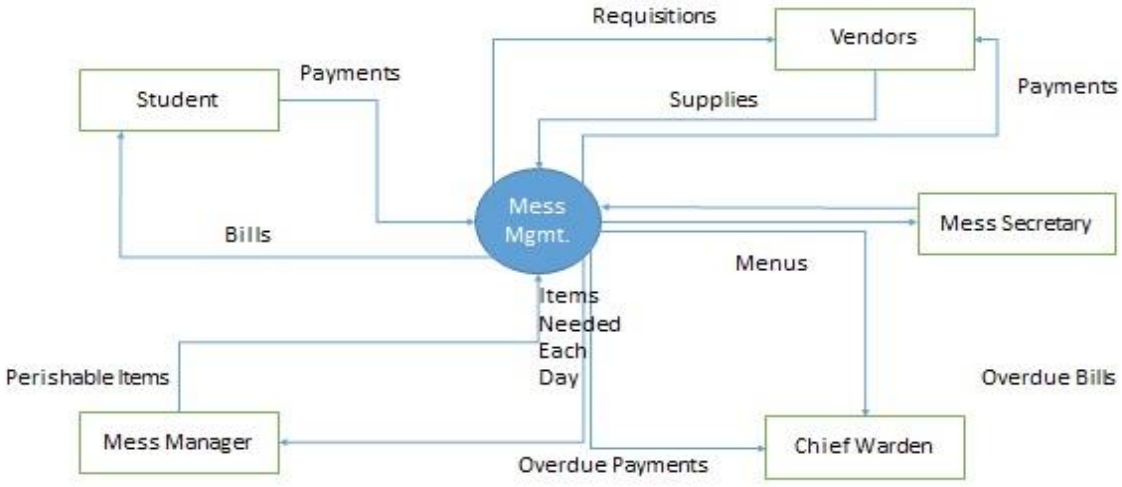


Fig 2.5 Context diagram

Data Dictionary

A data dictionary is a structured repository of data elements in the system. It stores the descriptions of all DFD data elements that is, details and definitions of data flows, data stores, data stored in data stores, and the processes.

A data dictionary improves the communication between the analyst and the user. It plays an important role in building a database. Most DBMSs have a data dictionary as a standard feature. For example, refer the following table –

Sr.No.	Data Name	Description	No. of Characters
1	ISBN	ISBN Number	10
2	TITLE	title	60
3	SUB	Book Subjects	80
4	ANAME	Author Name	15

Table 2.3 Data Dictionary

Decision Trees

Decision trees are a method for defining complex relationships by describing decisions and avoiding the problems in communication. A decision tree is a diagram that shows alternative

actions and conditions within horizontal tree framework. Thus, it depicts which conditions to consider first, second, and so on.

Decision trees depict the relationship of each condition and their permissible actions. A square node indicates an action and a circle indicates a condition. It forces analysts to consider the sequence of decisions and identifies the actual decision that must be made.

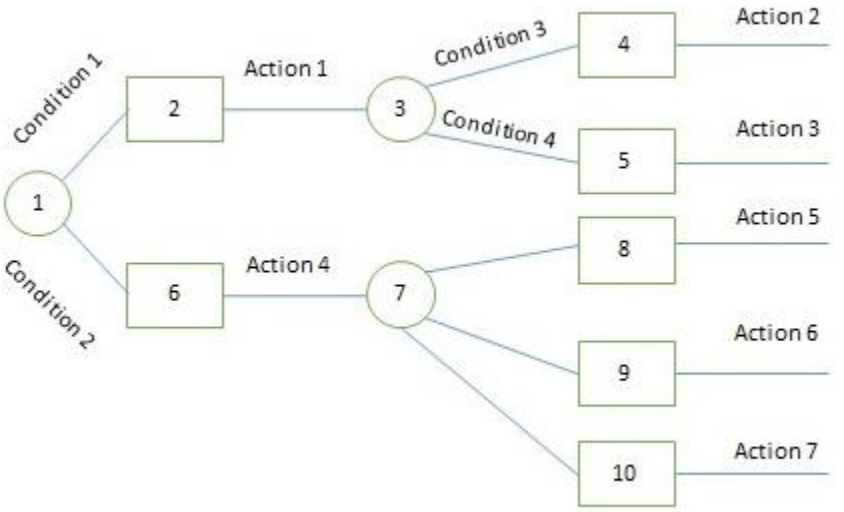


Fig 2.6 Decision Trees

The major limitation of a decision tree is that it lacks information in its format to describe what other combinations of conditions you can take for testing. It is a single representation of the relationships between conditions and actions.

For example, refer the following decision tree:

Decision Tables

Decision tables are a method of describing the complex logical relationship in a precise manner which is easily understandable.

- It is useful in situations where the resulting actions depend on the occurrence of one or several combinations of independent conditions.
- It is a matrix containing row or columns for defining a problem and the actions.

Components of a Decision Table

- **Condition Stub** − It is in the upper left quadrant which lists all the condition to be checked.
- **Action Stub** − It is in the lower left quadrant which outlines all the action to be carried out to meet such condition.
- **Condition Entry** − It is in upper right quadrant which provides answers to questions asked in condition stub quadrant.
- **Action Entry** − It is in lower right quadrant which indicates the appropriate action resulting from the answers to the conditions in the condition entry quadrant.

The entries in decision table are given by Decision Rules which define the relationships between combinations of conditions and courses of action. In rules section,

- Y shows the existence of a condition.
- N represents the condition, which is not satisfied.
- A blank - against action states it is to be ignored.
- X (or a check mark will do) against action states it is to be carried out.

For example, refer the following table –

CONDITIONS	Rule 1	Rule 2	Rule 3	Rule 4
Advance payment made	Y	N	N	N
Purchase amount = Rs 10,000/-	-	Y	Y	N
Regular Customer	-	Y	N	-
ACTIONS				
Give 5% discount	X	X	-	-
Give no discount	-	-	X	X

Table 2.4 Decision Table

Software Prototyping:

Prototype methodology is defined as a Software Development model in which a prototype is built, test, and then reworked when needed until an acceptable prototype is achieved. It also creates a base to produce the final system.

Software prototyping model works best in scenarios where the project's requirement are not known. It is an iterative, trial, and error method which take place between the developer and the client.

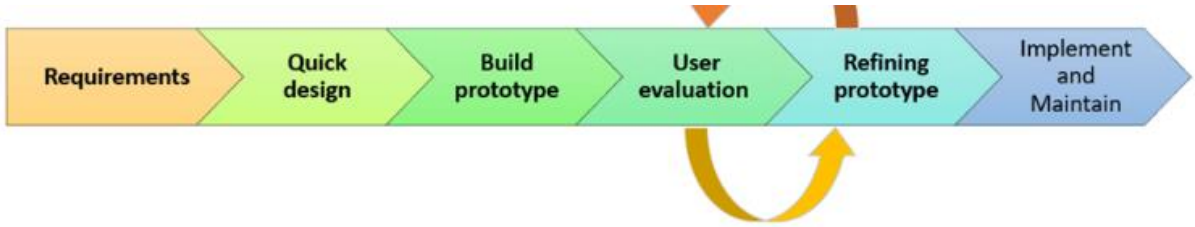


Fig 2.7 Prototype model

Prototyping Model has following six SDLC phases as follow:

Step 1: Requirements gathering and analysis

A prototyping model starts with requirement analysis. In this phase, the requirements of the system are defined in detail. During the process, the users of the system are interviewed to know what is their expectation from the system.

Step 2: Quick design

The second phase is a preliminary design or a quick design. In this stage, a simple design of the system is created. However, it is not a complete design. It gives a brief idea of the system to the user. The quick design helps in developing the prototype.

Step 3: Build a Prototype

In this phase, an actual prototype is designed based on the information gathered from quick design. It is a small working model of the required system.

Step 4: Initial user evaluation

In this stage, the proposed system is presented to the client for an initial evaluation. It helps to find out the strength and weakness of the working model. Comment and suggestion are collected from the customer and provided to the developer.

Step 5: Refining prototype

If the user is not happy with the current prototype, you need to refine the prototype according to the user's feedback and suggestions.

This phase will not over until all the requirements specified by the user are met. Once the user is satisfied with the developed prototype, a final system is developed based on the approved final prototype.

Step 6: Implement Product and Maintain

Once the final system is developed based on the final prototype, it is thoroughly tested and deployed to production. The system undergoes routine maintenance for minimizing downtime and prevent large-scale failures.

Types of Prototyping Models

Four types of Prototyping models are:

1. Rapid Throwaway prototypes
2. Evolutionary prototype
3. Incremental prototype
4. Extreme prototype

Rapid Throwaway Prototype

Rapid throwaway is based on the preliminary requirement. It is quickly developed to show how the requirement will look visually. The customer's feedback helps drives changes to the requirement, and the prototype is again created until the requirement is baselined.

In this method, a developed prototype will be discarded and will not be a part of the ultimately accepted prototype. This technique is useful for exploring ideas and getting instant feedback for customer requirements.

Evolutionary Prototyping

Here, the prototype developed is incrementally refined based on customer's feedback until it is finally accepted. It helps you to save time as well as effort. That's because developing a prototype from scratch for every interaction of the process can sometimes be very frustrating.

This model is helpful for a project which uses a new technology that is not well understood. It is also used for a complex project where every functionality must be checked once. It is helpful when the requirement is not stable or not understood clearly at the initial stage.

Incremental Prototyping

In incremental Prototyping, the final product is decimated into different small prototypes and developed individually. Eventually, the different prototypes are merged into a single product. This method is helpful to reduce the feedback time between the user and the application development team.

Extreme Prototyping:

Extreme prototyping method is mostly used for web development. It is consists of three sequential phases.

1. Basic prototype with all the existing page is present in the HTML format.
2. You can simulate data process using a prototype services layer.
3. The services are implemented and integrated into the final prototype.

Data modeling:

Data modeling (data modelling) is the process of creating a data model for the data to be stored in a Database. This data model is a conceptual representation of Data objects, the associations between different data objects and the rules. Data modeling helps in the visual representation of data and enforces business rules, regulatory compliances, and government policies on the data. Data Models ensure consistency in naming conventions, default values, semantics, security while ensuring quality of the data.

Data model emphasizes on what data is needed and how it should be organized instead of what operations need to be performed on the data. Data Model is like architect's building plan which helps to build a conceptual model and set the relationship between data items.

The two types of **Data Models techniques** are

- Entity Relationship (E-R) Model
- UML (Unified Modelling Language)

Goal of Data Model:

The primary goal of using data model are:

- Ensures that all data objects required by the database are accurately represented. Omission of data will lead to creation of faulty reports and produce incorrect results.
- A data model helps design the database at the conceptual, physical and logical levels.

- Data Model structure helps to define the relational tables, primary and foreign keys and stored procedures.
- It provides a clear picture of the base data and can be used by database developers to create a physical database.
- It is also helpful to identify missing and redundant data.
- Though the initial creation of data model is labor and time consuming, in the long run, it makes your IT infrastructure upgrade and maintenance cheaper and faster.

Types of Data Models

There are mainly three different types of data models:

1. **Conceptual:** This Data Model defines **WHAT** the system contains. This model is typically created by Business stakeholders and Data Architects. The purpose is to organize, scope and define business concepts and rules.
2. **Logical:** Defines **HOW** the system should be implemented regardless of the DBMS. This model is typically created by Data Architects and Business Analysts. The purpose is to developed technical map of rules and data structures.
3. **Physical**: This Data Model describes **HOW** the system will be implemented using a specific DBMS system. This model is typically created by DBA and developers. The purpose is actual implementation of the database.

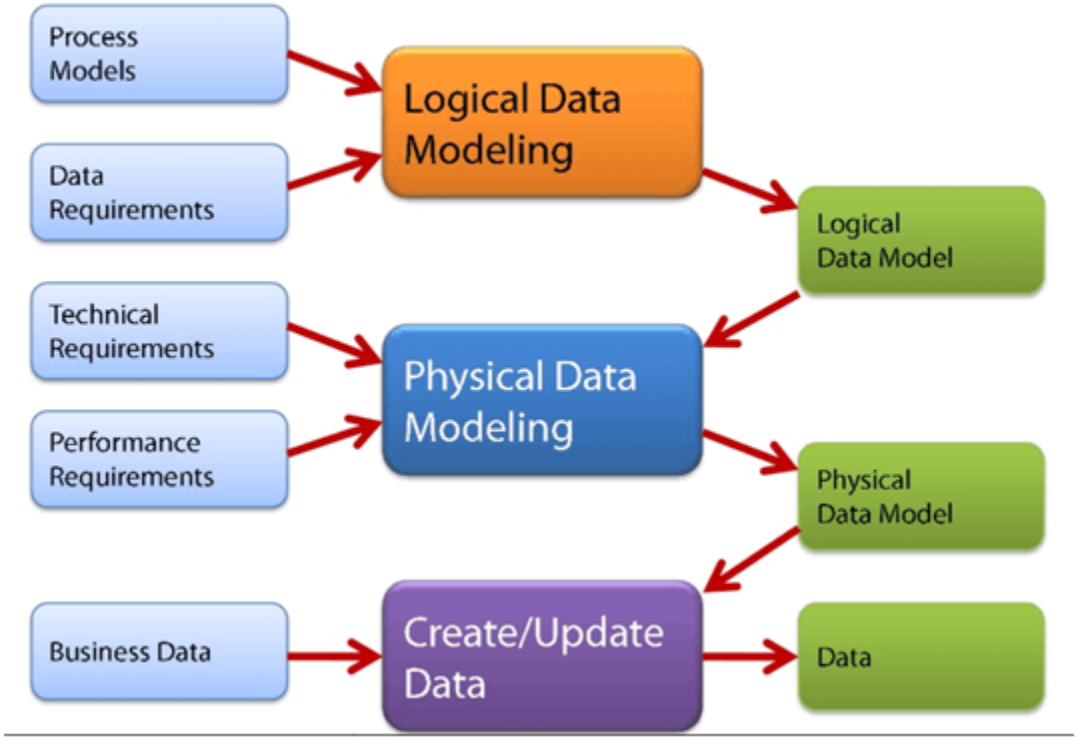


Fig 2.7 Data models

Conceptual Model

The main aim of this model is to establish the entities, their attributes, and their relationships. In this Data modeling level, there is hardly any detail available of the actual Database structure.

The 3 basic tenants of Data Model are

Entity: A real-world thing

Attribute: Characteristics or properties of an entity

Relationship: Dependency or association between two entities

For example:

- Customer and Product are two entities. Customer number and name are attributes of the Customer entity
- Product name and price are attributes of product entity
- Sale is the relationship between the customer and product

Fig 2.8 Conceptual Model

Logical Data Model

Logical data models add further information to the conceptual model elements. It defines the structure of the data elements and set the relationships between them.

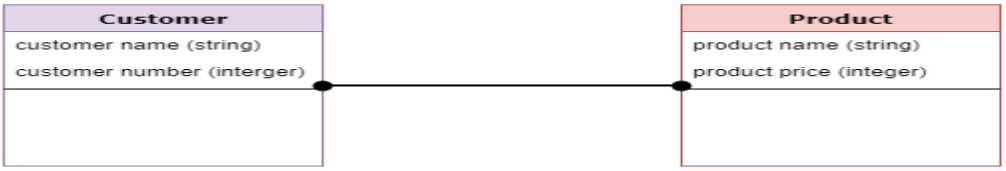


Fig 2.9 Logical Data Model

Physical Data Model

A Physical Data Model describes the database specific implementation of the data model. It offers an abstraction of the database and helps generate schema. This is because of the richness of meta-data offered by a Physical Data Model.

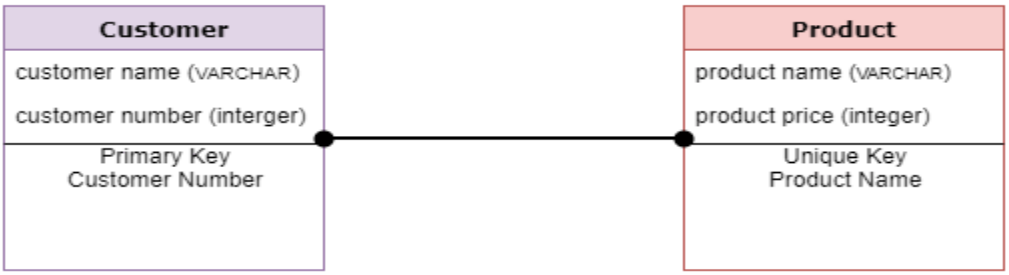


Fig 2.10 Physical Data Model

Behavioural Model:

All behavioural models really do is describe the control structure of a system. This can be things like:

- Sequence of operations
- Object states
- and Object interactions

Furthermore, this modelling layer can also be called Dynamic Modelling. The activity of creating a behavioural model is commonly known as behaviouralmodelling. As well as this, a system should also only have one behavioural model – much like functional modelling.

Representation:

we represent them with dynamic diagrams. For example:

- (Design) Sequence Diagrams
- Communication Diagrams or collaboration diagram
- State Diagrams or state machine diagram or state chart

For consistency use communication diagram and state diagram

If we have both a sequence diagram AND and communication diagram, then together these are known as interaction diagrams, this is because they both represent how objects interact with one another using messages.

Description:

A behavioural model describes when the system is changing.

The key feature (subject) of a behavioural model is – Objects.

Data Dictionaries

A data dictionary is a file or a set of files that includes a database's metadata. The data dictionary hold records about other objects in the database, such as data ownership, data relationships to other objects, and other data. The data dictionary is an essential component of any relational database. Ironically, because of its importance, it is invisible to most database users. Typically, only database administrators interact with the data dictionary.

The data dictionary, in general, includes information about the following:

- Name of the data item
- Aliases
- Description/purpose
- Related data items
- Range of values
- Data structure definition/Forms

The **name** of the data item is self-explanatory.

Aliases include other names by which this data item is called DEO for Data Entry Operator and DR for Deputy Registrar.

Description/purpose is a textual description of what the data item is used for or why it exists.

Related data items capture relationships between data items e.g., total_marks must always equal to internal_marks plus external_marks.

Range of values records all possible values, e.g. total marks must be positive and between 0 to 100.

Data structure Forms: Data flows capture the name of processes that generate or receive the data items. If the data item is primitive, then data structure form captures the physical structures of the data item. If the data is itself a data aggregate, then data structure form capture the composition of the data items in terms of other data items.

The mathematical operators used within the data dictionary are defined in the table:

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – III – Software Engineering – SCS1305

III.DESIGN PROCESS AND CONCEPTS

Design process - Modular design - Design heuristic - Design model and document - Architectural design - Software architecture - Data design - Architecture data - Transform and transaction mapping - User interface design - User interface design principles.

Design Process

Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language.

The software design phase is the first step in SDLC (Software Design Life Cycle), which moves the concentration from the problem domain to the solution domain. In software design, we consider the system to be a set of components or modules with clearly defined behaviors & boundaries.

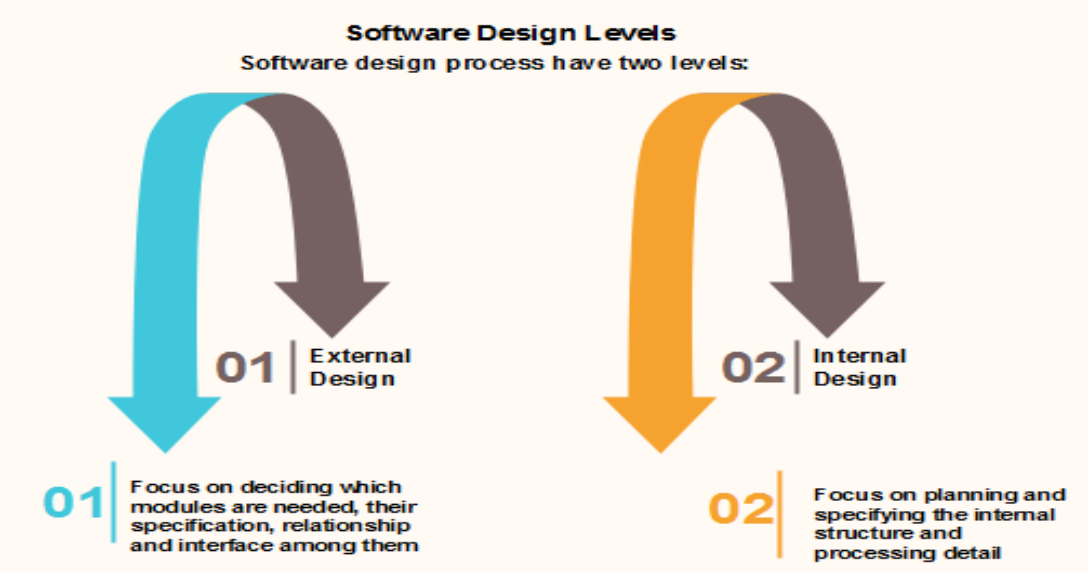


Fig-3.1

Objectives of Software Design

Following are the purposes of Software design:

Fig 3.2

- Correctness:Software design should be correct as per requirement.
- Completeness:The design should have all components like data structures, modules, and external interfaces, etc.
- Efficiency:Resources should be used efficiently by the program.
- Flexibility:Able to modify on changing needs.
- Consistency:There should not be any inconsistency in the design.
- Maintainability: The design should be so simple so that it can be easily maintainable by other designers

Software Design Principles

Software design principles are concerned with providing means to handle the complexity of the design process effectively. Effectively managing the complexity will not only reduce the effort needed for design but can also reduce the scope of introducing errors during design.

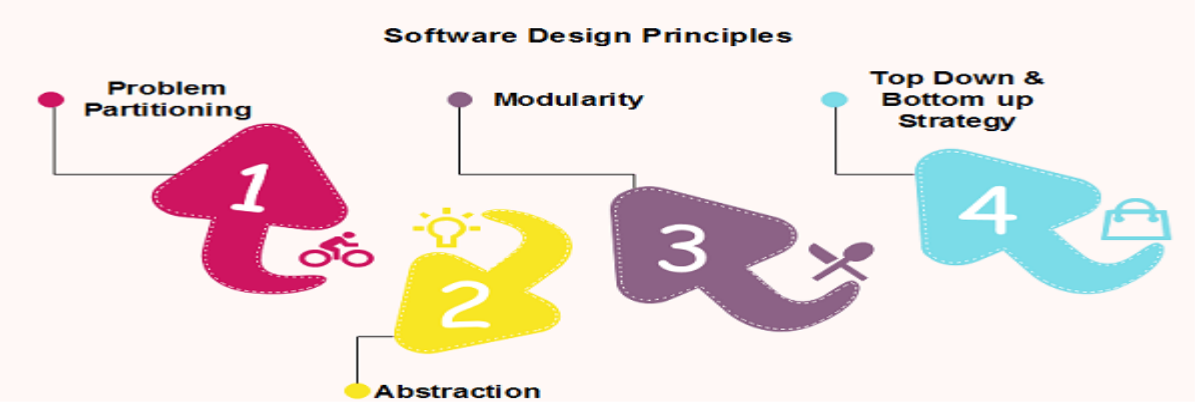


Fig 3.3

Problem Partitioning

For small problem, we can handle the entire problem at once but for the significant problem, divide the problems and conquer the problem it means to divide the problem into smaller pieces so that each piece can be captured separately.

For software design, the goal is to divide the problem into manageable pieces.

Benefits of Problem Partitioning

1. Software is easy to understand
2. Software becomes simple
3. Software is easy to test
4. Software is easy to modify
5. Software is easy to maintain
6. Software is easy to expand

These pieces cannot be entirely independent of each other as they together form the system. They have to cooperate and communicate to solve the problem. This communication adds complexity.

Abstraction

An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation. Abstraction can be used for existing element as well as the component being designed.

Here, there are two common abstraction mechanisms

1. Functional Abstraction
2. Data Abstraction

Functional Abstraction

i. A module is specified by the method it performs.
ii. The details of the algorithm to accomplish the functions are not visible to the user of the function.

Functional abstraction forms the basis for **Function oriented design approaches**.

Data Abstraction

Details of the data elements are not visible to the users of data. Data Abstraction forms the basis for **Object Oriented design approaches**.

Modular Design:

Modularity

Modularity specifies to the division of software into separate modules which are differently named and addressed and are integrated later on in to obtain the completely functional software. It is the only property that allows a program to be intellectually manageable. Single large programs are difficult to understand and read due to a large number of reference variables, control paths, global variables, etc.

The desirable properties of a modular system are:

- Each module is a well-defined system that can be used with other applications.
- Each module has single specified objectives.
- Modules can be separately compiled and saved in the library.
- Modules should be easier to use than to build.
- Modules are simpler from outside than inside.

Modularity

Modularity specifies to the division of software into separate modules which are differently named and addressed and are integrated later on in to obtain the completely functional software. It is the only property that allows a program to be intellectually manageable. Single large programs are difficult to understand and read due to a large number of reference variables, control paths, global variables, etc.

The desirable properties of a modular system are:

- Each module is a well-defined system that can be used with other applications.
- Each module has single specified objectives.
- Modules can be separately compiled and saved in the library.
- Modules should be easier to use than to build.
- Modules are simpler from outside than inside.

Advantages and Disadvantages of Modularity

Advantages of Modularity

There are several advantages of Modularity

- It allows large programs to be written by several or different people
- It encourages the creation of commonly used routines to be placed in the library and used by other programs.
- It simplifies the overlay procedure of loading a large program into main storage.
- It provides more checkpoints to measure progress.
- It provides a framework for complete testing, more accessible to test
- It produced the well designed and more readable program.

Disadvantages of Modularity

There are several disadvantages of Modularity

- Execution time maybe, but not certainly, longer
- Storage size perhaps, but is not certainly, increased
- Compilation and loading time may be longer
- Inter-module communication problems may be increased
- More linkage required, run-time may be longer, more source lines must be written, and more documentation has to be done

Modular Design

Modular design reduces the design complexity and results in easier and faster implementation by allowing parallel development of various parts of a system. We discuss a different section of modular design in detail in this section:

1. Functional Independence: Functional independence is achieved by developing functions that perform only one kind of task and do not excessively interact with other

modules. Independence is important because it makes implementation more accessible and faster. The independent modules are easier to maintain, test, and reduce error propagation and can be reused in other programs as well. Thus, functional independence is a good design feature which ensures software quality.

It is measured using two criteria:

- **Cohesion:** It measures the relative function strength of a module.
- **Coupling:** It measures the relative interdependence among modules.

2. Information hiding: The fundamental of Information hiding suggests that modules can be characterized by the design decisions that protect from the others, i.e., In other words, modules should be specified that data include within a module is inaccessible to other modules that do not need for such information.

The use of information hiding as design criteria for modular system provides the most significant benefits when modifications are required during testing's and later during software maintenance. This is because as most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to different locations within the software.

Strategy of Design

A good system design strategy is to organize the program modules in such a method that are easy to develop and latter too, change. Structured design methods help developers to deal with the size and complexity of programs. Analysts generate instructions for the developers about how code should be composed and how pieces of code should fit together to form a program.

To design a system, there are two possible approaches:

1. Top-down Approach
2. Bottom-up Approach

1. **Top-down Approach:** This approach starts with the identification of the main components and then decomposing them into their more detailed sub-components.

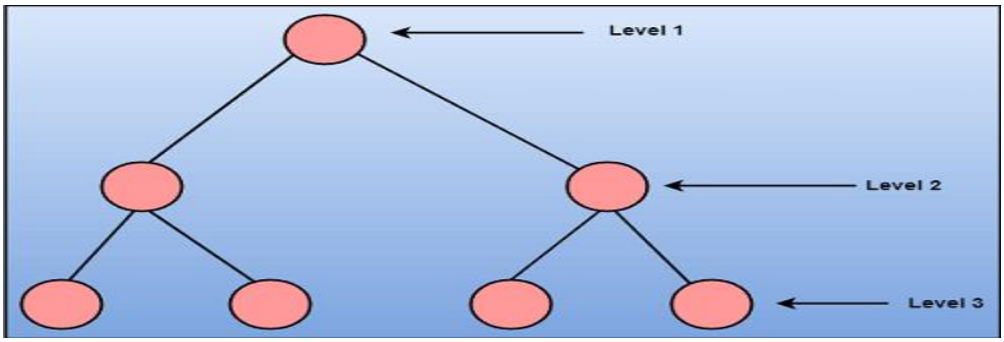


Fig 3.4

2. **Bottom-up Approach:** A bottom-up approach begins with the lower details and moves towards up the hierarchy, as shown in fig. This approach is suitable in case of an existing system.

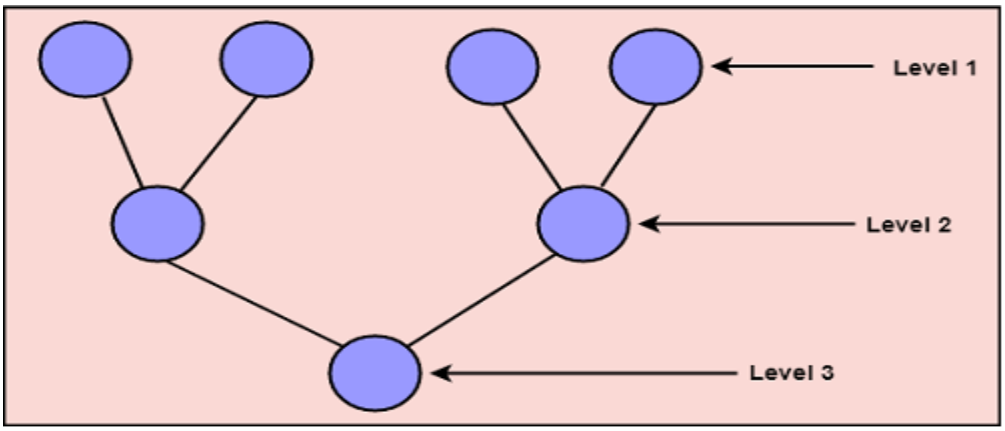


Fig 3.5

Coupling and Cohesion

Module Coupling

In software engineering, the coupling is the degree of interdependence between software modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. **Uncoupled modules** have no interdependence at all within them.

The various types of coupling techniques are shown in fig

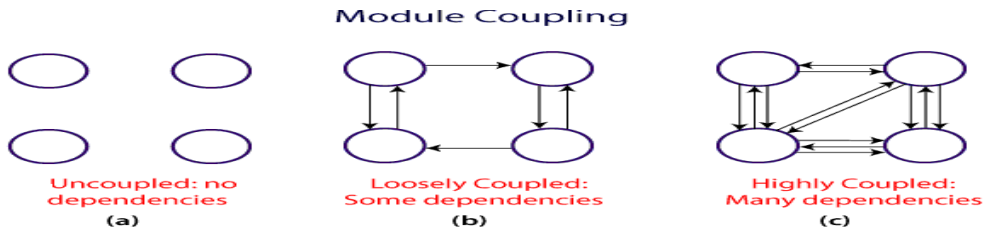


Fig 3.6

A good design is the one that has low coupling. Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors.

Types of Module Coupling

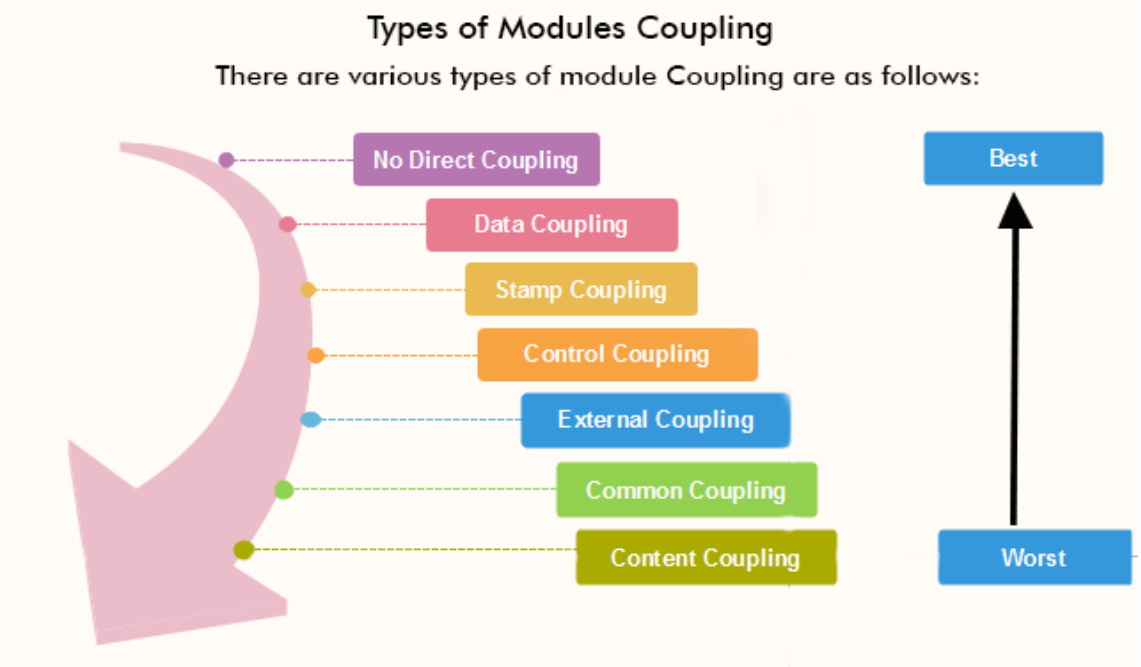
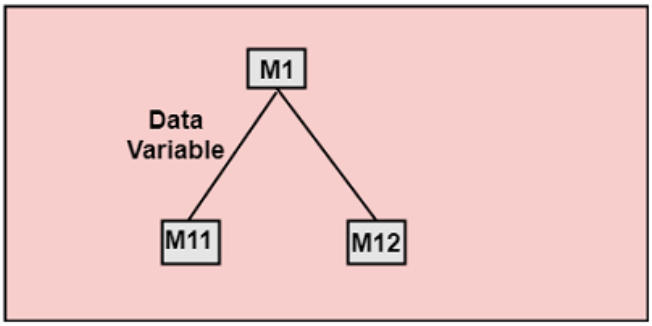


Fig 3.7

1. No Direct Coupling: There is no direct coupling between M1 and M2.

In this case, modules are subordinates to different modules. Therefore, no direct coupling.

2. Data Coupling: When data of one module is passed to another module, this is called data coupling.

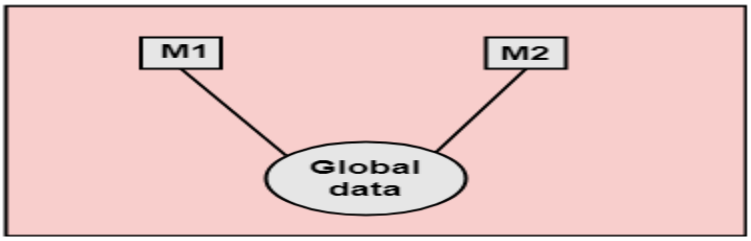


3. Stamp Coupling: Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.

4. Control Coupling: Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

5. External Coupling: External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

6. Common Coupling: Two modules are common coupled if they share information through some global data items.



7. **Content Coupling:** Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

Module Cohesion

In computer programming, cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.

Cohesion is an **ordinal** type of measurement and is generally described as "high cohesion" or "low cohesion."

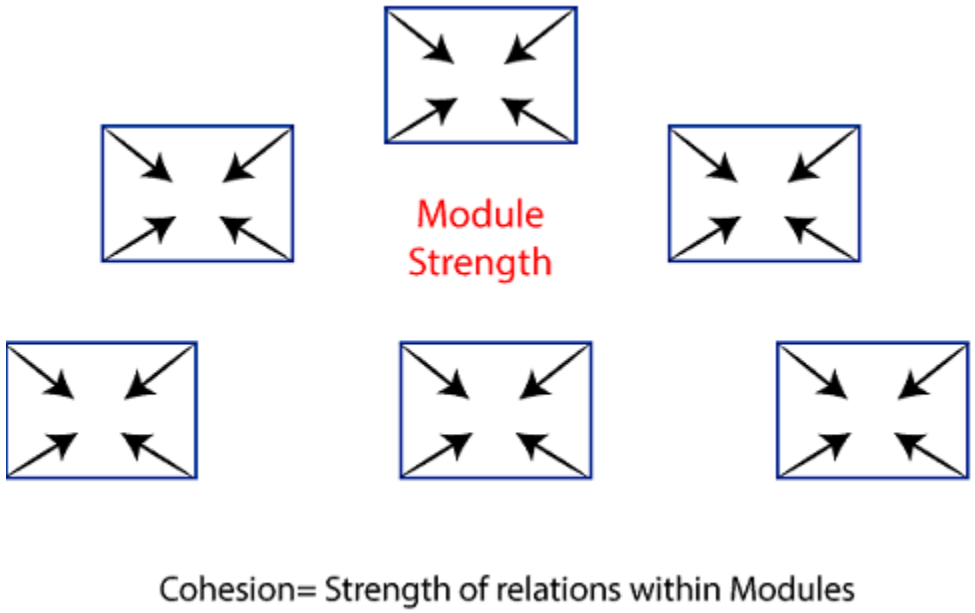


Fig 3.8

Types of Modules Cohesion

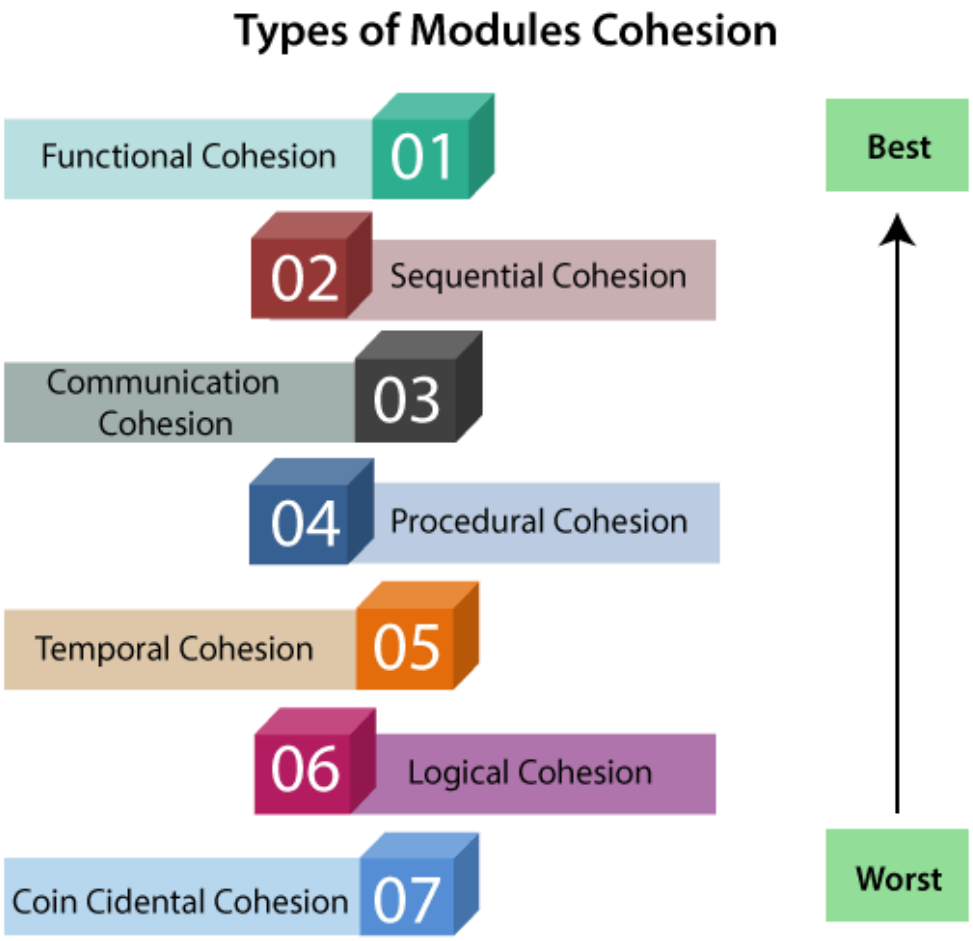


Fig 3.9

1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.
2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.
3. **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.
4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.

5. **Temporal Cohesion:** When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.
6. **Logical Cohesion:** A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.
7. **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all

Coupling vs Cohesion

Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding.
Coupling shows the relationships between modules.	Cohesion shows the relationship within the module.
Coupling shows the relative **independence** between the modules.	Cohesion shows the module's relative **functional** strength.
While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

Table 3.1

Function Oriented Design

Function Oriented design is a method to software design where the model is decomposed into a set of interacting units or modules where each unit or module has a clearly defined function. Thus, the system is designed from a functional viewpoint.

Design Notations

Design Notations are primarily meant to be used during the process of design and are used to represent design or design decisions. For a function-oriented design, the design can be represented graphically or mathematically by the following:

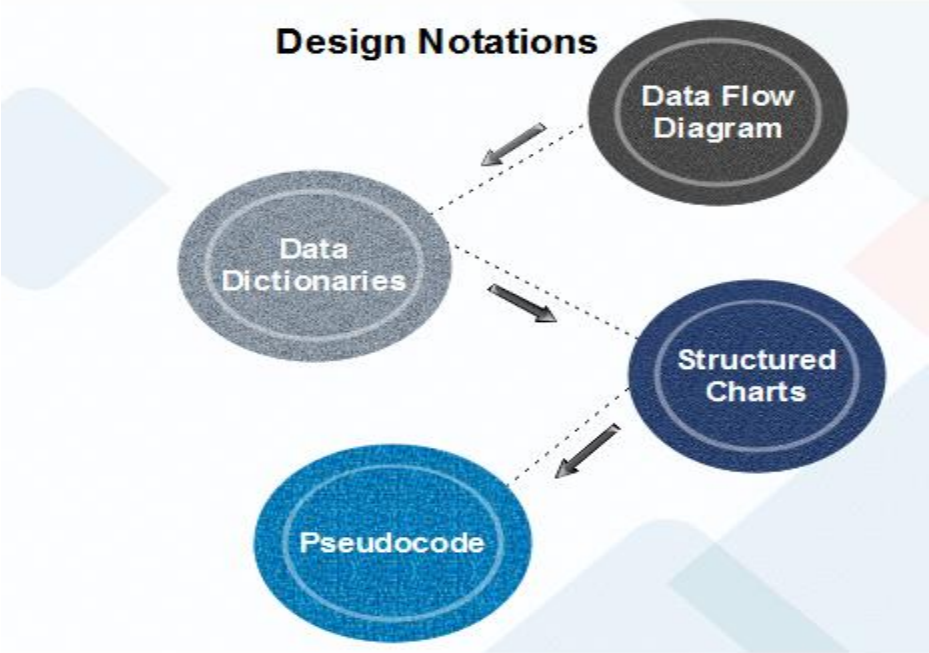


Fig 3.10 Design Notation

Data Flow Diagram

Data-flow design is concerned with designing a series of functional transformations that convert system inputs into the required outputs. The design is described as data-flow diagrams. These diagrams show how data flows through a system and how the output is derived from the input through a series of functional transformations.

Data-flow diagrams are a useful and intuitive way of describing a system. They are generally understandable without specialized training, notably if control information is excluded. They show end-to-end processing. That is the flow of processing from when data enters the system to where it leaves the system can be traced.

Data-flow design is an integral part of several design methods, and most CASE tools support data-flow diagram creation. Different ways may use different icons to represent data-flow diagram entities, but their meanings are similar.

The notation which is used is based on the following symbols:

Symbol	Name	Meaning
	Rounded Rectangle	It represents functions which transforms input to output. The transformation name indicates its function.
	Rectangle	It represents data stores. Again, they should give a descriptive name.
	Circle	It represents user interactions with the system that provides input or receives output.
	Arrows	It shows the direction of data flow. Their name describes the data flowing along the path.
"and" and "or"	Keywords	The keywords "and" and "or". These have their usual meanings in boolean expressions. They are used to link data flows when more than one data flow may be input or output from a transformation.

Table 3.2

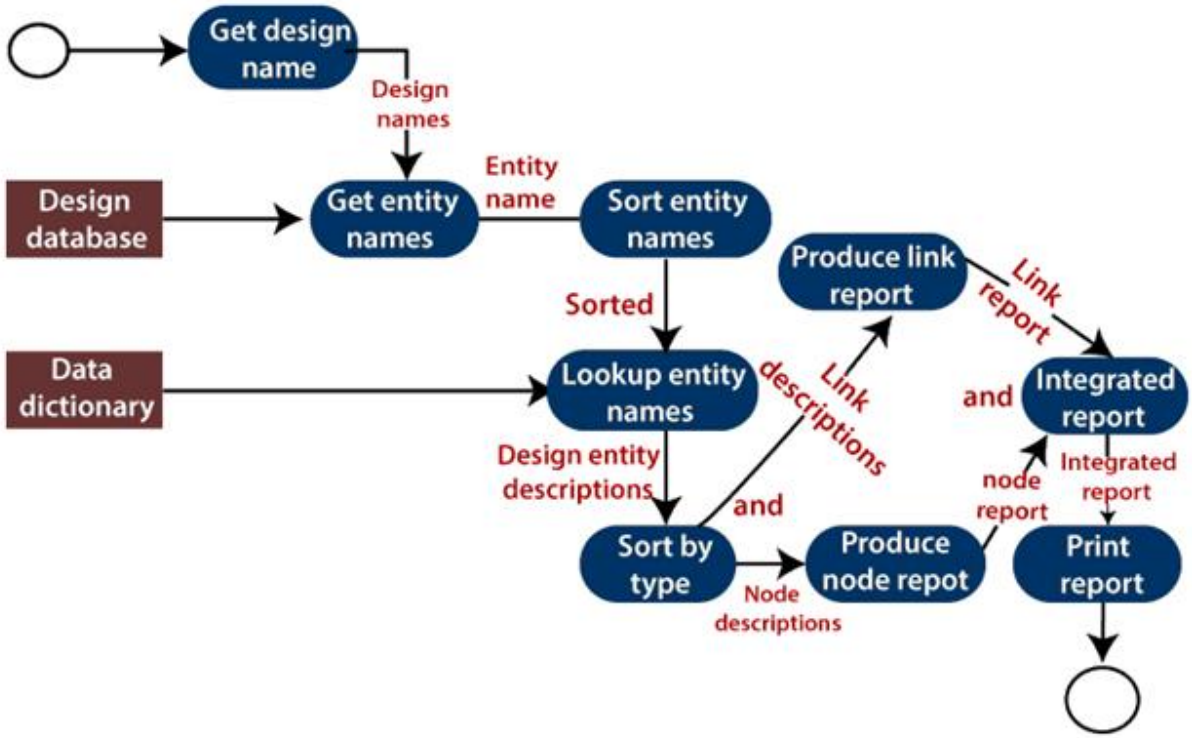


Fig 3.11- DFD Report generator

The report generator produces a report which describes all of the named entities in a data-flow diagram. The user inputs the name of the design represented by the diagram. The report generator then finds all the names used in the data-flow diagram. It looks up a data dictionary and retrieves information about each name. This is then collated into a report which is output by the system.

Object-Oriented Design

In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities). The state is distributed among the objects, and each object handles its state data. For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data. The tasks defined for one purpose cannot refer or change data of other objects. Objects have their internal data which represent their state. Similar objects create a class. In other words, each object is a member of some class. Classes may inherit features from the superclass.

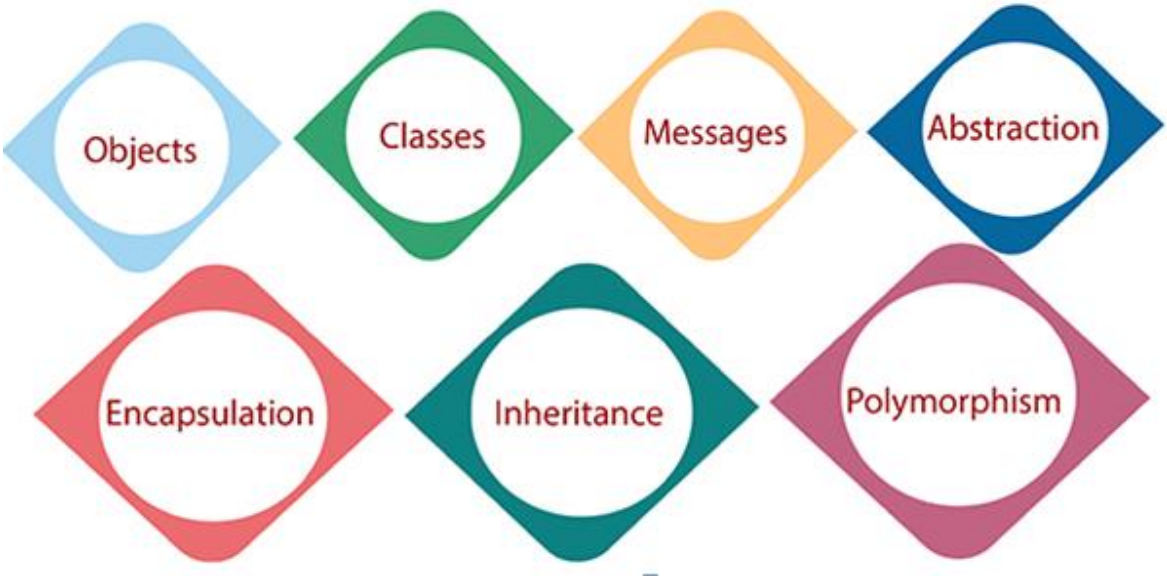


Fig 3.12-OOAD Concepts

1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.
2. **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.

3. **Messages:** Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.
4. **Abstraction** In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.
5. **Encapsulation:** Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.
6. **Inheritance:** OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses.Thisproperty of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.
7. **Polymorphism:** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types. Depending upon how the service is invoked, the respective portion of the code gets executed.

Design Heuristic

Heuristic evaluations are one of the most informal methods of usability inspection in the field of human-computer interaction. There are many sets of usability design heuristics; they are not mutually exclusive and cover many of the same aspects of user interface design.

Match between system and the real world:

The system should speak the user's language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

User control and freedom:

Users often choose system functions by mistake and will need a clearly marked "emergency exit" to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

Consistency and standards:

Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

Error prevention:
Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

Recognition rather than recall:
Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate.

Flexibility and efficiency of use:

Accelerators unseen by the novice user—may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

Aesthetic and minimalist design:
Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

Flexibility and efficiency of use:
Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

Help and documentation:
Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be toolarge.

Design Model

The design concepts provide the software designer with a foundation from which more sophisticated methods can be applied. A set of fundamental design concepts has evolved. They are:

Abstraction - Abstraction is the process or result of generalization by reducing theinformationcontent of a concept or an observable phenomenon, typically in order to retain only information which is relevant for a particular purpose.

1. **Refinement** - It is the process of elaboration. A hierarchy is developed by decomposing a macroscopic statement of function in a step-wise fashion until programming language statements are reached. In each step, one or several instructions of a given program are decomposed into more detailed instructions. Abstraction and Refinement are complementaryconcepts.
2. **Modularity** - Software architecture is divided into components calledmodules.
3. **Software Architecture** - It refers to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. A good software architecture will yield a good return on investment with respect to the desired outcome of the project, e.g. in terms of performance, quality, schedule andcost.
4. **Control Hierarchy** - A program structure that represents the organization of a program component and implies a hierarchy ofcontrol.
5. **Structural Partitioning** - The program structure can be divided both horizontally and vertically. Horizontal partitions define separate branches of modular hierarchy for each major program function. Vertical partitioning suggests that control and work should be distributed top down in the programstructure.
6. **Data Structure** - It is a representation of the logical relationship among individual elements ofdata.
7. **Software Procedure** - It focuses on the processing of each modulesindividually
8. **Information Hiding** - Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information

Software Design Documentation

It acts as ‘a description of software created to facilitate analysis, planning, implementation, and decision-making. This design description is -used as a medium for communicating software design information and can be considered as a blueprint or model of the system.

While developing SDD, the design should be described up to the refinement level that is sufficient for explaining every task including inter-task communications, data structures, and databases. No refinement of any task should be left to be made during the coding phase.

The information that the software design document should describe depends on various factors including the type of software being developed and the approach used in its development. A number of standards have been suggested to develop a software design document. However, the most widely used standard is by IEEE, which acts as a general framework. This general framework can be customized and adapted to meet the needs of a particular organization. This template consists of several sections, which are listed below.

1.Scope:

Identifies the release or version of the system being designed. The system is divided into modules; the relationship between them and functionalities will be defined. Every iteration of the SDD document describes and identifies the software modules to be added or changed in a release.

2.References:

Lists references (both hardware and software documents and manuals) used in the creation of the SDD that may be of use to the designer, programmer, user, or management personnel. This document is also considered useful for the readers of the document. In this section, any references made to the other documents including references to related project documents, especially the SRS are also listed. The existing software documentation (if any) is also listed.

3.Definition:

Provides a glossary of technical terms used in the document along with their definitions.

4.Purpose:

States the purpose of this document and its intended audience. This is meant primarily for individuals who will be implementing the system.

5.Design description information content: Consists of the following subsections.

i.Introduction: Since SDD represents the software design that is to be implemented, it should describe the design entities into which the system has been partitioned along with their significant properties and relationships.

ii.Design entity: It is a software design component that is different from other design entities in terms of structure and function. The objective of creating design entities is to partition the system into a set of components that can be implemented and modified independently. Note that each design entity is assigned with a unique name and serves a specific purpose and function but all possess some common characteristics.

iii.Design entity attributes: They are properties of the design entity and provides some factual information regarding the entity. Every attribute has an attached description, which includes references and design considerations. The attributes and their associated information are listed in Table.

Attributes	Description
Identification	Identifies name of the entity. All the entities have a unique name.
Type	Describes the kind of entity. This specifies the nature of the entity.

Purpose	Specifies why the entity exists.
Function	Specifies what the entity does.
Subordinates	Identifies sub-ordinate entity of an entity.
Dependencies	Describes relationships that exist between one entity and other entities.
Interface	Describes how entities interact among themselves.
Resources	Describes elements used by the entity that are external to the design.
Processing	Specifies rules used to achieve the specified functions.
Data	Identifies data elements that form part of the internal entity.

6.**Design description organization:**

7.Design views: They describe the software design in a comprehensive manner so that the process of information access and integration is simplified. The design of software can be viewed in multiple ways and each design view describes a distinct aspect of the system. Table lists various design views and their attributes.

Table Design Views and their Description

Design View	Description	Attribute
Decomposition description	Partitions the system into design entities.	Identification, type, purpose, function, and subordinate
Dependency description	Describes relationships between entities.	Identification, type, purpose, dependencies, and resources
Interface description	Consists of list that is required by the stakeholders (designers, developers, and testers) in	Identification, function and interfaces

	order to design entities.	
Detail description	Describes internal details of the design entity.	Identification, processing, and data

Architectural Design:

Dynamic model: Specifies the behavioral aspect of the software architecture and indicates how the structure or system configuration changes as the function changes due to change in the external environmentNeed of Architecture:

The architecture is not the operational software. Rather, it is a representation that enables a

software engineer to:

(1) analyze the effectiveness of the design in meeting its stated requirements,

(2) consider architectural alternatives at a stage when making design changes is still relatively

easy, and

(3) reduce the risks associated with the construction of the software.

Importance :

Software architecture representations enable communications among stakeholders

Architecture highlights early design decisions that will have a profound impact on the

1. ultimate success of the system as an operational entity
2. • The architecture constitutes an intellectually graspable model of how the system is
3. structured and how its components work together
4. Refining architecture into components Instantiations of the system is called as architecture design
5. Architectural Design Representation
6. Architectural design can be represented using the following models.
7. structural model: Illustrates architecture as an ordered collection of program components
8. **Process model:** Focuses on the design of the business or technical process, which must be implemented in the system
9. **Functional model:** Represents the functional hierarchy of a system
10. **Framework model:** Attempts to identify repeatable architectural design patterns encountered in similar types of application. This leads to an increase in the level of abstraction.

Architectural Styles

Architectural styles define a group of interlinked systems that share structural and semantic properties. In short, the objective of using architectural styles is to establish a structure for all the components present in a system. If an existing architecture is to be re-engineered, then imposition of an architectural style results in fundamental changes in the structure of the system. This change also includes re-assignment of the functionality performed by the components.

By applying certain constraints on the design space, we can make different style-specific analysis from an architectural style. In addition, if conventional structures are used for an architectural style, the other stakeholders can easily understand the organization of the system.

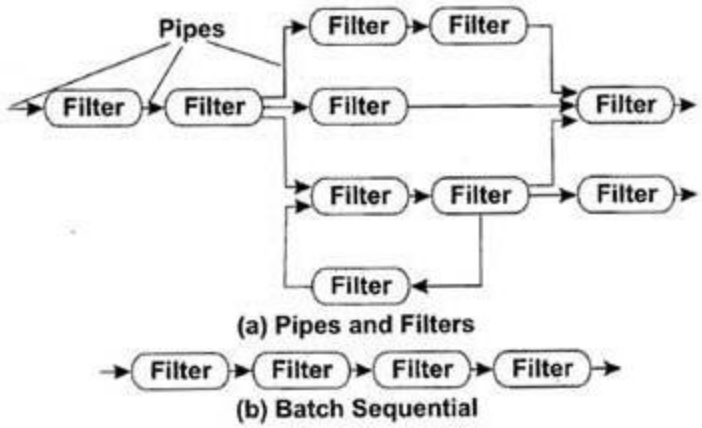
A computer-based system (software is part of this system) exhibits one of the many available architectural styles. Every architectural style describes a system category that includes the following.

1. Computational components such as clients, server, filter, and database to execute the desired system function
2. A set of *connectors* such as procedure call, events broadcast, database protocols, and pipes to provide communication among the computational components
3. Constraints to define integration of components to form a system
4. A *semantic* model, which enable the software designer to identify the characteristics of the system as a whole by studying the characteristics of its components.

Some of the commonly used architectural styles are data-flow architecture, object oriented architecture, layered system architecture, data-centered architecture, and call and return architecture. Note that the use of an appropriate architectural style promotes design reuse, leads to code reuse, and supports interoperability.

Data-flow Architecture

Data-flow architecture is mainly used in the systems that accept some inputs and transform it into the desired outputs by applying a series of transformations. Each component, known as **filter,** transforms the data and sends this transformed data to other filters for further processing using the connector, known as **pipe**. Each filter works as an independent entity, that is, it is not concerned with the filter which is producing or consuming the data. A pipe is a unidirectional channel which transports the data received on one end to the other end. It does not change the data in anyway; it merely supplies the data to the filter on the receiver end.



Most of the times, the data-flow architecture degenerates a batch sequential system. In this system, a batch of data is accepted as input and then a series of sequential filters are applied to transform this data. One common example of this architecture is UNIX shell programs. In these programs, UNIX processes act as filters and the file system through which UNIX processes interact, act as pipes. Other well-known examples of this architecture are compilers, signal processing systems, parallel programming, functional programming, and distributed systems. Some advantages associated with the data-flow architecture are listed below.

- It supports reusability.
- It is maintainable and modifiable.
- It supports concurrent execution.
- Some disadvantages associated with the data-flow architecture are listed below.
- It often degenerates to batch sequential system.
- It does not provide enough support for applications requires user interaction.
- It is difficult to synchronize two different but related streams.

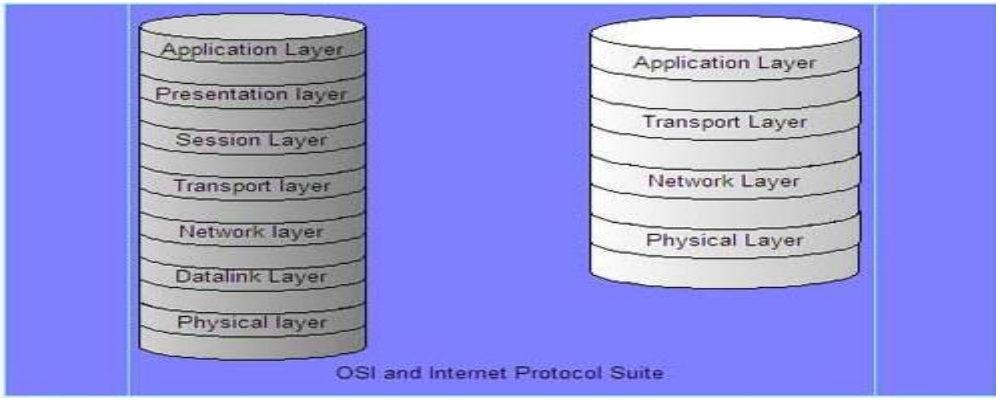
Object-oriented Architecture

In object-oriented architectural style, components of a system encapsulate data and operations, which are applied to manipulate the data. In this style, components are represented as *objects* and they interact with each other through methods (connectors). This architectural style has two important characteristics, which are listed below.

- Objects maintain the integrity of the system.
- An object is not aware of the representation of other objects.
- Some of the advantages associated with the object-oriented architecture are listed below.
- It allows designers to decompose a problem into a collection of independent objects.
- The implementation detail of objects is hidden from each other and hence, they can be changed without affecting other objects.

Layered Architecture

In layered architecture, several layers (components) are defined with each layer performing a well-defined set of operations. These layers are arranged in a hierarchical manner, each one built upon the one below it. Each layer provides a set of services to the layer above it and acts as a client to the layer below it. The interaction between layers is provided through protocols (connectors) that define a set of rules to be followed during interaction. One common example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organization for Standardization) communication system.



Data-centered Architecture

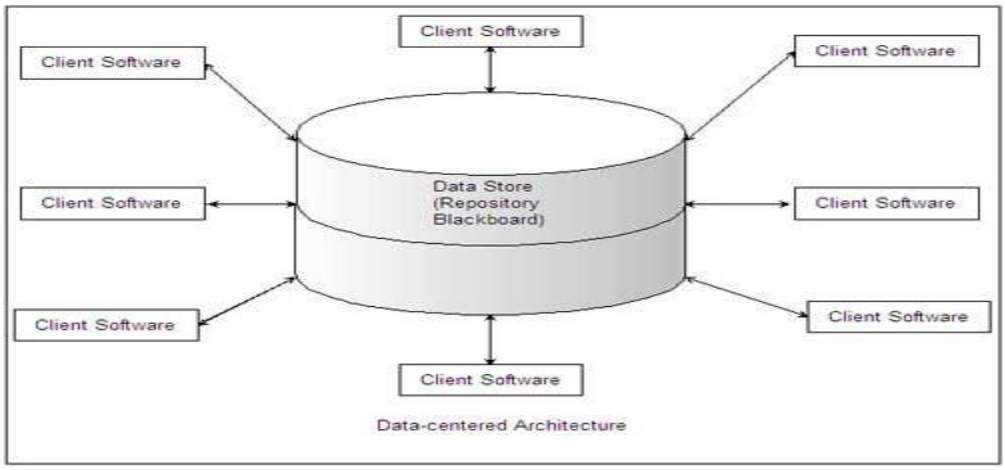
A data-centered architecture has two distinct components: a **central data structure** or data store (central repository) and a **collection of client software.** The datastore (for example, a database or a file) represents the current state of the data andthe client software performs several operations like add, delete, update, etc., onthe data stored in the data store. In some cases, the data storeallows the client software to access the data independent of any changes or theactions of other client software.

In this architectural style, new components corresponding to clients can be added and existing components can be modified easily without taking into account other clients. This is because client components operate independently of one another.

A variation of this architectural style is blackboard system in which the data store is transformed into a blackboard that notifies the client software when the data (of their interest) changes. In addition, the information can be transferred among the clients through the blackboard component.

Some advantages of the data-centered architecture are listed below.

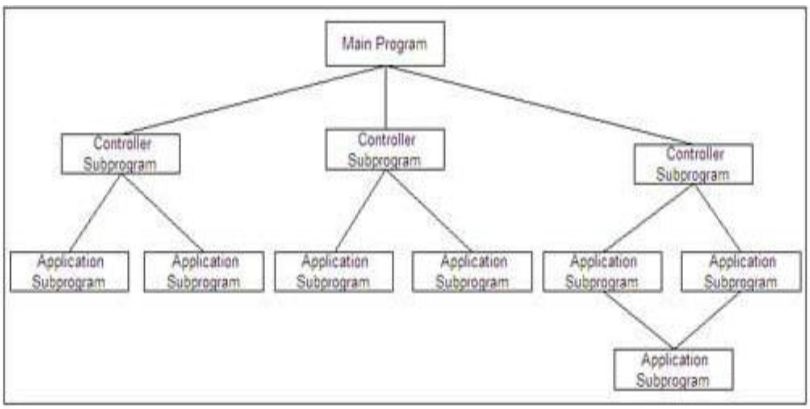
1. Clients operate independently of one another.
2. Data repository is independent of the clients.
3. It adds scalability (that is, new clients can be added easily).
4. It supports modifiability.
5. It achieves data integration in component-based development using blackboard.



Call and Return Architecture

A call and return architecture enables software designers to achieve a program structure, which can be easily modified. This style consists of the following two substyles.

1.Main program/subprogram architecture: In this, function is decomposed into a control hierarchy where the main program invokes a number of program components, which in turn may invoke other components.



2.Remote procedure call architecture: In this, components of the main or subprogram architecture are distributed over a network across multiple computers.

Software Architecture

The fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. Over time the set of significant design decisions about the organization of a software system which encompass:

(i) the selection of structural elements, their interfaces, and their collaborative behavior.

(ii) the composition of elements into progressively larger subsystems.

(iii) the architectural style that guides this organization.

(iv) system-level properties concerning usage, functionality, performance, resilience, reuse, constraints, trade- offs, and aesthetics.

(v) Roles of Architecture Description Shows the big picture about the software solution structures. Captures significant, early design decisions of a software system.

(vi) Enables to characterize or evaluate externally visible properties of the software system. (vii)Defines Constraints on the selection of implementation alternatives.

(viii)Serves as a skeleton around which full- fledged systems can be fleshed out.

Transform and Transcation Mapping:

Transform Mapping (Analysis)

A structure chart is produced by the conversion of a DFD diagram; this conversion is described as 'transform mapping (analysis)'. It is applied through the 'transforming' of input data flow into output data flow.

Transform analysis establishes the modules of the system, also known as the primary functional components, as well as the inputs and outputs of the identified modules in the DFD. Transform analysis is made up of a number of steps that need to be carried out. The first one is the dividing of the DFD into 3 parts:

- Input
- Logical processing
- Output

The '**input**' part of the DFD covers operations that change high level input data from physical to logical form e.g. from a keyboard input to storing the characters typed into a database. Each individual instance of an input s called an 'afferent branch'.

The '**output**' part of the DFD is similar to the 'input' part in that it acts as a conversion process. However, the conversion is concerned with the logical output of the system into a physical one e.g. text stored in a database converted into a printed version through a printer. Also, similar to the 'input', each individual instance of an output is called as 'efferent branch'. The remaining part of a DFD is called the central transform.

Once the above step has been conducted, transform analysis moves onto the second step; the structure chart is established by identifying one module for the central transform, the afferent and efferent branches. These are controlled by a 'root module' which acts as an 'invoking' part of the DFD.

In order to establish the highest input and output conversions in the system, a 'bubble' is drawn out. In other words, the inputs are mapped out to their outputs until an output is found that cannot be traced back to its input. Central transforms can be classified as processes that manipulate the inputs/outputs of a system e.g. sorting input, prioritizing it or filtering data. Processes which check the inputs/outputs or attach additional information to them cannot be classified as central transforms. Inputs and outputs are represented as boxes in the first level structure chart and central transforms as single boxes.

Moving on to the third step of transform analysis, sub-functions (formed from the breaking up of high-level functional components, a process called 'factoring') are added to the structure chart. The factoring process adds sub-functions that deal with error-handling and sub-functions that determine the start and end of a process.

Transform analysis is a set of design steps that allows a DFD with transform flow characteristics to be mapped into specific architectural style. These steps are as follows:

- Step1: Review the fundamental system model
- Step2: Review and refine DFD for the SW
- Step3: Assess the DFD in order to decide the usage of transform or transaction flow.
- Step4: Identify incoming and outgoing boundaries in order to establish the transform center.
- Step5: Perform "first-level factoring".
- Step6: Perform "second-level factoring".
- Step7: Refine the first-iteration architecture using design heuristics for improved SW quality.

Transaction Mapping (Analysis)

Similar to 'transform mapping', transaction analysis makes use of DFDs diagrams to establish the various transactions involved and producing a structure chart as a result.

Transaction mapping Steps:

- Review the fundamental system model
- Review and refine DFD for the SW
- Assess the DFD in order to decide the usage of transform or transaction flow.
- Identify the transaction center and the flow characteristics along each action path
- Find transaction center
- Identify incoming path and isolate action paths
- Evaluate each action path for transform vs. transaction characteristics
- Map the DFD in a program structure agreeable to transaction processing
- Carry out the 'factoring' process
- Refine the first iteration program/ architecture using design heuristics for improved SW quality

In transaction mapping a single data item triggers one of a number of information flows that affect a function implied by the triggering data item. The data item implies a transaction.

User Interface Design:

The visual part of a computer application or operating system through which a client interacts with a computer or software. It determines how commands are given to the computer or the program and how data is displayed on the screen.

Types of User Interface

There are two main types of User Interface:

- Text-Based User Interface or Command Line Interface
- Graphical User Interface (GUI)

Text-Based User Interface: This method relies primarily on the keyboard. A typical example of this is UNIX.

Advantages

- Many and easier to customizations options.
- Typically capable of more important tasks.

Disadvantages

- Relies heavily on recall rather than recognition.
- Navigation is often more difficult.

Graphical User Interface (GUI): GUI relies much more heavily on the mouse. A typical example of this type of interface is any versions of the Windows operating systems.

GUI Characteristics

Characteristics	Descriptions
Windows	Multiple windows allow different information to be displayed simultaneously on the user's screen.
Icons	Icons different types of information. On some systems, icons represent files. On other icons describes processes.
Menus	Commands are selected from a menu rather than typed in a command language.
Pointing	A pointing device such as a mouse is used for selecting choices from a menu or indicating items of interests in a window.
Graphics	Graphics elements can be mixed with text or the same display.

Table 3.3

Advantages

- Less expert knowledge is required to use it.
- Easier to Navigate and can look through folders quickly in a guess and check manner.
- The user may switch quickly from one task to another and can interact with several different applications.

Disadvantages

- Typically decreased options.
- Usually less customizable. Not easy to use one button for tons of different variations.

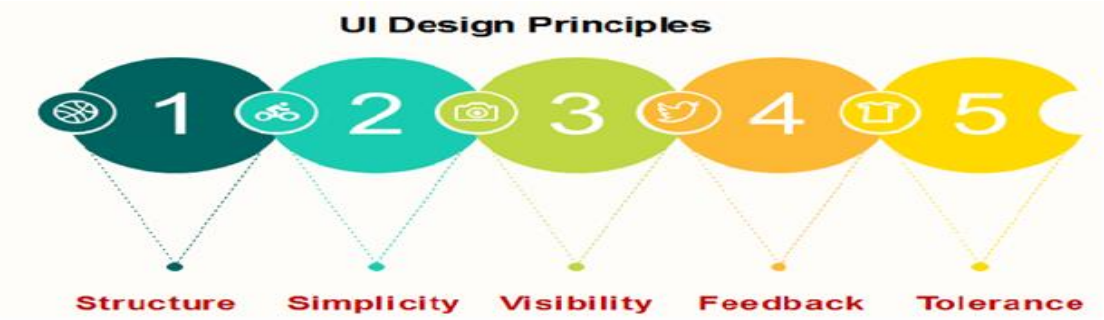


Fig 3.15

Structure: Design should organize the user interface purposefully, in the meaningful and usual based on precise, consistent models that are apparent and recognizable to users, putting related things together and separating unrelated things, differentiating dissimilar things and making similar things resemble one another. The structure principle is concerned with overall user interface architecture.

Simplicity: The design should make the simple, common task easy, communicating clearly and directly in the user's language, and providing good shortcuts that are meaningfully related to longer procedures.

Visibility: The design should make all required options and materials for a given function visible without distracting the user with extraneous or redundant data.

Feedback: The design should keep users informed of actions or interpretation, changes of state or condition, and bugs or exceptions that are relevant and of interest to the user through clear, concise, and unambiguous language familiar to users.

Tolerance: The design should be flexible and tolerant, decreasing the cost of errors and misuse by allowing undoing and redoing while also preventing bugs wherever possible by tolerating varied inputs and sequences and by interpreting all reasonable actions.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – IV – Software Engineering – SCS1305

IV BASIC CONCEPTS OF SOFTWARE TESTING

Levels - Software Testing Fundamentals - Types of s/w test - White box testing- Basis path testing - Black box testing - Control Structure testing- Regression testing - Testing in the large- S/W testing strategies - Strategic approach and issues - UNIT testing - Integration testing - Validation testing - System testing and debugging. Case studies - Writing black box and white box testing.

Levels:

Tests are grouped together based on where they are added in SDLC or the by the level of detailing they contain. In general, there are four levels of testing: unit testing, integration testing, system testing, and acceptance testing. The purpose of Levels of testing is to make software testing systematic and easily identify all possible test cases at a particular level.

There are many different testing levels which help to check behavior and performance for software testing. These testing levels are designed to recognize missing areas and reconciliation between the development lifecycle states. In SDLC models there are characterized phases such as requirement gathering, analysis, design, coding or execution, testing, and deployment.

All these phases go through the process of software testing levels. There are mainly four testing levels are:

1. Unit Testing
2. Integration Testing
3. System Testing
4. Acceptance Testing

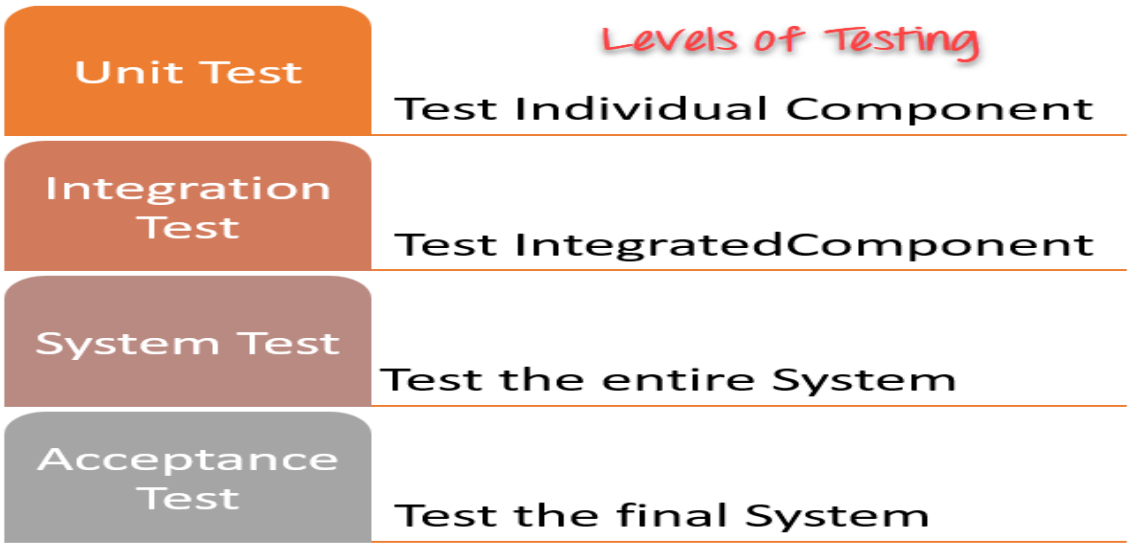


Fig 4.1 Levels

Each of these testing levels has a specific purpose. These testing level provide value to the software development lifecycle.

1) **Unit testing:**

A Unit is a smallest testable portion of system or application which can be compiled, liked, loaded, and executed. This kind of testing helps to test each module separately.

The aim is to test each part of the software by separating it. It checks that component are fulfilling functionalities or not. This kind of testing is performed by developers.

2) **Integration testing:**

Integration means combining. For Example, In this testing phase, different software modules are combined and tested as a group to make sure that integrated system is ready for system testing.

Integrating testing checks the data flow from one module to other modules. This kind of testing is performed by testers.

3) **System testing:**

System testing is performed on a complete, integrated system. It allows checking system's compliance as per the requirements. It tests the overall interaction of components. It involves load, performance, reliability and security testing.

System testing most often the final test to verify that the system meets the specification. It evaluates both functional and non-functional need for the testing.

4) **Acceptance testing:**

Acceptance testing is a test conducted to find if the requirements of a specification or contract are met as per its delivery. Acceptance testing is basically done by the user or customer. However, other stockholders can be involved in this process.

Software Testing Fundamentals:

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. In this video we describe the fundamentals of software testing.

Different types of Software Testing processes are described below:

- **Unit-Testing**
  It is a method by which individual units of source code are tested to determine if they are fit for use.
- **Integration-Testing**
  Here individual software modules are combined and tested as a group.
- **Functionality-Testing**
  It is a type of black box testing that bases its test cases on the specifications of the software component under test.
- **Usability-Testing**
  It is a technique used to evaluate a product by testing it on users.
- **System-Testing**
  It is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.
- **Performance-Testing**
  It is testing that is performed, to determine how fast some aspect of a system performs under a particular workload.
- **Load-Testing**
  It refers to the practice of modeling the expected usage of a software program by simulating multiple users accessing the program concurrently.
- **Stress-Testing**
  It is a form of testing that is used to determine the stability of a given system or entity.

White Box Testing

It is testing of a software solution's internal structure, design, and coding. In this type of testing, the code is visible to the tester. It focuses primarily on verifying the flow of inputs and outputs through the application, improving design and usability, strengthening security. White box testing is also known as Clear Box testing, Open Box testing, Structural testing, Transparent Box testing, Code-Based testing, and Glass Box testing. It is usually performed by developers.

It is one of two parts of the **Box Testing** approach to software testing. Its counterpart, **Blackbox testing**, involves testing from an external or end-user type perspective. On the other hand, Whitebox testing is based on the inner workings of an application and revolves around internal testing.

The term "WhiteBox" was used because of the see-through box concept. The clear box or WhiteBox name symbolizes the ability to see through the software's outer shell (or "box") into its inner workings. Likewise, the "black box" in "Black Box Testing" symbolizes not being able to see the inner workings of the software so that only the end-user experience can be tested.

White box testing involves the testing of the software code for the following:

- Internal security holes
- Broken or poorly structured paths in the coding processes
- The flow of specific inputs through the code
- Expected output

- The functionality of conditional loops
- Testing of each statement, object, and function on an individual basis

The testing can be done at system, integration and unit levels of software development. One of the basic goals of white box testing is to verify a working flow for an application. It involves testing a series of predefined inputs against expected or desired outputs so that when a specific input does not result in the expected output, you have encountered a bug.

Perform White Box Testing

To give you a simplified explanation of white box testing, we have divided it into **two basic steps**. This is what testers do when testing an application using the white box testing technique:

Step 1) Understand the Source Code

The first thing a tester will often do is learn and understand the source code of the application. Since white box testing involves the testing of the inner workings of an application, the tester must be very knowledgeable in the programming languages used in the applications they are testing. Also, the testing person must be highly aware of secure coding practices. Security is often one of the primary objectives of testing software. The tester should be able to find security issues and prevent attacks from hackers and naive users who might inject malicious code into the application either knowingly or unknowingly.

Step 2) CREATE TEST CASES AND EXECUTE

The second basic step to white box testing involves testing the application's source code for proper flow and structure. One way is by writing more code to test the application's source code. The tester will develop little tests for each process or series of processes in the application. This method requires that the tester must have intimate knowledge of the code and is often done by the developer. Other methods include Manual Testing, trial, and error testing and the use of testing tools as we will explain further on in this article.

WhiteBox Testing Example

Consider the following piece of code

```
Printme (int a, int b) {                    ----------  Printme is a function
int result = a+ b;
   If (result> 0)
          Print ("Positive", result)
   Else
          Print ("Negative", result)
   }                                ----------   End of the source code
```

The goal of WhiteBox testing is to verify all the decision branches, loops, statements in the code.

To exercise the statements in the above code, WhiteBox test cases would be

- A = 1, B = 1
- A = -1, B = -3

White Box Testing Techniques

A major White box testing technique is Code Coverage analysis. Code Coverage analysis eliminates gaps in a Test Case suite. It identifies areas of a program that are not exercised by a set of test cases. Once gaps are identified, you create test cases to verify untested parts of the code, thereby increasing the quality of the software product

There are automated tools available to perform Code coverage analysis. Below are a few coverage analysis techniques

Statement Coverage:- This technique requires every possible statement in the code to be tested at least once during the testing process of software engineering.

Branch Coverage - This technique checks every possible path (if-else and other conditional loops) of a software application.

Apart from above, there are numerous coverage types such as Condition Coverage, Multiple Condition Coverage, Path Coverage, Function Coverage etc. Each technique has its own merits and attempts to test (cover) all parts of software code. Using Statement and Branch coverage you generally attain 80-90% code coverage which is sufficient.

Types of White Box Testing

White box testing encompasses several testing types used to evaluate the usability of an application, block of code or specific software package. There are listed below --

- **Unit Testing:** It is often the first type of testing done on an application. Unit Testing is performed on each unit or block of code as it is developed. Unit Testing is essentially done by the programmer. As a software developer, you develop a few lines of code, a single function or an object and test it to make sure it works before continuing Unit Testing helps identify a majority of bugs, early in the software development lifecycle. Bugs identified in this stage are cheaper and easy to fix.
- **Testing for Memory Leaks**: Memory leaks are leading causes of slower running applications. A QA specialist who is experienced at detecting memory leaks is essential in cases where you have a slow running software application.

Apart from above, a few testing types are part of both black box and white box testing. They are listed as below

- **White Box Penetration Testing:** In this testing, the tester/developer has full information of the application's source code, detailed network information, IP addresses involved and

all server information the application runs on. The aim is to attack the code from several angles to expose security threats
- **White Box Mutation Testing**: Mutation testing is often used to discover the best coding techniques to use for expanding a software solution.

White Box Testing Tools

Below is a list of top white box testing tools.

- ParasoftJtest
- EclEmma
- NUnit
- PyUnit
- HTMLUnit
- CppUnit

Advantages of White Box Testing

- Code optimization by finding hidden errors.
- White box tests cases can be easily automated.
- Testing is more thorough as all code paths are usually covered.
- Testing can start early in SDLC even if GUI is not available.

Disadvantages of WhiteBox Testing

- White box testing can be quite complex and expensive.
- Developers who usually execute white box test cases detest it. The white box testing by developers is not detailed can lead to production errors.
- White box testing requires professional resources, with a detailed understanding of programming and implementation.
- White-box testing is time-consuming, bigger programming applications take the time to test fully

Basis Path Testing:

Path Testing:

Path testing is a structural testing method that involves using the source code of a program in order to find every possible executable path. It helps to determine all faults lying within a piece of code. This method is designed to execute all or selected path through a computer program.

Any software program includes, multiple entry and exit points. Testing each of these points is a challenging as well as time-consuming. In order to reduce the redundant tests and to achieve maximum test coverage, basis path testing is used.

Basis Path Testing?

The basis path testing is same, but it is based on a White Box Testing method, that defines test cases based on the flows or logical path that can be taken through the program. In software engineering, Basis path testing involves execution of all possible blocks in a program and achieves maximum path coverage with the least number of test cases. It is a hybrid of branch testing and path testing methods.

The objective behind basis path in software testing is that it defines the number of independent paths, thus the number of test cases needed can be defined explicitly (maximizes the coverage of each test case).

Here we will take a simple example, to get a better idea what is basis path testing include

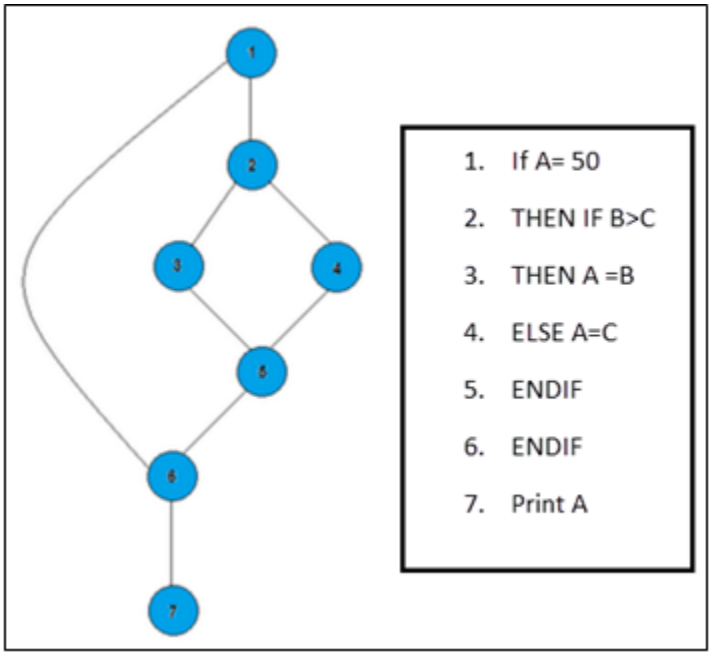


Fig 4.2 Path Testing example

In the above example, we can see there are few conditional statements that is executed depending on what condition it suffice. Here there are 3 paths or condition that need to be tested to get the output,

- **Path 1**: 1,2,3,5,6, 7
- **Path 2**: 1,2,4,5,6, 7
- **Path 3**: 1, 6, 7

Steps for Basis Path testing

The basic steps involved in basis path testing include

- Draw a control graph (to determine different program paths)
- Calculate Cyclomatic complexity (metrics to determine the number of independent paths)
- Find a basis set of paths
- Generate test cases to exercise each path

Advantages of Basic Path Testing

- It helps to reduce the redundant tests
- It focuses attention on program logic
- It helps facilitates analytical versus arbitrary case design
- Test cases which exercise basis set will execute every statement in a program at least once

Black Box Testing:

It is defined as a testing technique in which functionality of the Application Under Test (AUT) is tested without looking at the internal code structure, implementation details and knowledge of internal paths of the software. This type of testing is based entirely on software requirements and specifications. In BlackBox Testing we just focus on inputs and output of the software system without bothering about internal knowledge of the software program.

Fig 4.3 Black box testing

The above Black-Box can be any software system you want to test. For Example, an operating system like Windows, a website like Google, a database like Oracle or even your own custom application. Under Black Box Testing, you can test these applications by just focusing on the inputs and outputs without knowing their internal code implementation.

BlackBox Testing Steps:

Here are the generic steps followed to carry out any type of Black Box Testing.

- Initially, the requirements and specifications of the system are examined.
- Tester chooses valid inputs (positive test scenario) to check whether SUT processes them correctly. Also, some invalid inputs (negative test scenario) are chosen to verify that the SUT is able to detect them.
- Tester determines expected outputs for all those inputs.
- Software tester constructs test cases with the selected inputs.
- The test cases are executed.
- Software tester compares the actual outputs with the expected outputs.

- Defects if any are fixed and re-tested.

Types of Black Box Testing

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

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

Tools used for Black Box Testing:

Tools used for Black box testing largely depends on the type of black box testing you are doing.

- For Functional/ Regression Tests you can use - QTP, Selenium
- For Non-Functional Tests, you can use - LoadRunner, Jmeter

Black Box Testing Techniques

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

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

Black Box Testing	White Box Testing
the main focus of black box testing is on the validation of your functional requirements.	White Box Testing (Unit Testing) validates internal structure and working of your software code
Black box testing gives abstraction from code and focuses on testing effort on the software system behavior.	To conduct White Box Testing, knowledge of underlying programming language is essential. Current day software systems use a variety of programming languages and technologies and its not possible to know all of them.
Black box testing facilitates testing communication amongst modules	White box testing does not facilitate testing communication amongst modules

Table 4.1 Black box vs White box

Black Box Testing and Software Development Life Cycle (SDLC)

Black box testing has its own life cycle called Software Testing Life Cycle (STLC) and it is relative to every stage of Software Development Life Cycle of Software Engineering.

- **Requirement** - This is the initial stage of SDLC and in this stage, a requirement is gathered. Software testers also take part in this stage.
- **Test Planning & Analysis** - Testing Types applicable to the project are determined. A Test Plan is created which determines possible project risks and their mitigation.
- **Design** - In this stage Test cases/scripts are created on the basis of software requirement documents
- **Test Execution**- In this stage Test Cases prepared are executed. Bugs if any are fixed and re-tested.

Control structure testing

Control structure testing is a part of white box testing. it includes following methods:

1) Condition testing

2) Loop testing

3) Data validation testing.

4) Branch testing/Path testing.

Control structure testing is a group of white-box testing methods.

Condition Testing

- It is a test case design method.
- It works on logical conditions in program module.
- It involves testing of both relational expressions and arithmetic expressions.
- If a condition is incorrect, then at least one component of the condition is incorrect.

Types of errors in condition testing are

- boolean operator errors
- boolean variable errors
- boolean parenthesis errors
- relational operator errors
- arithmetic expression errors

Simple condition: Boolean variable or relational expression, possibly proceeded by a NOT operator.

Compound condition: It is composed of two or more simple conditions, Boolean operators and parentheses.

Boolean expression: It is a condition without Relational expressions.

Data Flow Testing:

Data flow testing method is effective for error protection because it is based on the relationship between statements in the program according to the definition and uses of variables.

Test paths are selected according to the location of definitions and uses of variables in the program.

It is unrealistic to assume that data flow testing will be used extensively when testing a large system, However, it can be used in a targeted fashion for areas of software that are suspect.

Loop Testing :

- Loop testing method concentrates on validity of the loop structures.

- Loops are fundamental to many algorithms and need thorough testing.

- Loops can be defined as simple, concatenated, nested, and unstructured.

Simple loops :

- The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.
- Skip the loop entirely.
- Only one pass through the loop.
- Two passes through the loop.
- M passes through the loop where m < n
- n –1, n, n + 1 passes through the loop.

Nested loops : If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests.

1. Start at the innermost loop. Set all other loops to minimum values.

2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iter

3.Conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.

4. Continue until all loops have been tested.

Concatenated loops: Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent

Unstructured loops: Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs

Validation Testing:

The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.

Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.

Path Testing:

Path testing is a structural testing method that involves using the source code of a program in order to find every possible executable path. It helps to determine all faults lying within a piece of code. This method is designed to execute all or selected path through a computer program.

Any software program includes, multiple entry and exit points. Testing each of these points is a challenging as well as time-consuming. In order to reduce the redundant tests and to achieve maximum test coverage, basis path testing is used.

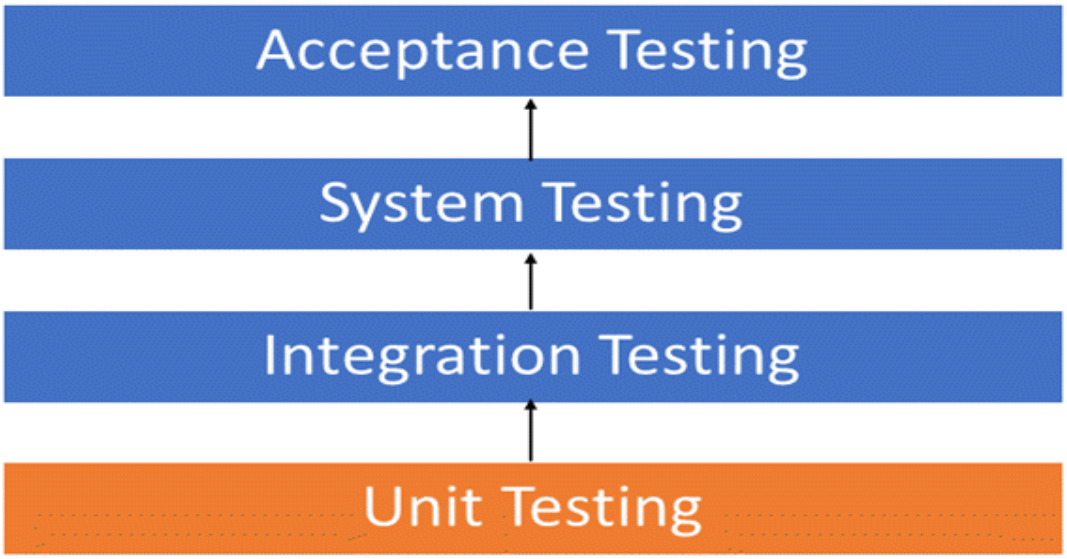
Unit Testing:

It is a type of software testing where individual units or components of a software are tested. The purpose is to validate that each unit of the software code performs as expected. Unit Testing is done during the development (coding phase) of an application by the developers. Unit Tests isolate a section of code and verify its correctness. A unit may be an individual function, method, procedure, module, or object.

In SDLC, STLC, V Model, Unit testing is first level of testing done before integration testing. Unit testing is a WhiteBox testing technique that is usually performed by the developer. Though, in a practical world due to time crunch or reluctance of developers to tests, QA engineers also do unit testing.

Need of Unit Testing:

Sometimes software developers attempt to save time by doing minimal unit testing. This is a myth because skipping on unit testing leads to higher Defect fixing costs during System Testing, Integration Testing and even Beta Testing after the application is completed. Proper unit testing done during the development stage saves both time and money in the end. Here, are key reasons to perform unit testing.



Unit Testing Levels

1. Unit tests help to fix bugs early in the development cycle and save costs.
2. It helps the developers to understand the code base and enables them to make changes quickly
3. Good unit tests serve as project documentation
4. Unit tests help with code re-use. Migrate both your code **and** your tests to your new project. Tweak the code until the tests run again.

Need of Unit Testing:

Unit Testing is of two types

- Manual
- Automated

Unit testing is commonly automated but may still be performed manually. Software Engineering does not favor one over the other but automation is preferred. A manual approach to unit testing may employ a step-by-step instructional document.

Under the automated approach-

- A developer writes a section of code in the application just to test the function. They would later comment out and finally remove the test code when the application is deployed.
- A developer could also isolate the function to test it more rigorously. This is a more thorough unit testing practice that involves copy and paste of code to its own testing environment than its natural environment. **Isolating the code helps in revealing unnecessary dependencies between the code being tested and other units or data spaces** in the product. These dependencies can then be eliminated.

- A coder generally uses a UnitTest Framework to develop automated test cases. Using an automation framework, the developer codes criteria into the test to verify the correctness of the code. During execution of the test cases, the framework logs failing test cases. Many frameworks will also automatically flag and report, in summary, these failed test cases. Depending on the severity of a failure, the framework may halt subsequent testing.
- The workflow of Unit Testing is 1) Create Test Cases 2) Review/Rework 3) Baseline 4) Execute Test Cases.

Unit Testing Techniques

Code coverage techniques used in united testing are listed below:

- Statement Coverage
- Decision Coverage
- Branch Coverage
- Condition Coverage
- Finite State Machine Coverage

Unit Testing Example: Mock Objects

Unit testing relies on mock objects being created to test sections of code that are not yet part of a complete application. Mock objects fill in for the missing parts of the program.

For example, you might have a function that needs variables or objects that are not created yet. In unit testing, those will be accounted for in the form of mock objects created solely for the purpose of the unit testing done on that section of code.

Unit Testing Tools

There are several automated tools available to assist with unit testing. We will provide a few examples below:

1. Junit: Junit is a free to use testing tool used for Java programming language. It provides assertions to identify test method. This tool test data first and then inserted in the piece of code.
2. NUnit: NUnit is widely used unit-testing framework use for all .net languages. It is an open source tool which allows writing scripts manually. It supports data-driven tests which can run in parallel.
3. JMockit: JMockit is open source Unit testing tool. It is a code coverage tool with line and path metrics. It allows mocking API with recording and verification syntax. This tool offers Line coverage, Path Coverage, and Data Coverage.
4. EMMA: EMMA is an open-source toolkit for analyzing and reporting code written in Java language. Emma support coverage types like method, line, basic block. It is Java-based so it is without external library dependencies and can access the source code.

5. PHPUnit: PHPUnit is a unit testing tool for PHP programmer. It takes small portions of code which is called units and test each of them separately. The tool also allows developers to use pre-define assertion methods to assert that a system behave in a certain manner.

Those are just a few of the available unit testing tools. There are lots more, especially for C languages and Java, but you are sure to find a unit testing tool for your programming needs regardless of the language you use.

Test Driven Development (TDD) & Unit Testing

Unit testing in TDD involves an extensive use of testing frameworks. A unit test framework is used in order to create automated unit tests. Unit testing frameworks are not unique to TDD, but they are essential to it. Below we look at some of what TDD brings to the world of unit testing:

- Tests are written before the code
- Rely heavily on testing frameworks
- All classes in the applications are tested
- Quick and easy integration is made possible

Unit Testing Myth

Myth: It requires time, and I am always overscheduled
My code is rock solid! I do not need unit tests.

Myths by their very nature are false assumptions. These assumptions lead to a vicious cycle as follows –

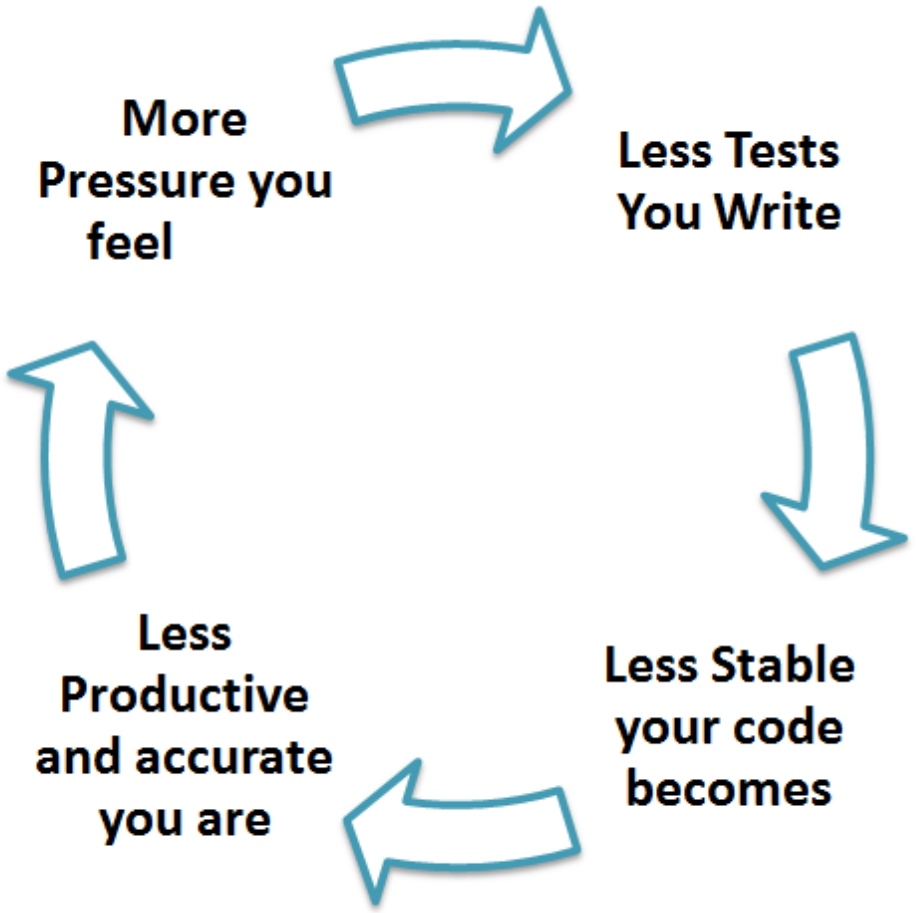


Fig 4.4 Unit testing scenario

Truth is Unit testing increase the speed of development.

Programmers think that Integration Testing will catch all errors and do not execute the unit test. Once units are integrated, very simple errors which could have very easily found and fixed in unit tested take a very long time to be traced and fixed.

Advantages:

- Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit API.
- Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (i.e. Regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified and fixed.

- Due to the modular nature of the unit testing, we can test parts of the project without waiting for others to be completed.

Disadvantages:

- Unit testing can't be expected to catch every error in a program. It is not possible to evaluate all execution paths even in the most trivial programs
- Unit testing by its very nature focuses on a unit of code. Hence it can't catch integration errors or broad system level errors.

Regression Testing:

Itis defined as a type of software testing to confirm that a recent program or code change has not adversely affected existing features.

Regression Testing is nothing but a full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine.

This testing is done to make sure that new code changes should not have side effects on the existing functionalities. It ensures that the old code still works once the latest code changes are done.

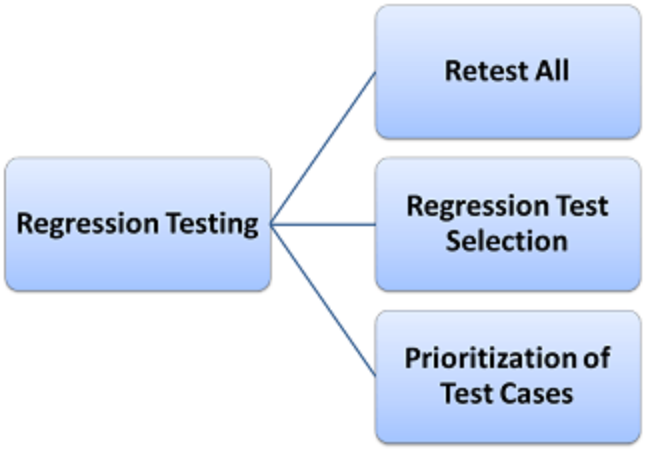
Need of Regression Testing

Regression Testing is required when there is a

- Change in requirements and code is modified according to the requirement
- New feature is added to the software
- Defect fixing
- Performance issue fix

Regression Testing procedure:

Software maintenance is an activity which includes enhancements, error corrections, optimization and deletion of existing features. These modifications may cause the system to work incorrectly. Therefore, Regression Testing becomes necessary. Regression Testing can be carried out using the following techniques:



Retest All

- This is one of the methods for Regression Testing in which all the tests in the existing test bucket or suite should be re-executed. This is very expensive as it requires huge time and resources.

Regression Test Selection

- Instead of re-executing the entire test suite, it is better to select part of the test suite to be run
- Test cases selected can be categorized as 1) Reusable Test Cases 2) Obsolete Test Cases.
- Re-usable Test cases can be used in succeeding regression cycles.
- Obsolete Test Cases can't be used in succeeding cycles.

Prioritization of Test Cases

- Prioritize the test cases depending on business impact, critical & frequently used functionalities. Selection of test cases based on priority will greatly reduce the regression test suite.

Selecting test cases for regression testing

It was found from industry data that a good number of the defects reported by customers were due to last minute bug fixes creating side effects and hence selecting the Test Case for regression testing is an art and not that easy. Effective Regression Tests can be done by selecting the following test cases -

- Test cases which have frequent defects
- Functionalities which are more visible to the users
- Test cases which verify core features of the product
- Test cases of Functionalities which has undergone more and recent changes
- All Integration Test Cases

- All Complex Test Cases
- Boundary value test cases
- A sample of Successful test cases
- A sample of Failure test cases

Regression Testing Tools

If your software undergoes frequent changes, regression testing costs will escalate.

In such cases, Manual execution of test cases increases test execution time as well as costs.

Automation of regression test cases is the smart choice in such cases.

The extent of automation depends on the number of test cases that remain re-usable for successive regression cycles.

Following are the most important tools used for both functional and regression testing in software engineering.

Ranorex Studio: all-in-one regression test automation for desktop, web, and mobile apps with built-in Selenium WebDriver. Includes a full IDE plus tools for codeless automation.

Selenium: This is an open source tool used for automating web applications. Selenium can be used for browser-based regression testing.

Quick Test Professional (QTP): HP Quick Test Professional is automated software designed to automate functional and regression test cases. It uses VBScript language for automation. It is a Data-driven, Keyword based tool.

Rational Functional Tester (RFT): IBM's rational functional tester is a Java tool used to automate the test cases of software applications. This is primarily used for automating regression test cases and it also integrates with Rational Test Manager.

Regression Testing and Configuration Management

Configuration Management during Regression Testing becomes imperative in Agile Environments where a code is being continuously modified. To ensure effective regression tests, observe the following :

- Code being regression tested should be under a configuration management tool
- No changes must be allowed to code, during the regression test phase. Regression test code must be kept immune to developer changes.
- The database used for regression testing must be isolated. No database changes must be allowed

Difference between Re-Testing and Regression Testing:

Retesting means testing the functionality or bug again to ensure the code is fixed. If it is not fixed,Defect needs to be re-opened. If fixed, Defect is closed.

Regression testing means testing your software application when it undergoes a code change to ensure that the new code has not affected other parts of the software.

Challenges in Regression Testing:

Following are the major testing problems for doing regression testing:

- With successive regression runs, test suites become fairly large. Due to time and budget constraints, the entire regression test suite cannot be executed
- Minimizing the test suite while achieving maximum Test coverage remains a challenge
- Determination of frequency of Regression Tests, i.e., after every modification or every build update or after a bunch of bug fixes, is a challenge.

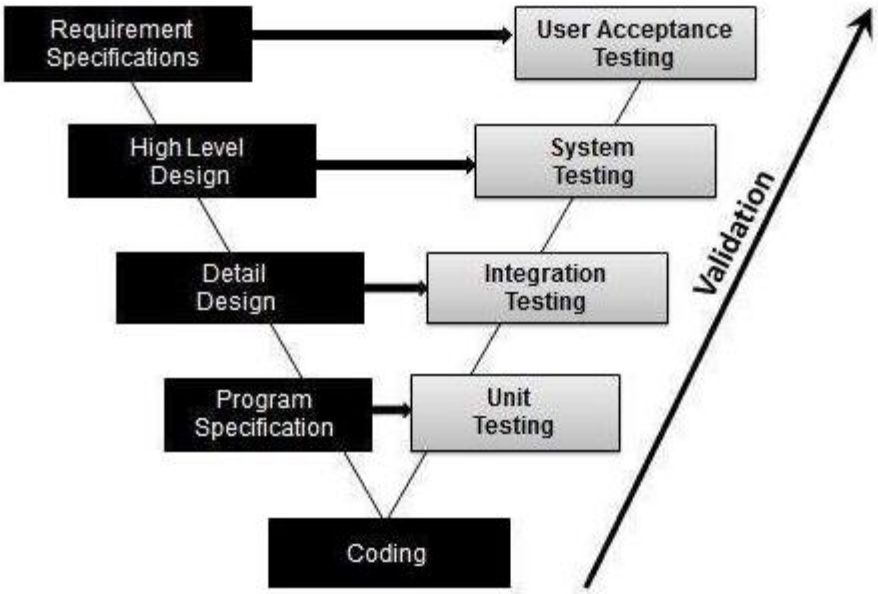
Validation Testing:

The process of evaluating software during the development process or at the end of the development process to determine whether it satisfies specified business requirements.

Validation Testing ensures that the product actually meets the client's needs. It can also be defined as to demonstrate that the product fulfills its intended use when deployed on appropriate environment.

Validation Testing - Workflow:

Validation testing can be best demonstrated using V-Model. The Software/product under test is evaluated during this type of testing.



Activities:

- Unit Testing
- Integration Testing
- System Testing
- User Acceptance Testing

System Testing:

Itis a level of testing that validates the complete and fully integrated software product. The purpose of a system test is to evaluate the end-to-end system specifications. Usually, the software is only one element of a larger computer-based system. Ultimately, the software is interfaced with other software/hardware systems. System Testing is actually a series of different tests whose sole purpose is to exercise the full computer-based system.

System Testing is Blackbox

Two Category of Software Testing

- Black Box Testing
- White Box Testing

System test falls under the **black box testing** category of software testing.

White box testing is the testing of the internal workings or code of a software application. In contrast, black box or System Testing is the opposite. System test involves the external workings of the software from the user's perspective.

System Testing involves testing the software code for following

- Testing the fully integrated applications including external peripherals in order to check how components interact with one another and with the system as a whole. This is also called End to End testing scenario.
- Verify thorough testing of every input in the application to check for desired outputs.
- Testing of the user's experience with the application.

That is a very basic description of what is involved in system testing. You need to build detailed test cases and test suites that test each aspect of the application as seen from the outside without looking at the actual source code.

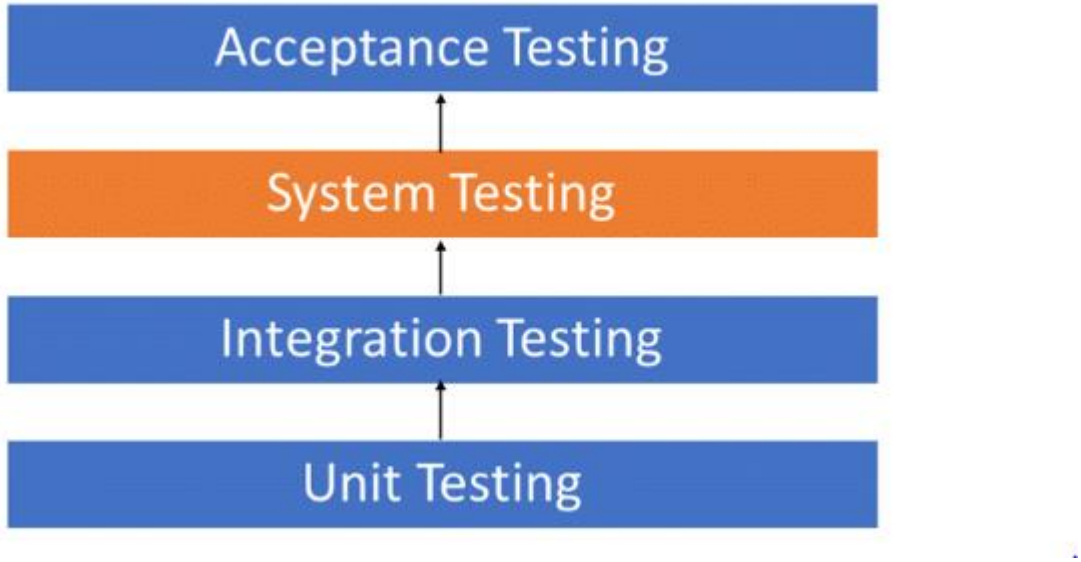


Fig 4.5 System Testing Level

As with almost any software engineering process, software testing has a prescribed order in which things should be done. The following is a list of software testing categories arranged in chronological order. These are the steps taken to fully test new software in preparation for marketing it:

- Unit testing performed on each module or block of code during development. Unit Testing is normally done by the programmer who writes the code.

- Integration testing done before, during and after integration of a new module into the main software package. This involves testing of each individual code module. One piece

of software can contain several modules which are often created by several different programmers. It is crucial to test each module's effect on the entire program model.

- System testing done by a professional testing agent on the completed software product before it is introduced to the market.

- Acceptance testing - beta testing of the product done by the actual end users.

Different Types of System Testing

Below we have listed types of system testing a large software development company would typically use

1. **Usability Testing**- mainly focuses on the user's ease to use the application, flexibility in handling controls and ability of the system to meet its objectives
2. **Load Testing**- is necessary to know that a software solution will perform under real-life loads.
3. **Regression Testing**- involves testing done to make sure none of the changes made over the course of the development process have caused new bugs. It also makes sure no old bugs appear from the addition of new software modules over time.
4. **Recovery testing** - is done to demonstrate a software solution is reliable, trustworthy and can successfully recoup from possible crashes.
5. **Migration testing**- is done to ensure that the software can be moved from older system infrastructures to current system infrastructures without any issues.
6. **Functional Testing** - Also known as functional completeness testing, Functional Testing involves trying to think of any possible missing functions. Testers might make a list of additional functionalities that a product could have to improve it during functional testing.
7. **Hardware/SoftwareTesting** - IBM refers to Hardware/Software testing as "HW/SW Testing". This is when the tester focuses his/her attention on the interactions between the hardware and software during system testing.

Types of System Testing Should Testers Use:

There are over 50 different types of system testing. The specific types used by a tester depend on several variables. Those variables include:

- Who the tester works for - This is a major factor in determining the types of system testing a tester will use. Methods used by large companies are different than that used by medium and small companies.

- Time available for testing - Ultimately, all 50 testing types could be used. Time is often what limits us to using only the types that are most relevant for the software project.
- Resources available to the tester - Of course some testers will not have the necessary resources to conduct a testing type. For example, if you are a tester working for a large software development firm, you are likely to have expensive automated testing software not available to others.
- Software Tester's Education- There is a certain learning curve for each type of software testing available. To use some of the software involved, a tester has to learn how to use it.
- Testing Budget - Money becomes a factor not just for smaller companies and individual software developers but large companies as well.

Debugging:

It is a systematic process of spotting and fixing the number of bugs, or defects, in a piece of software so that the software is behaving as expected. Debugging is harder for complex systems in particular when various subsystems are tightly coupled as changes in one system or interface may cause bugs to emerge in another.

Debugging is a developer activity and effective debugging is very important before testing begins to increase the quality of the system. Debugging will not give confidence that the system meets its requirements completely but testing gives confidence.

Debugging is considered to be a complex and time-consuming process since it attempts to remove errors at all the levels of testing. To perform debugging, debugger (debugging tool) is used to reproduce the conditions in which failure occurred, examine the program state, and locate the cause.

With the help of debugger, programmers trace the program execution step by step (evaluating the value of variables) and halt the execution wherever required to reset the program variables. Note that some programming language packages include a debugger for checking the code for errors while it is being written.

Some guidelines that are followed while performing debugging are discussed here.

1. Debugging is the process of solving a problem. Hence, individuals involved in debugging should understand all the causes of an error before starting with debugging.
2. No experimentation should be done while performing debugging. The experimental changes instead of removing errors often increase the problem by adding new errors in it.
3. When there is an error in one segment of a program, there is a high possibility that some other errors also exist in the program. Hence, if an error is found in one segment of a program, rest of the program should be properly examined for errors.
4. It should be ensured that the new code added in a program to fix errors is correct and is not introducing any new error in it. Thus, to verify the correctness of a new code and to ensure that no new errors are introduced, regression testing should be performed.

The Debugging Process

During debugging, errors are encountered that range from less damaging (like input of an incorrect function) to catastrophic (like system failure, which lead to economic or physical damage). Note that with the increase in number of errors, the amount of effort to find their causes also increases.

Once errors are identified in a software system, to debug the problem, a number of steps are followed, which are listed below.

Defect confirmation/identification: A problem is identified in a system and a defect report is created. A software engineer maintains and analyzes this error report and finds solutions to the following questions.

Does a .defect exist in the system?

Can the defect be reproduced?

What is the expected/desired behavior of the system?

What is the actual behavior?

Defect analysis: If the defect is genuine, the next step is to understand the root cause of the problem. Generally, engineers debug by starting a debugging tool (debugger) and they try to understand the root cause of the problem by following a step-by-step execution of the program.

Defect resolution: Once the root cause of a problem is identified, the error can be resolved by making an appropriate change to the system by fixing the problem.

When the debugging process ends, the software is retested to ensure that no errors are left undetected. Moreover, it checks that no new errors are introduced in the software while making some changes to it during the debugging process.

SATHYABAMA
INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE
www.sathyabama.ac.in

SCHOOL OF COMPUTING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIT – V – Software Engineering – SCS1305

COST ESTIMATION & MAINTENENCE

Software cost estimation - COCOMO model - Quality management - Quality concepts- SQA - Software reviews - Formal technical reviews - Formal approaches of SQA and software reliability - Software maintenance - SCM - Need for SCM - Version control - Introduction to SCM process - Software configuration items. Re-Engineering - Software reengineering - Reverse engineering - Restructuring - Forward engineering.

Software Cost Estimation

For any new software project, it is necessary to know how much it will cost to develop and how much development time will it take. These estimates are needed before development is initiated, but how is this done? Several estimation procedures have been developed and are having the following attributes in common.

1. Project scope must be established in advanced.
2. Software metrics are used as a support from which evaluation is made.
3. The project is broken into small PCs which are estimated individually. To achieve true cost & schedule estimate, several option arise.
4. Delay estimation
5. Used symbol decomposition techniques to generate project cost and schedule estimates.
6. Acquire one or more automated estimation tools.

Uses of Cost Estimation

1. During the planning stage, one needs to choose how many engineers are required for the project and to develop a schedule.
2. In monitoring the project's progress, one needs to access whether the project is progressing according to the procedure and takes corrective action, if necessary.

Cost Estimation Models

A model may be static or dynamic. In a static model, a single variable is taken as a key element for calculating cost and time. In a dynamic model, all variable are interdependent, and there is no basic variable.

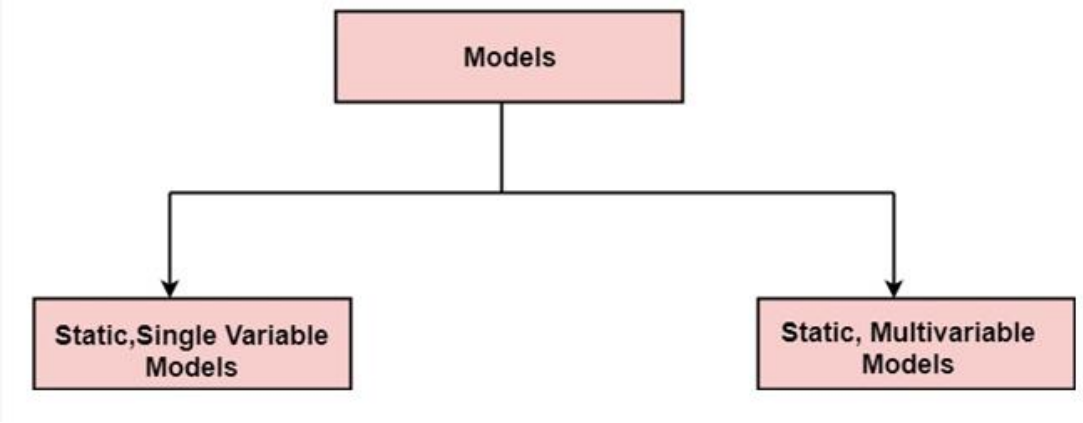


Fig 5.1 Models

Static, Single Variable Models: When a model makes use of single variables to calculate desired values such as cost, time, efforts, etc. is said to be a single variable model. The most common equation is:

$$C=aL^b$$

Where C = Costs
L= size
a and b are constants

The Software Engineering Laboratory established a model called SEL model, for estimating its software production. This model is an example of the static, single variable model.

$$E=1.4L^{0.93}$$
$$DOC=30.4L^{0.90}$$
$$D=4.6L^{0.26}$$

Where E= Efforts (Person Per Month)
DOC=Documentation (Number of Pages)
D = Duration (D, in months)
L = Number of Lines per code

Static, Multivariable Models: These models are based on method (1), they depend on several variables describing various aspects of the software development environment. In some model, several variables are needed to describe the software development process, and selected equation combined these variables to give the estimate of time & cost. These models are called multivariable models.

WALSTON and FELIX develop the models at IBM provide the following equation gives a relationship between lines of source code and effort:

$$E=5.2L^{0.91}$$

In the same manner duration of development is given by

$$\mathbf{D=4.1L^{0.36}}$$

The productivity index uses 29 variables which are found to be highly correlated productivity as follows:

Where $\mathbf{W_i}$ is the weight factor for the $\mathbf{i^{th}}$variable and $\mathbf{X_i}$**={-1,0,+1}** the estimator gives $\mathbf{X_i}$one of the values **-1, 0 or +1** depending on the variable decreases, has no effect or increases the productivity.

Example: Compare the Walston-Felix Model with the SEL model on a software development expected to involve 8 person-years of effort.

a. Calculate the number of lines of source code that can be produced.
b. Calculate the duration of the development.
c. Calculate the productivity in LOC/PY
d. Calculate the average manning

Solution:

The amount of manpower involved = 8PY=96persons-months

(a)Number of lines of source code can be obtained by reversing equation to give:

Then

$$L\ (SEL) = (96/1.4)1/0.93=94264\ LOC$$
$$L\ (SEL) = (96/5.2)1/0.91=24632\ LOC$$

(b)Duration in months can be calculated by means of equation

$$D\ (SEL) = 4.6\ (L)\ 0.26$$
$$= 4.6\ (94.264)0.26 = 15\ months$$
$$D\ (W\text{-}F) = 4.1\ L^{0.36}$$
$$= 4.1\ (24.632)0.36 = 13\ months$$

(c) Productivity is the lines of code produced per persons/month (year)

(d)Average manning is the average number of persons required per month in the project

COCOMO Model

Boehm proposed COCOMO (Constructive Cost Estimation Model) in 1981.COCOMO is one of the most generally used software estimation models in the world. COCOMO predicts the efforts and schedule of a software product based on the size of the software.

The necessary steps in this model are:

1. Get an initial estimate of the development effort from evaluation of thousands of delivered lines of source code (KDLOC).
2. Determine a set of 15 multiplying factors from various attributes of the project.
3. Calculate the effort estimate by multiplying the initial estimate with all the multiplying factors i.e., multiply the values in step1 and step2.

The initial estimate (also called nominal estimate) is determined by an equation of the form used in the static single variable models, using KDLOC as the measure of the size. To determine the initial effort E_i in person-months the equation used is of the type is shown below

E_i=a*(KDLOC)b

The value of the constant a and b are depends on the project type.

In COCOMO, projects are categorized into three types:

1. Organic
2. Semidetached
3. Embedded

Estimation of development effort

For the three classes of software products, the formulas for estimating the effort based on the code size are shown below:

Organic: Effort = 2.4(KLOC) 1.05 PM

Semi-detached: Effort = 3.0(KLOC) 1.12 PM

Embedded: Effort = 3.6(KLOC) 1.20 PM

Estimation of development time

For the three classes of software products, the formulas for estimating the development time based on the effort are given below:

Organic: Tdev = 2.5(Effort) 0.38 Months

Semi-detached: Tdev = 2.5(Effort) 0.35 Months

Embedded: Tdev = 2.5(Effort) 0.32 Months

Some insight into the basic COCOMO model can be obtained by plotting the estimated characteristics for different software sizes. Fig shows a plot of estimated effort versus product size. From fig, we can observe that the effort is somewhat superliner in the size of the software product. Thus, the effort required to develop a product increases very rapidly with project size.

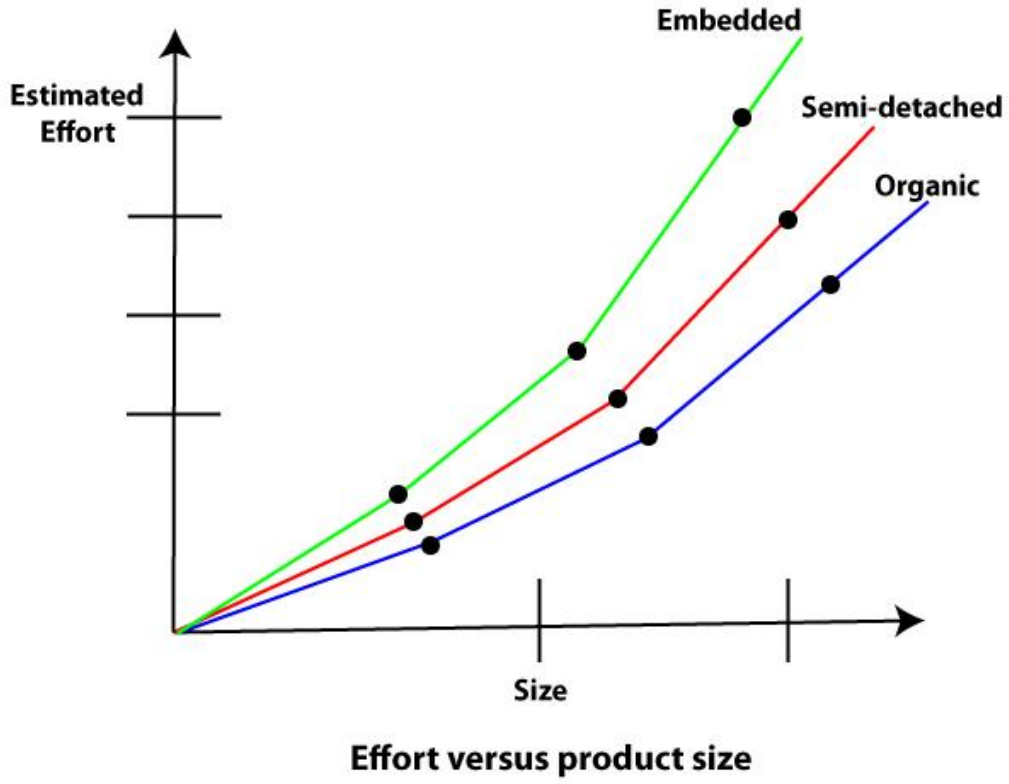


Fig 5.2 Effort vs size

The development time versus the product size in KLOC is plotted in fig. From fig it can be observed that the development time is a sub linear function of the size of the product, i.e. when the size of the product increases by two times, the time to develop the product does not double but rises moderately. This can be explained by the fact that for larger products, a larger number of activities which can be carried out concurrently can be identified. The parallel activities can be carried out simultaneously by the engineers. This reduces the time to complete the project.

Further, from fig, it can be observed that the development time is roughly the same for all three categories of products. For example, a 60 KLOC program can be developed in approximately 18 months, regardless of whether it is of organic, semidetached, or embedded type.

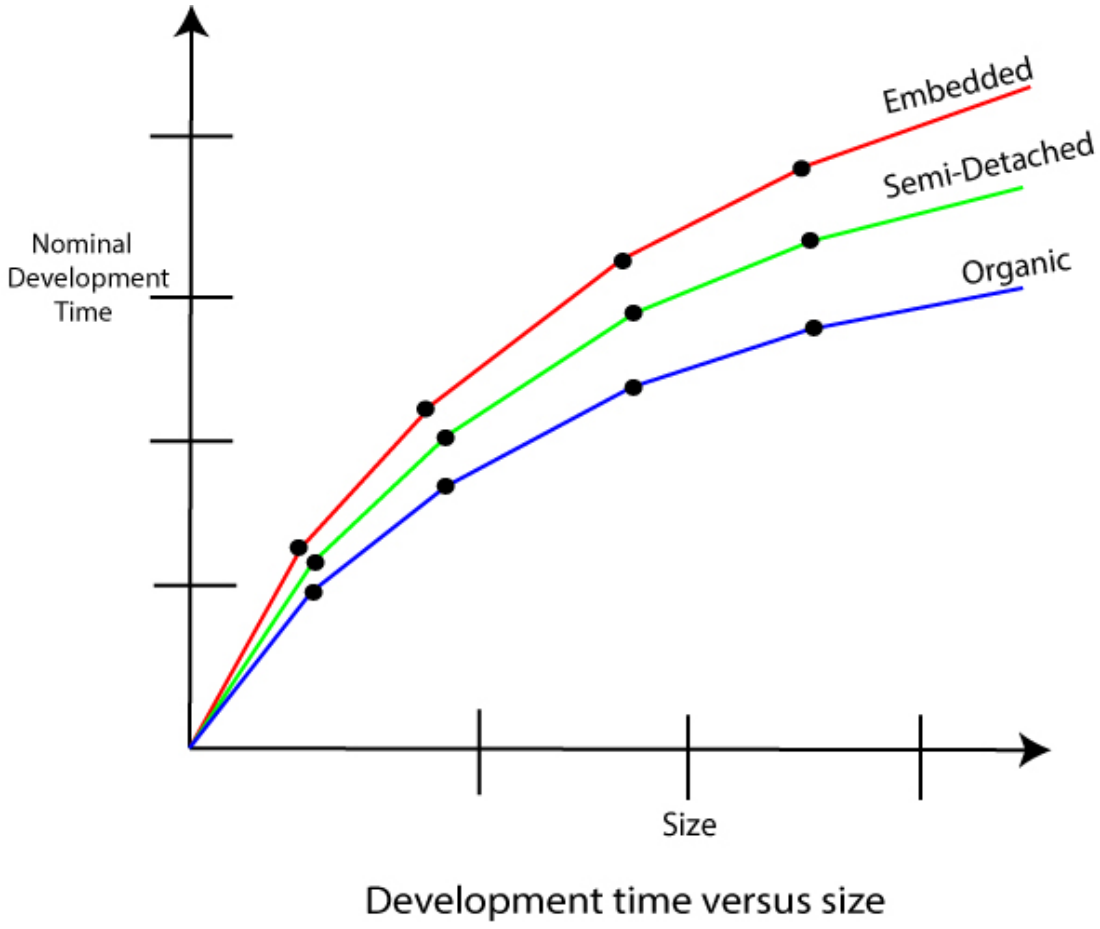


Fig 5.3 Time vs Size

From the effort estimation, the project cost can be obtained by multiplying the required effort by the manpower cost per month. But, implicit in this project cost computation is the assumption that the entire project cost is incurred on account of the manpower cost alone. In addition to manpower cost, a project would incur costs due to hardware and software required for the project and the company overheads for administration, office space, etc.

It is important to note that the effort and the duration estimations obtained using the COCOMO model are called a nominal effort estimate and nominal duration estimate. The term nominal implies that if anyone tries to complete the project in a time shorter than the estimated duration, then the cost will increase drastically. But, if anyone completes the project over a longer period of time than the estimated, then there is almost no decrease in the estimated cost value.

Example1: Suppose a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three model i.e., organic, semi-detached & embedded.

Solution: The basic COCOMO equation takes the form:

Effort=a_1*(KLOC) a_2 PM
Tdev=b_1*(efforts)b_2 Months
Estimated Size of project= 400 KLOC

(i)Organic Mode

E = 2.4 * (400)1.05 = 1295.31 PM
D = 2.5 * (1295.31)0.38=38.07 PM

(ii)Semidetached Mode

E = 3.0 * (400)1.12=2462.79 PM
D = 2.5 * (2462.79)0.35=38.45 PM

(iii) Embedded Mode

E = 3.6 * (400)1.20 = 4772.81 PM
D = 2.5 * (4772.8)0.32 = 38 PM

Example2: A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the Effort, development time, average staff size, and productivity of the project.

Solution: The semidetached mode is the most appropriate mode, keeping in view the size, schedule and experience of development time.

Hence E=3.0(200)1.12=1133.12PM
D=2.5(1133.12)0.35=29.3PM

P = 176 LOC/PM

2. Intermediate Model: The basic Cocomo model considers that the effort is only a function of the number of lines of code and some constants calculated according to the various software systems. The intermediate COCOMO model recognizes these facts and refines the initial estimates obtained through the basic COCOMO model by using a set of 15 cost drivers based on various attributes of software engineering.

Classification of Cost Drivers and their attributes:

(i) Product attributes -

- Required software reliability extent
- Size of the application database
- The complexity of the product

Hardware attributes -

- Run-time performance constraints
- Memory constraints
- The volatility of the virtual machine environment
- Required turnabout time

Personnel attributes -

- Analyst capability
- Software engineering capability
- Applications experience
- Virtual machine experience
- Programming language experience

Project attributes -

- Use of software tools
- Application of software engineering methods
- Required development schedule

The cost drivers are divided into four categories:

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very High	Extra High
Product Attributes						
RELY	0.75	0.88	1.00	1.15	1.40	..
DATA	..	0.94	1.00	1.08	1.16	..
CPLX	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME	..	..	1.00	1.11	1.30	1.66
STOR	..	..	1.00	1.06	1.21	1.56
VIRT	..	0.87	1.00	1.15	1.30	..
TURN	..	0.87	1.00	1.07	1.15	..

Table 5.1 Ratings

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Personnel Attributes						
ACAP	1.46	1.19	1.00	0.86	0.71	..
AEXP	1.29	1.13	1.00	0.91	0.82	..
PCAP	1.42	1.17	1.00	0.86	0.70	..
VEXP	1.21	1.10	1.00	0.90	..	..
LEXP	1.14	1.07	1.00	0.95	..	..
Project Attributes						
MODP	1.24	1.10	1.00	0.91	0.82	..
TOOL	1.24	1.10	1.00	0.91	0.83	..
SCED	1.23	1.08	1.00	1.04	1.10	..

Intermediate COCOMO equation:

$$E=a_i\ (KLOC)\ b_i*EAF$$

$$D=c_i\ (E)d_i$$

Coefficients for intermediate COCOMO

Project	a_i	b_i	c_i	d_i
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

3. Detailed COCOMO Model: Detailed COCOMO incorporates all qualities of the standard version with an assessment of the cost driver?s effect on each method of the softwareengineering

process. The detailed model uses various effort multipliers for each cost driver property. In detailed cocomo, the whole software is differentiated into multiple modules, and then we apply COCOMO in various modules to estimate effort and then sum the effort.

The Six phases of detailed COCOMO are:

1. Planning and requirements
2. System structure
3. Complete structure
4. Module code and test
5. Integration and test
6. Cost Constructive model

The effort is determined as a function of program estimate, and a set of cost drivers are given according to every phase of the software lifecycle.

Software Quality Management

Software Quality Management – abbreviated SQM – is a term used to describe the management aspects of developing quality software. SQM begins with an idea for a product and continues through the design, testing and launch phases.

A management process that is made up of a few different steps, SQM can be broken down most simply into three phases: Quality planning, assurance and control.

Quality Planning

Before software development begins, quality planning must take place. Quality planning involves the creation of goals and objectives for your software, as well as the creation of a strategic plan that will help you to successfully meet the objectives you lay out.

Quality planning is often considered the most important aspect of SQM, as it develops a strong blueprint for the rest of the process to follow – leading to the best-possible end product.

Quality Assurance

The quality assurance phase of SQM involves the actual building of the software program. With good SQM in place, product performance will be checked along the way to ensure that all standards are being followed. Audits may be performed and data will be collected throughout the entirety of the process.

Quality Control

The step of SQM where testing finally comes into play, quality control is in place to discover bugs, evaluate functionality and more. Depending on the results of the quality control phase, you may need to go back to development to iron out kinks and make some small final adjustments. Having a software quality management plan in place can guarantee that all industry standards are being followed and that your end-user will receive a well-developed, high quality product.

Quality Concepts

Qulaity:

Quality defines to any measurable characteristics such as correctness, maintainability, portability, testability, usability, reliability, efficiency, integrity, reusability, and interoperability.

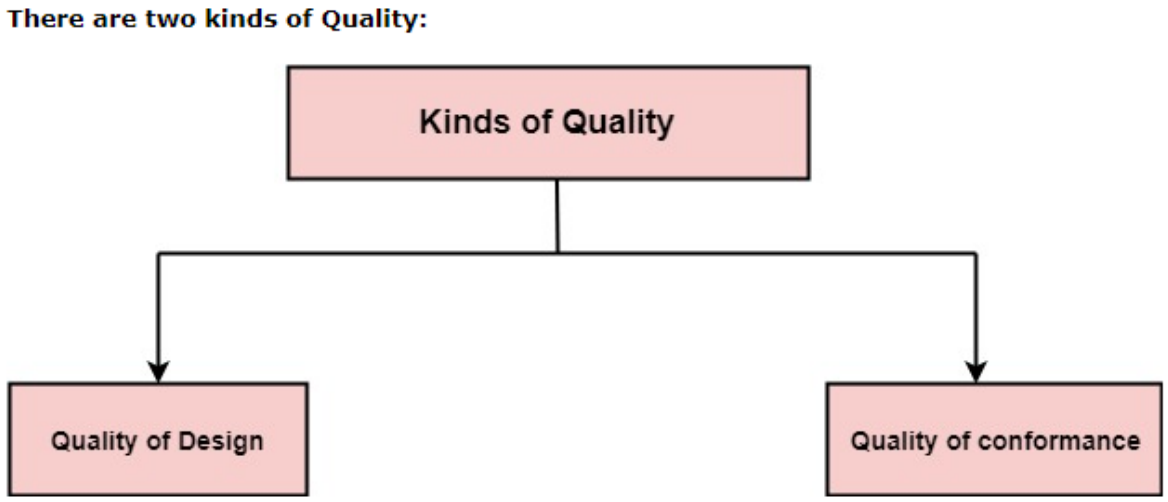


Fig 5.5 Types of quality

Quality of Design: Quality of Design refers to the characteristics that designers specify for an item. The grade of materials, tolerances, and performance specifications that all contribute to the quality of design.

Quality of conformance: Quality of conformance is the degree to which the design specifications are followed during manufacturing. Greater the degree of conformance, the higher is the level of quality of conformance.

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

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

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

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

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

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

Importance of Quality

We would expect the quality to be a concern of all producers of goods and services. However, the distinctive characteristics of software and in particular its intangibility and complexity, make special demands.

Increasing criticality of software: The final customer or user is naturally concerned about the general quality of software, especially its reliability. This is increasing in the case as organizations become more dependent on their computer systems and software is used more and more in safety-critical areas. For example, to control aircraft.

The intangibility of software: This makes it challenging to know that a particular task in a project has been completed satisfactorily. The results of these tasks can be made tangible by demanding that the developers produce 'deliverables' that can be examined for quality.

Accumulating errors during software development: As computer system development is made up of several steps where the output from one level is input to the next, the errors in the earlier ?deliverables? will be added to those in the later stages leading to accumulated determinable effects. In general the later in a project that an error is found, the more expensive it will be to fix. In addition, because the number of errors in the system is unknown, the debugging phases of a project are particularly challenging to control.

Software Quality Assurance

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

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

SQA Encompasses

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

SQA Activities

Software quality assurance is composed of a variety of functions associated with two different constituencies ?the software engineers who do technical work and an SQA group that has responsibility for quality assurance planning, record keeping, analysis, and reporting.

Following activities are performed by an independent SQA group:

1. **Prepares an SQA plan for a project:** The program is developed during project planning and is reviewed by all stakeholders. The plan governs quality assurance activities performed by the software engineering team and the SQA group. The plan identifies calculation to be performed, audits and reviews to be performed, standards that apply to the project, techniques for error reporting and tracking, documents to be produced by the SQA team, and amount of feedback provided to the software project team.
2. **Participates in the development of the project's software process description:** The software team selects a process for the work to be performed. The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g. ISO-9001), and other parts of the software project plan.
3. **Reviews software engineering activities to verify compliance with the defined software process:** The SQA group identifies, reports, and tracks deviations from the process and verifies that corrections have been made.
4. **Audits designated software work products to verify compliance with those defined as a part of the software process:** The SQA group reviews selected work products, identifies, documents and tracks deviations, verify that corrections have been made, and periodically reports the results of its work to the project manager.
5. **Ensures that deviations in software work and work products are documented and handled according to a documented procedure:** Deviations may be encountered in the project method, process description, applicable standards, or technical work products.

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

Quality Assurance	Quality Control
Quality Assurance (QA) is the set of actions including facilitation, training, measurement, and analysis needed to provide adequate confidence that processes are established and continuously improved to produce products or services that conform to specifications and are fit for use.	**Quality Control (QC)** is described as the processes and methods used to compare product quality to requirements and applicable standards, and the actions are taken when a nonconformance is detected.
QA is an activity that establishes and calculates the processes that produce the product. If there is no process, there is no role for QA.	**QC** is an activity that demonstrates whether or not the product produced met standards.
QA helps establish process	**QC** relates to a particular product or service
QA sets up a measurement program to evaluate processes	**QC** verified whether particular attributes exist, or do not exist, in a explicit product or service.
QA identifies weakness in processes and improves them	**QC** identifies defects for the primary goals of correcting errors.
Quality Assurance is a managerial tool.	Quality Control is a corrective tool.
Verification is an example of QA.	Validation is an example of QC.

Table 5.5 QA vs QC

Software Reviews:

A software review is a process or meeting during which a software product is examined by a project personnel, managers, users, customers, user representatives, or other interested parties for comment or approval.

Formal Review: Formal reviews follow a formal process. It is well structured and regulated. A formal review process consists of six main steps:

1. Planning
2. Kick-off
3. Preparation
4. Review meeting
5. Rework
6. Follow-up

1. Planning: The first phase of the formal review is the Planning phase. In this phase thereview process begins with a request for review by the author to the moderator (or inspection leader). A moderator has to take care of the scheduling like date, time, place and invitation of the review.

For the formal reviews the moderator performs the entry check and also defines the formal exit criteria.

The **entry check** is done to ensure that the reviewer's time is not wasted on a document that is not ready for review. After doing the entry check if the document is found to have very little defects then it's ready to go for the reviews. So, the **entry criteria** are to check that whether the document is ready to enter the formal review process or not. Hence the entry criteria for any document to go for the reviews are:

1.
    1.
    - The documents should not reveal a large number of major defects.
    - The documents to be reviewed should be with line numbers.
    - The documents should be cleaned up by running any automated checks that apply.
    - The author should feel confident about the quality of the document so that he can join the review team with that document.

Once, the document clear the entry check the moderator and author decides that which part of the document is to be reviewed. Since the human mind can understand only a limited set of pages at one time so in a review the maximum size is between 10 and 20 pages. Hence checking the documents improves the moderator ability to lead the meeting because it ensures the better understanding.

2. Kick-off: This kick-off meeting is an optional step in a review procedure. The goal of this step is to give a short introduction on the objectives of the review and the documents to everyone in the meeting. The relationships between the document under review and the other documents are also explained, especially if the numbers of related documents are high. At customer sites, we have measured results up to 70% more major defects found per page as a result of performing a kick-off, [van Veenendaal and van der Zwan, 2000].

3. Preparation: In this step the reviewers review the document individually using the related documents, procedures, rules and checklists provided. Each participant while reviewing individually identifies the defects, questions and comments according to their understanding of the document and role. After that all issues are recorded using a logging form. The success factor for a thorough preparation is the number of pages checked per hour. This is called the **checking rate.** Usually the checking rate is in the range of 5 to 10 pages per hour.

4. Review meeting: The review meeting consists of three phases:

- **Logging phase:** In this phase the issues and the defects that have been identified during the preparation step are logged page by page. The logging is basically done by the author or by a **scribe.** Scribe is a separate person to do the logging and is especially useful for the formal review types such as an inspection.

- Every defects and it's severity should be logged in any of the three severity
    - **Critical,Major Minor**

During the logging phase the moderator focuses on logging as many defects as possible within a certain time frame and tries to keep a good logging rate (number of defects logged per minute). In formal review meeting the good logging rate should be between one and two defects logged per minute.

- **Discussion phase:** If any issue needs discussion then the item is logged and then handled in the discussion phase. As chairman of the discussion meeting, the moderator takes care of the people issues and prevents discussion from getting too personal and calls for a break to cool down the heated discussion. The outcome of the discussions is documented for the future reference.

- **Decision phase:** At the end of the meeting a decision on the document under review has to be made by the participants, sometimes based on formal **exit criteria. Exit criteria** are the average number of critical and/or major defects found per page (for example no more than three critical/major defects per page). If the number of defects found per page is more than a certain level then the document must be reviewed again, after it has been reworked.

5. Rework: In this step if the number of defects found per page exceeds the certain level then the document has to be reworked. Not every defect that is found leads to rework. It is the author's responsibility to judge whether the defect has to be fixed. If nothing can be done about an issue then at least it should be indicated that the author has considered the issue.

6. Follow-up: In this step the moderator check to make sure that the author has taken action on all known defects. If it is decided that all participants will check the updated documents then the moderator takes care of the distribution and collects the feedback. It is the responsibility of the moderator to ensure that the information is correct and stored for future analysis.

Informal Reviews:

Informal reviews are applied many times during the early stages of the life cycle of the document. A two person team can conduct an informal review. In later stages these reviews often involve more people and a meeting. The goal is to keep the author and to improve the quality of the document. The most important thing to keep in mind about the informal reviews is that they are not documented.

Technical Review:

- It is less formal review
- It is led by the trained moderator but can also be led by a technical expert
- It is often performed as a peer review without management participation

- Defects are found by the experts (such as architects, designers, key users) who focus on the content of the document.
- In practice, technical reviews vary from quite informal to very formal

The goals of the technical review are:

1. To ensure that an early stage the technical concepts are used correctly
2. To access the value of technical concepts and alternatives in the product
3. To have consistency in the use and representation of technical concepts
4. To inform participants about the technical content of the document

Walkthroughs :

Walkthroughs are represented by the below characteristics:

- It is not a formal process/review
- It is led by the authors
- Author guide the participants through the document according to his or her thought process to achieve a common understanding and to gather feedback.
- Useful for the people if they are not from the software discipline, who are not used to or cannot easily understand software development process.
- Is especially useful for higher level documents like requirement specification, etc.

The goals of a walkthrough:

1. To present the documents both within and outside the software discipline in order to gather the information regarding the topic under documentation.
2. To explain or do the knowledge transfer and evaluate the contents of the document
3. To achieve a common understanding and to gather feedback.
4. To examine and discuss the validity of the proposed solutions

Quality:

Quality is extremely hard to define, and it is simply stated: "Fit for use or purpose." It is all about meeting the needs and expectations of customers with respect to functionality, design, reliability, durability, & price of the product.

Assurance:

Assurance is nothing but a positive declaration on a product or service, which gives confidence. It is certainty of a product or a service, which it will work well. It provides a guarantee that the product will work without any problems as per the expectations or requirements.

Quality Assurance:

Quality Assurance (QA) is defined as an activity to ensure that an organization is providing the best possible product or service to customers. QA focuses on improving the processes to deliver Quality Products to the customer. An organization has to ensure, that processes are efficient and effective as per the quality standards defined for software products. Quality Assurance is popularly known as QA Testing.

Complete Process

Quality assurance has a defined cycle called PDCA cycle or Deming cycle. The phases of this cycle are:

- Plan
- Do
- Check
- Act

Fig 5.5 QA

These above steps are repeated to ensure that processes followed in the organization are evaluated and improved on a periodic basis. Let's look into the above steps in detail -

- Plan - Organization should plan and establish the process related objectives and determine the processes that are required to deliver a high-Quality end product.
- Do - Development and testing of Processes and also "do" changes in the processes
- Check - Monitoring of processes, modify the processes, and check whether it meets the predetermined objectives
- Act - Implement actions that are necessary to achieve improvements in the processes

An organization must use Quality Assurance to ensure that the product is designed and implemented with correct procedures. This helps reduce problems and errors, in the final product.

Quality Control

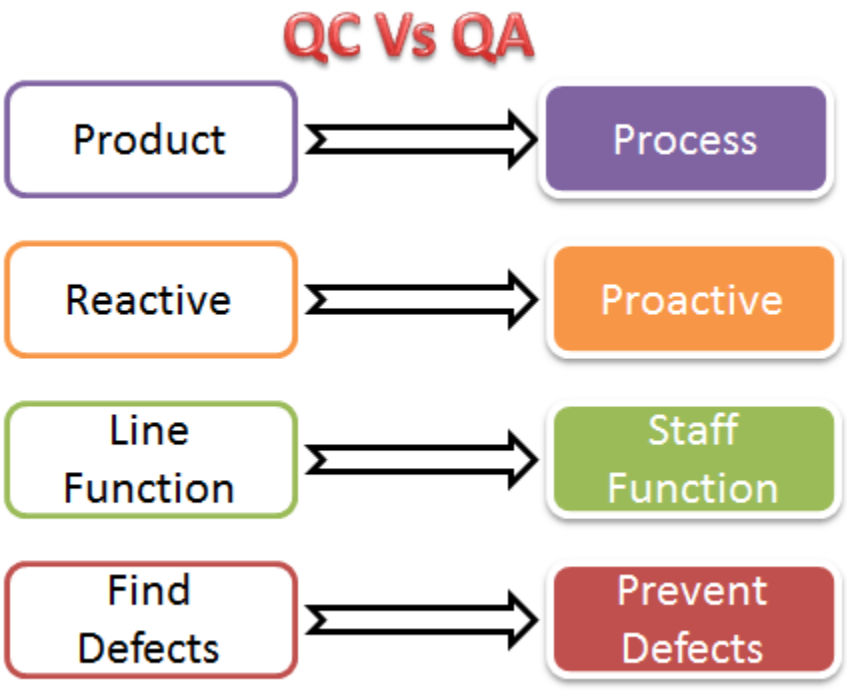
Quality control popularly abbreviated as QC. It is a Software Engineering process used to ensure quality in a product or a service. It does not deal with the processes used to create a product; rather it examines the quality of the "end products" and the final outcome.

The main aim of Quality control is to check whether the products meet the specifications and requirements of the customer. If an issue or problem is identified, it needs to be fixed before delivery to the customer.

QC also evaluates people on their quality level skill sets and imparts training and certifications. This evaluation is required for the service based organization and helps provide "perfect" service to the customers.

Quality Control and Quality Assurance

Sometimes, QC is confused with the QA. Quality control is to examine the product or service and check for the result. Quality assurance is to examine the processes and make changes to the processes which led to the end-product.



Quality Assurance Functions:

There are 5 primary Quality Assurance Functions:

1. **Technology transfer:** This function involves getting a product design document as well as trial and error data and its evaluation. The documents are distributed, checked and approved
2. **Validation:** Here validation master plan for the entire system is prepared. Approval of test criteria for validating product and process is set. Resource planning for execution of a validation plan is done.
3. **Documentation:** This function controls the distribution and archiving of documents. Any change in a document is made by adopting the proper change control procedure. Approval of all types of documents.
4. **Assuring Quality of products**
5. **Quality improvement plans**

Software Reliability Models

A software reliability model indicates the form of a random process that defines the behavior of software failures to time.

Software reliability models have appeared as people try to understand the features of how and why software fails, and attempt to quantify software reliability.

There is no individual model that can be used in all situations. No model is complete or even representative.

Most software models contain the following parts:

- Assumptions
- Factors

A mathematical function that includes the reliability with the elements. The mathematical function is generally higher-order exponential or logarithmic.

Software Reliability Modeling Techniques:

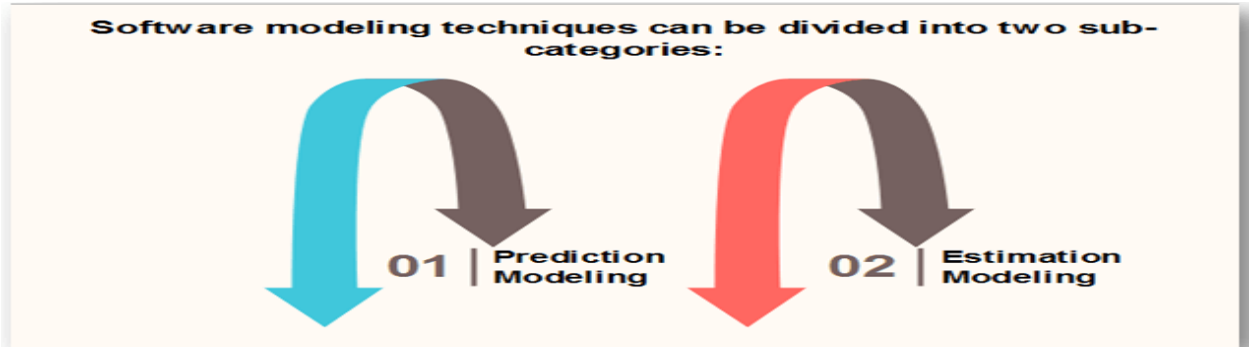


Fig 5.6 software modeling types

Both kinds of modeling methods are based on observing and accumulating failure data and analyzing with statistical inference.

Differentiate between software reliability prediction models and software reliability estimation models

Basics	Prediction Models	Estimation Models
Data Reference	Uses historical information	Uses data from the current software development effort.
When used in development cycle	Usually made before development or test phases; can be used as early as concept phase.	Usually made later in the life cycle (after some data have been collected); not typically used in concept or development phases.
Time Frame	Predict reliability at some future time.	Estimate reliability at either present or some next time.

Table 4.3 Prediction vs Estimation

Reliability Models

A reliability growth model is a numerical model of software reliability, which predicts how software reliability should improve over time as errors are discovered and repaired. These models help the manager in deciding how much efforts should be devoted to testing. The objective of the project manager is to test and debug the system until the required level of reliability is reached.

Following are the Software Reliability Models are:

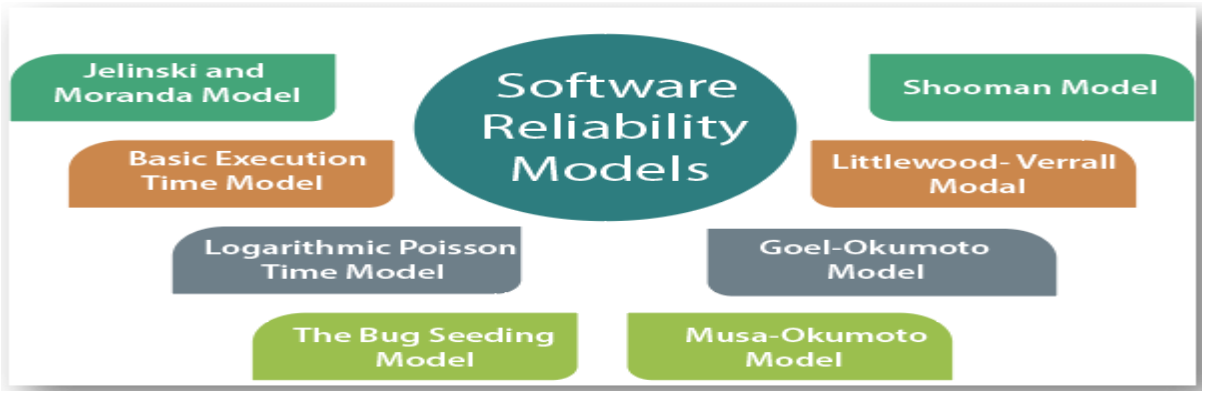


Fig 5.6 Software Reliability Models

Software Maintenance:

Software maintenance is widely accepted part of SDLC now a days. It stands for all the modifications and updations done after the delivery of software product. There are number of reasons, why modifications are required, some of them are briefly mentioned below:

- **Market Conditions** - Policies, which changes over the time, such as taxation and newly introduced constraints like, how to maintain bookkeeping, may trigger need for modification.
- **Client Requirements** - Over the time, customer may ask for new features or functions in the software.
- **Host Modifications** - If any of the hardware and/or platform (such as operating system) of the target host changes, software changes are needed to keep adaptability.
- **Organization Changes** - If there is any business level change at client end, such as reduction of organization strength, acquiring another company, organization venturing into new business, need to modify in the original software may arise.

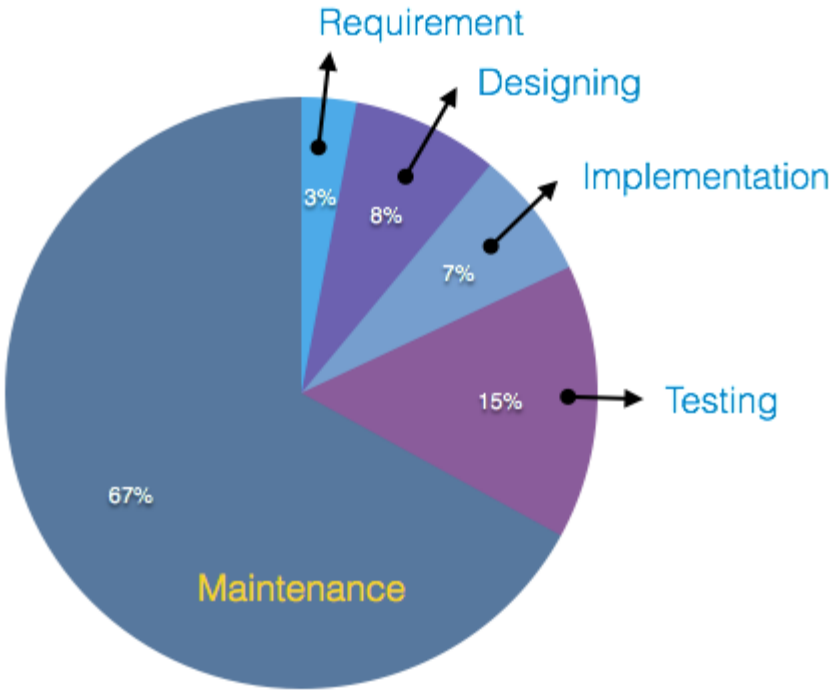
Types of maintenance

In a software lifetime, type of maintenance may vary based on its nature. It may be just a routine maintenance tasks as some bug discovered by some user or it may be a large event in itself based on maintenance size or nature. Following are some types of maintenance based on their characteristics:

- **Corrective Maintenance** - This includes modifications and updations done in order to correct or fix problems, which are either discovered by user or concluded by user error reports.
- **Adaptive Maintenance** - This includes modifications and updations applied to keep the software product up-to date and tuned to the ever changing world of technology and business environment.
- **Perfective Maintenance** - This includes modifications and updates done in order to keep the software usable over long period of time. It includes new features, new user requirements for refining the software and improve its reliability and performance.
- **Preventive Maintenance** - This includes modifications and updations to prevent future problems of the software. It aims to attend problems, which are not significant at this moment but may cause serious issues in future.

Cost of Maintenance

Reports suggest that the cost of maintenance is high. A study on estimating software maintenance found that the cost of maintenance is as high as 67% of the cost of entire software process .



On an average, the cost of software maintenance is more than 50% of all SDLC phases. There are various factors, which trigger maintenance cost go high, such as:

Real-world factors affecting Maintenance Cost

- The standard age of any software is considered up to 10 to 15 years.
- Older softwares, which were meant to work on slow machines with less memory and storage capacity cannot keep themselves challenging against newly coming enhanced softwares on modern hardware.
- As technology advances, it becomes costly to maintain old software.
- Most maintenance engineers are newbie and use trial and error method to rectify problem.
- Often, changes made can easily hurt the original structure of the software, making it hard for any subsequent changes.
- Changes are often left undocumented which may cause more conflicts in future.

Software-end factors affecting Maintenance Cost

- Structure of Software Program
- Programming Language
- Dependence on external environment
- Staff reliability and availability

Maintenance Activities

IEEE provides a framework for sequential maintenance process activities. It can be used in iterative manner and can be extended so that customized items and processes can be include.

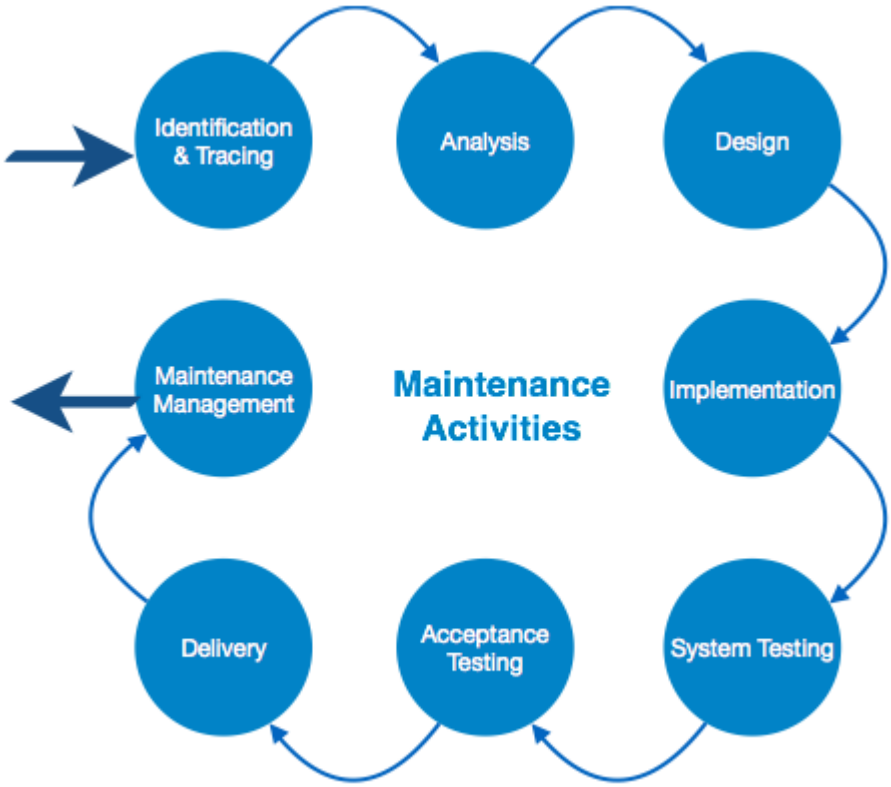


Fig 5.6-Maintanenece Activities

These activities go hand-in-hand with each of the following phase:

- **Identification & Tracing** - It involves activities pertaining to identification of requirement of modification or maintenance. It is generated by user or system may itself report via logs or error messages.Here, the maintenance type is classified also.
- **Analysis** - The modification is analyzed for its impact on the system including safety and security implications. If probable impact is severe, alternative solution is looked for. A set of required modifications is then materialized into requirement specifications. The cost of modification/maintenance is analyzed and estimation is concluded.
- **Design** - New modules, which need to be replaced or modified, are designed against requirement specifications set in the previous stage. Test cases are created for validation and verification.
- **Implementation** - The new modules are coded with the help of structured design created in the design step.Every programmer is expected to do unit testing in parallel.

- **System Testing** - Integration testing is done among newly created modules. Integration testing is also carried out between new modules and the system. Finally the system is tested as a whole, following regressive testing procedures.
- **Acceptance Testing** - After testing the system internally, it is tested for acceptance with the help of users. If at this state, user complaints some issues they are addressed or noted to address in next iteration.
- **Delivery** - After acceptance test, the system is deployed all over the organization either by small update package or fresh installation of the system. The final testing takes place at client end after the software is delivered.

  Training facility is provided if required, in addition to the hard copy of user manual.
- **Maintenance management** - Configuration management is an essential part of system maintenance. It is aided with version control tools to control versions, semi-version or patch management.

Software Configuration Management:

Software Configuration Management is defined as a process to systematically manage, organize, and control the changes in the documents, codes, and other entities during the Software Development Life Cycle. It is abbreviated as the SCM process in software engineering. The primary goal is to increase productivity with minimal mistakes.

Need of Configuration management:

The primary reasons for Implementing Software Configuration Management System are**:**

- There are multiple people working on software which is continually updating
- It may be a case where multiple version, branches, authors are involved in a software project, and the team is geographically distributed and works concurrently
- Changes in user requirement, policy, budget, schedule need to be accommodated.
- Software should able to run on various machines and Operating Systems
- Helps to develop coordination among stakeholders
- SCM process is also beneficial to control the costs involved in making changes to a system

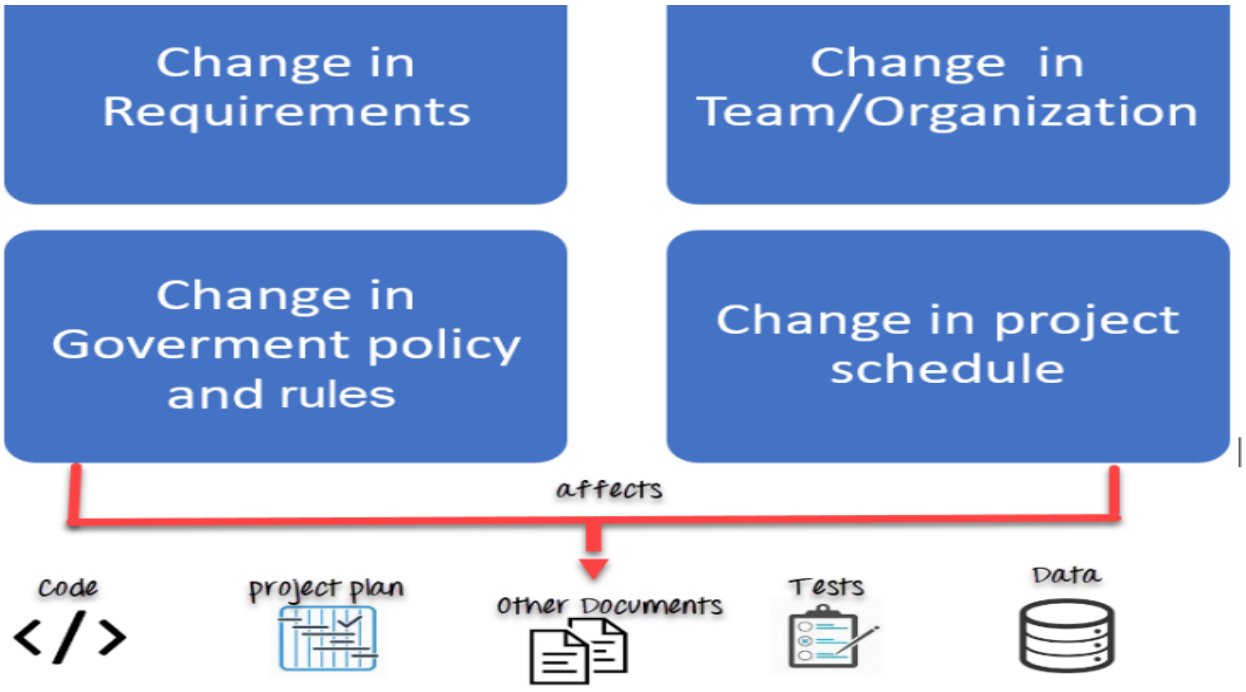


Fig 5.7 SCM

Any change in the software configuration Items will affect the final product. Therefore, changes to configuration items need to be controlled and managed.

Tasks in SCM process

Configuration Identification

Baselines

Change Control

Configuration Status Accounting

Configuration Audits and Reviews

Configuration Identification:

Configuration identification is a method of determining the scope of the software system. With the help of this step, you can manage or control something even if you don't know what it is. It is a description that contains the CSCI type (Computer Software Configuration Item), a project identifier and version information.

Activities during this process:

- Identification of configuration Items like source code modules, test case, and requirements specification.
- Identification of each CSCI in the SCM repository, by using an object-oriented approach
- The process starts with basic objects which are grouped into aggregate objects. Details of what, why, when and by whom changes in the test are made
- Every object has its own features that identify its name that is explicit to all other objects
- List of resources required such as the document, the file, tools, etc.

Example:

Instead of naming a File login.phpits should be named login_v1.2.php where v1.2 stands for the version number of the file

Instead of naming folder "Code" it should be named "Code_D" where D represents code should be backed up daily.

Baseline:

A baseline is a formally accepted version of a software configuration item. It is designated and fixed at a specific time while conducting the SCM process. It can only be changed through formal change control procedures.

Activities during this process:

- Facilitate construction of various versions of an application
- Defining and determining mechanisms for managing various versions of these work products
- The functional baseline corresponds to the reviewed system requirements
- Widely used baselines include functional, developmental, and product baselines

In simple words, baseline means ready for release.

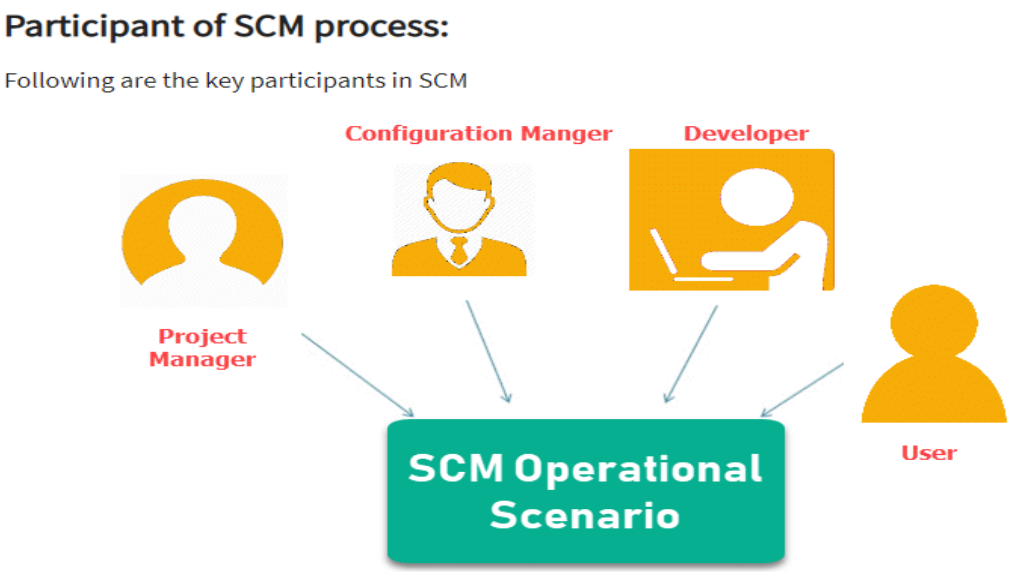


Fig 5.8 SCM operational scenario

Change Control:

Change control is a procedural method which ensures quality and consistency when changes are made in the configuration object. In this step, the change request is submitted to software configuration manager.

Activities during this process:

- Control ad-hoc change to build stable software development environment. Changes are committed to the repository
- The request will be checked based on the technical merit, possible side effects and overall impact on other configuration objects.
- It manages changes and making configuration items available during the software lifecycle

Configuration Status Accounting:

Configuration status accounting tracks each release during the SCM process. This stage involves tracking what each version has and the changes that lead to this version.

Activities during this process:

- Keeps a record of all the changes made to the previous baseline to reach a new baseline
- Identify all items to define the software configuration
- Monitor status of change requests
- Complete listing of all changes since the last baseline

- Allows tracking of progress to next baseline
- Allows to check previous releases/versions to be extracted for testing

Configuration Audits and Reviews:

Software Configuration audits verify that all the software product satisfies the baseline needs. It ensures that what is built is what is delivered.

Activities during this process:

- Configuration auditing is conducted by auditors by checking that defined processes are being followed and ensuring that the SCM goals are satisfied.
- To verify compliance with configuration control standards. auditing and reporting the changes made
- SCM audits also ensure that traceability is maintained during the process.
- Ensures that changes made to a baseline comply with the configuration status reports
- Validation of completeness and consistency

1. Configuration Manager

- Configuration Manager is the head who is Responsible for identifying configuration items.
- CM ensures team follows the SCM process
- He/She needs to approve or reject change requests

2. Developer

- The developer needs to change the code as per standard development activities or change requests. He is responsible for maintaining configuration of code.
- The developer should check the changes and resolves conflicts

3. Auditor

- The auditor is responsible for SCM audits and reviews.
- Need to ensure the consistency and completeness of release.

4. Project Manager:

- Ensure that the product is developed within a certain time frame
- Monitors the progress of development and recognizes issues in the SCM process
- Generate reports about the status of the software system
- Make sure that processes and policies are followed for creating, changing, and testing

5. User

The end user should understand the key SCM terms to ensure he has the latest version of the software

Software Configuration Management Plan

The SCMP (Software Configuration management planning) process planning begins at the early phases of a project. The outcome of the planning phase is the SCM plan which might be stretched or revised during the project.

- The SCMP can follow a public standard like the IEEE 828 or organization specific standard
- It defines the types of documents to be management and a document naming. Example Test_v1
- SCMP defines the person who will be responsible for the entire SCM process and creation of baselines.
- Fix policies for version management & change control
- Define tools which can be used during the SCM process
- Configuration management database for recording configuration information.

Software Configuration Management Tools

Any Change management software should have the following 3 Key features:

Concurrency Management:

When two or more tasks are happening at the same time, it is known as concurrent operation. Concurrency in context to SCM means that the same file being edited by multiple persons at the same time.

If concurrency is not managed correctly with SCM tools, then it may create many pressing issues.

Version Control:

SCM uses archiving method or saves every change made to file. With the help of archiving or save feature, it is possible to roll back to the previous version in case of issues.

Synchronization:

Users can checkout more than one files or an entire copy of the repository. The user then works on the needed file and checks in the changes back to the repository.They can synchronize their local copy to stay updated with the changes made by other team members.

Following are popular tools

1. Git: Git is a free and open source tool which helps version control. It is designed to handle all types of projects with speed and efficiency.

2. Team Foundation Server: Team Foundation is a group of tools and technologies that enable the team to collaborate and coordinate for building a product.

3. Ansible: It is an open source Software configuration management tool. Apart from configuration management it also offers application deployment & task automation.

Software Re-engineering

When we need to update the software to keep it to the current market, without impacting its functionality, it is called software re-engineering. It is a thorough process where the design of software is changed and programs are re-written.

Legacy software cannot keep tuning with the latest technology available in the market. As the hardware become obsolete, updating of software becomes a headache. Even if software grows old with time, its functionality does not.

For example, initially Unix was developed in assembly language. When language C came into existence, Unix was re-engineered in C, because working in assembly language was difficult.

Other than this, sometimes programmers notice that few parts of software need more maintenance than others and they also need re-engineering.

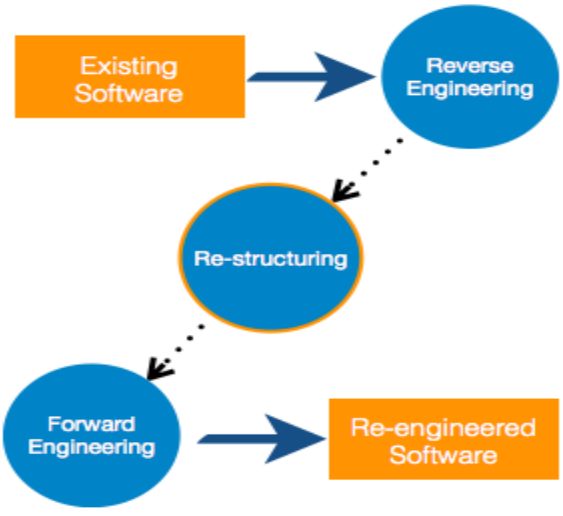


Fig 5.8 Re-Engineering

Re-Engineering Process

- **Decide** what to re-engineer. Is it whole software or a part of it?
- **Perform** Reverse Engineering, in order to obtain specifications of existing software.

- **Restructure Program** if required. For example, changing function-oriented programs into object-oriented programs.
- **Re-structure data** as required.
- **Apply Forward engineering** concepts in order to get re-engineered software.

There are few important terms used in Software re-engineering

Reverse Engineering:

It is a process to achieve system specification by thoroughly analyzing, understanding the existing system. This process can be seen as reverse SDLC model, i.e. we try to get higher abstraction level by analyzing lower abstraction levels.

An existing system is previously implemented design, about which we know nothing. Designers then do reverse engineering by looking at the code and try to get the design. With design in hand, they try to conclude the specifications. Thus, going in reverse from code to system specification.

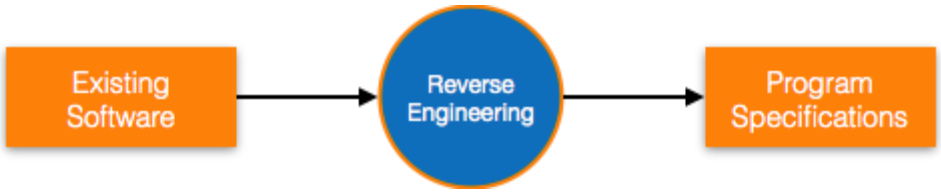


Fig 5.9 Reverse Engineering

Program Restructuring:

It is a process to re-structure and re-construct the existing software. It is all about re-arranging the source code, either in same programming language or from one programming language to a different one. Restructuring can have either source code-restructuring and data-restructuring or both.

Re-structuring does not impact the functionality of the software but enhance reliability and maintainability. Program components, which cause errors very frequently can be changed, or updated with re-structuring.

The dependability of software on obsolete hardware platform can be removed via re-structuring.

Forward Engineering

Forward engineering is a process of obtaining desired software from the specifications in hand which were brought down by means of reverse engineering. It assumes that there was some software engineering already done in the past.

Forward engineering is same as software engineering process with only one difference – it is carried out always after reverse engineering.

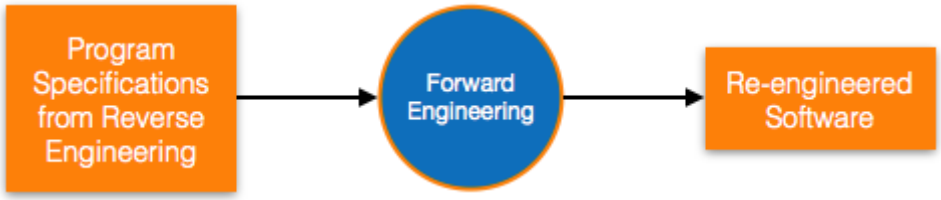


Fig 5.10 Forward Engineering

Component reusability

A component is a part of software program code, which executes an independent task in the system. It can be a small module or sub-system itself.

Example:

The login procedures used on the web can be considered as components, printing system in software can be seen as a component of the software.

Components have high cohesion of functionality and lower rate of coupling, i.e. they work independently and can perform tasks without depending on other modules.

In OOP, the objects are designed are very specific to their concern and have fewer chances to be used in some other software.

In modular programming, the modules are coded to perform specific tasks which can be used across number of other software programs.

There is a whole new vertical, which is based on re-use of software component, and is known as Component Based Software Engineering (CBSE).

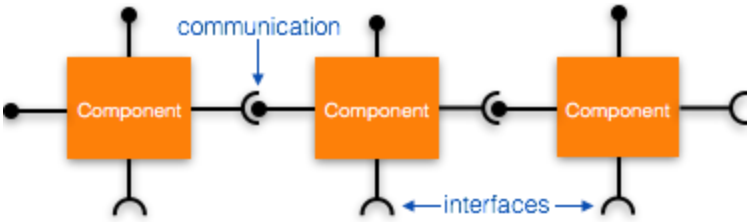


Fig 5.11- Component reusability

Re-use can be done at various levels

- **Application level** - Where an entire application is used as sub-system of new software.
- **Component level** - Where sub-system of an application is used.
- **Modules level** - Where functional modules are re-used.

Software components provide interfaces, which can be used to establish communication among different components.

Reuse Process:

Two kinds of method can be adopted: either by keeping requirements same and adjusting components or by keeping components same and modifying requirements.

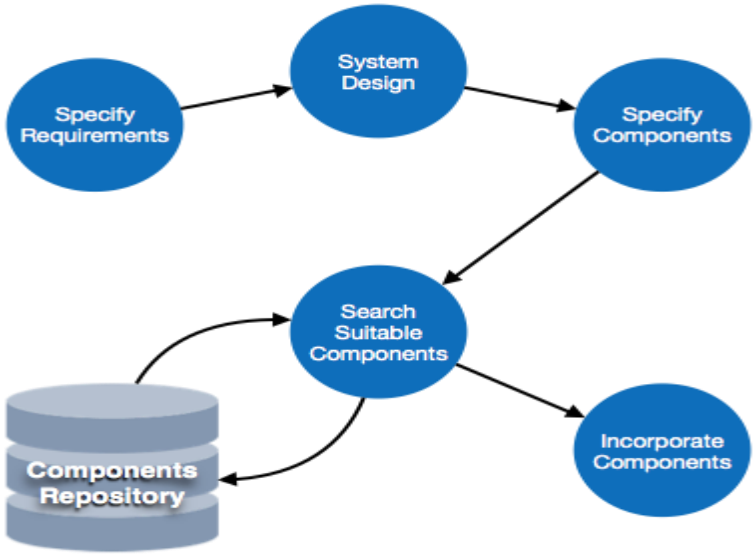


Fig 5.12 Reuse Process

- **Requirement Specification** - The functional and non-functional requirements are specified, which a software product must comply to, with the help of existing system, user input or both.
- **Design** - This is also a standard SDLC process step, where requirements are defined in terms of software parlance. Basic architecture of system as a whole and its sub-systems are created.
- **Specify Components** - By studying the software design, the designers segregate the entire system into smaller components or sub-systems. One complete software design turns into a collection of a huge set of components working together.
- **Search Suitable Components** - The software component repository is referred by designers to search for the matching component, on the basis of functionality and intended software requirements..
- **Incorporate Components** - All matched components are packed together to shape them as complete software.