Semester-3

Software Engineering

(According to Purvanchal UniversitySyllabus)

Unit – 1

Introduction

Introduction to Software Engineering–

- 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.
    - Software project management has wider scope than software engineering process as it involves communication, pre and post delivery support etc.
- 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.

Importance of Software–

- Talking about Software than it plays a major role in daily life. It is an application which gives a helping hand to the organization to manage data, resources, performance, and also manages all the internal task and communication for fruitful result.
- Software development is an umbrella term used to refer to the overall process that involves several tasks, such as computer programming, documenting, repairing and testing that concern both the creation and the maintenance of applications and frameworks. Some of the software like Performance management software, Hotel Management Software, MR reporting software, Construction Management Software and many other are crucial for the organization.

- Although software development is found for a wide variety of purposes, the one we will refer to here is custom software. This is perhaps one of the most common purposes of developing software;

The Features of 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 software has the capabilities to maintain itself This aspect briefs about how well software has the capabilities to maintain itself

This aspect briefs about how well software has the capabilities to maintain

itself in the ever-changing environment:

changing environment:

- Modularity
- Maintainability
- Flexibility
- Scalability

Software Development Life Cycle-

- Software Development Life Cycle, SDLC for short, is a well

  Software Development Life Cycle, SDLC for short, is a well-defined, structured sequence of stages in software engineering to develop the.
- SDLC provides a series of steps to be followed to design and develop a

Communication-

This is the first step where the user initiates the request for a desired software product. He contacts the service provider and tries to negotiate the terms. He submits his request to the service providing organization in writing.

Requirement Gathering-

This step onwards the software development team works to carry on the project. The team holds discussions with various stakeholders from problem domain and tries to bring out as much information as possible on their requirements. The requirements are contemplated and segregated into user requirements, system requirements and functional requirements. The requirements are collected using a number of practices as given -

- studying the existing or obsolete system and software,
- conducting interviews of users and developers,
- referring to the database or
- Collecting answers from the questionnaires.

Feasibility Study-

After requirement gathering, the team comes up with a rough plan of software process. At this step the team analyzes if software can be made to fulfill all requirements of the user and if there is any possibility of software being no more useful. It is found out, if the project is financially, practically and technologically feasible for the organization to take up. There are many algorithms available, which help the developers to conclude the feasibility of a software project.

System Analysis-

At this step the developers decide a roadmap of their plan and try to bring up the best software model suitable for the project. System analysis includes Understanding of software product limitations, learning system related problems or changes to be done in existing systems beforehand, identifying and addressing the impact of project on organization and personnel etc. The project team analyzes the scope of the project and plans the schedule and resources accordingly.

Software Design-

Next step is to bring down whole knowledge of requirements and analysis on the desk and design the software product. The inputs from users and information gathered in requirement gathering phase are the inputs of this step. The output of this step comes in the form of two designs; logical design and physical design. Engineers produce meta-data and data dictionaries, logical diagrams, data-flow diagrams and in some cases pseudo codes.

Coding-

This step is also known as programming phase. The implementation of software design starts in terms of writing program code in the suitable programming language and developing error-free executable programs efficiently.

Testing-

An estimate says that 50% of whole software development process should be tested. Errors may ruin the software from critical level to its own removal. Software testing is done while coding by the developers and thorough testing is conducted by testing experts at various levels of code such as module testing, program testing, product testing, in-house testing and testing the product at user's end. Early discovery of errors and their remedy is the key to reliable software.

Integration-

Software may need to be integrated with the libraries, databases and other program(s). This stage of SDLC is involved in the integration of software with outer world entities.

Implementation-

This means installing the software on user machines. At times, software needs post-installation configurations at user end. Software is tested for portability and adaptability and integration related issues are solved during implementation.

Operation and Maintenance-

This phase confirms the software operation in terms of more efficiency andless errors. If required, the users are trained on, or aided with the documentation on

how to operate the software and how to keep the software operational. The software is maintained timely by updating the code according to the changes taking place in user end environment or technology. This phase may face challenges from hidden bugs and real-world unidentified problems.

Disposition-

As time elapses, the software may decline on the performance front. It may go completely obsolete or may need intense up gradation. Hence a pressing need to eliminate a major portion of the system arises. This phase includes archiving data and required software components, closing down the system, planning disposition activity and terminating system at appropriate end-of-system time.

Unit – 2

Software Requirement Specification

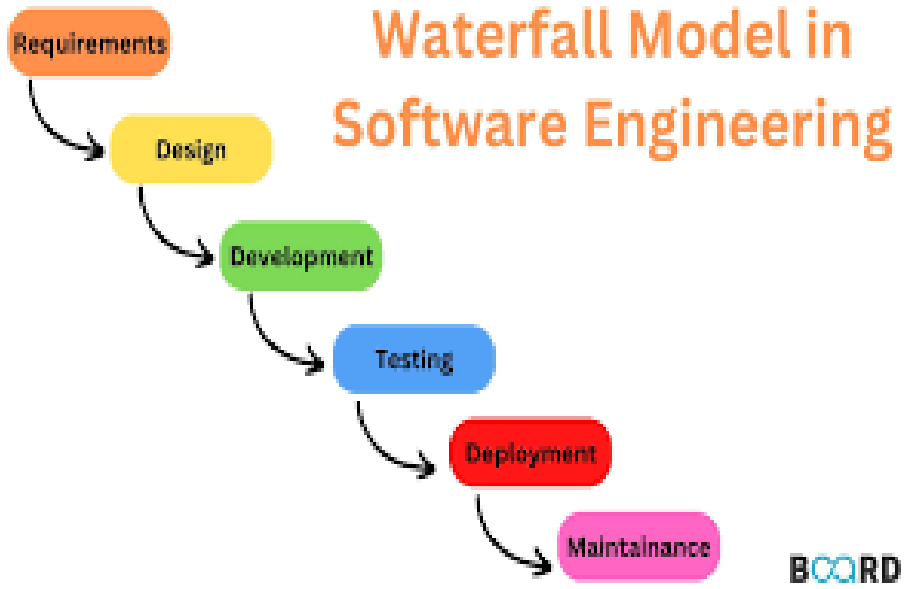
Software Process–

- In software engineering, a software development process is the process of dividing software development work into distinct phases to improve design, product management, and project management. It is also known as a software development life cycle.

Water Fall Model–

- Waterfall approach was first SDLC Model to be used widely in Software Engineering to ensure success of the project. In "The Waterfall" approach, the whole process of software development is divided into separate phases. In this Waterfall model, typically, the outcome of one phase acts as the input for the next phase sequentially.

The following illustration is a representation of the different phases of the Waterfall Model.



The sequential phases in Waterfall model are −

Requirement Gathering and analysis − All possible requirements of the system to be developed are captured in this phase and documented in a requirement specification document.

System Design − The requirement specifications from first phase are studied in this phase and the system design is prepared. This system design helps in specifying hardware and system requirements and helps in defining the overall system architecture.

Implementation − With inputs from the system design, the system is first developed in small programs called units, which are integrated in the next phase. Each unit is developed and tested for its functionality, which is referred to as Unit Testing.

Integration and Testing − All the units developed in the implementation phase are integrated into a system after testing of each unit. Post integration the entire system is tested for any faults and failures.

Deployment of system − Once the functional and non-functional testing is done; the product is deployed in the customer environment or released into the market.

Maintenance − There are some issues which come up in the client environment. To fix those issues, patches are released. Also to enhance the product some better versions are released. Maintenance is done to deliver these changes in the customer environment.

Increment Model–

- In the Iterative Increment model, iterative process starts with a simple implementation of a small set of the software requirements and iteratively enhances the evolving versions until the complete system is implemented and ready to be deployed.
- Iterative process starts with a simple implementation of a subset of the software requirements and iteratively enhances the evolving versions until the full system is implemented.

- The following illustration is a representation of the Iterative and Incremental model −

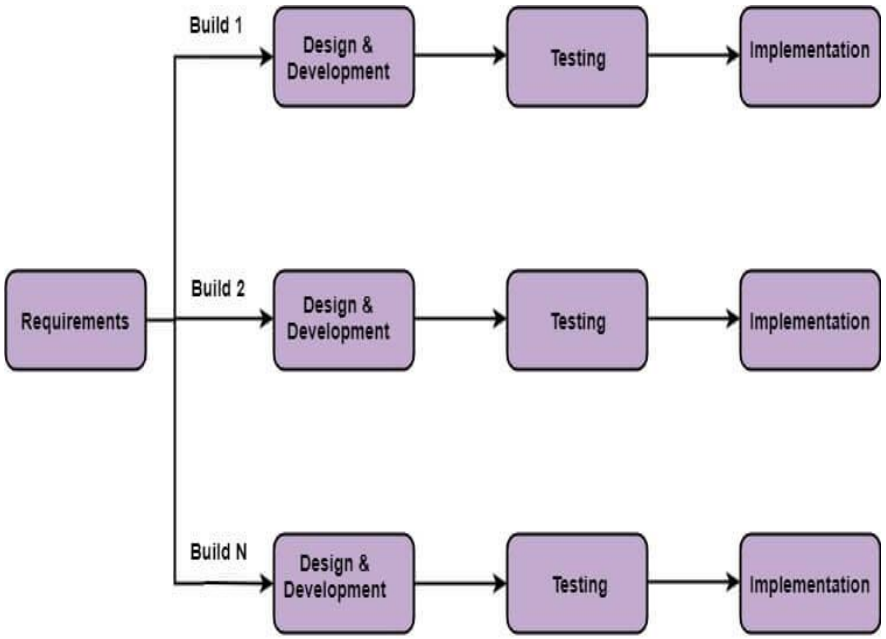


Fig: Incremental Model

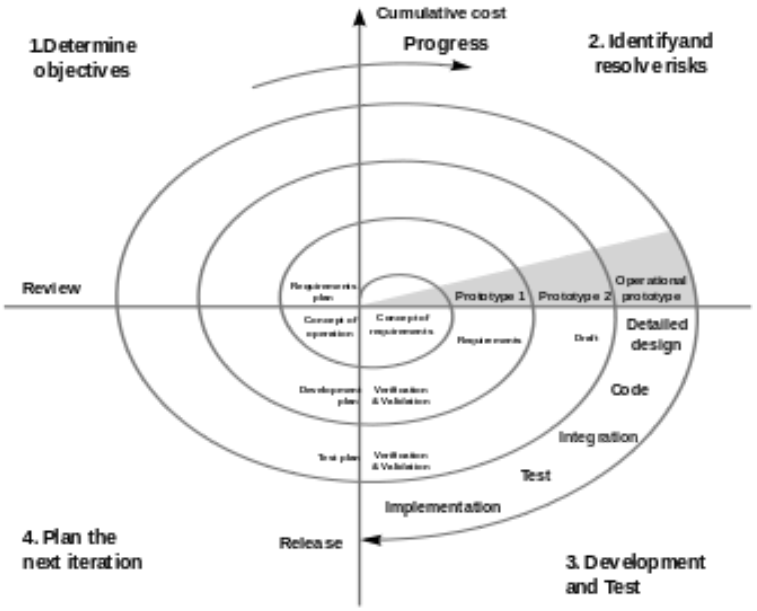
- Iterative and Incremental development is a combination of both iterative design or iterative method and incremental build model for development. "During software development, more than one iteration of the software development cycle may be in progress at the same time." This process may be described as an "evolutionary acquisition" or "incremental build" approach."
- In this incremental model, the whole requirement is divided into various builds. During each iteration, the development module goes through the requirements, design, implementation and testing phases. Each subsequent release of the module adds function to the previous release. The process continues till the complete system is ready as per the requirement.

Prototyping Spiral Model–

- The spiral model combines the idea of iterative development with the systematic, controlled aspects of the waterfall model. This Spiral model is a combination of iterative development process model and sequential linear development model i.e. the waterfall model with a very high emphasis on risk analysis. It allows incremental releases of the product or incremental refinement through each

Spiral Model - Design

- The spiral model has four phases. A software project repeatedly passes through these phases in iterations called Spirals.



Identification-

- This phase starts with gathering the business requirements in the baseline spiral. In the subsequent spirals as the product matures, identification of system requirements, subsystem requirements and unit requirements are all done in this phase.
- This phase also includes understanding the system requirements by continuous communication between the customer and the system analyst. At the end of the spiral, the product is deployed in the identified market.

Design-

- The Design phase starts with the conceptual design in the baseline spiral and involves architectural design, logical design of modules, physical product design and the final design in the subsequent spirals.

Construct or Build-

- The Construct phase refers to production of the actual software product at every spiral. In the baseline spiral, when the product is just thought of and the design is being developed a POC (Proof of Concept) is developed in this phase to get customer feedback.
- Then in the subsequent spirals with higher clarity on requirements and design details a working model of the software called build is produced with a version number. These builds are sent to the customer for feedback.

Evaluation and Risk Analysis-

- Risk Analysis includes identifying, estimating and monitoring the technical feasibility and management risks, such as schedule slippage and cost overrun. After testing the build, at the end of first iteration, the customer evaluates the software and provides feedback.
- The following illustration is a representation of the Spiral Model, listing the activities in each phase.

Role of management in software development–

A project is well-defined task, which is a collection of several operations done in order to achieve a goal (for example, software development and delivery). Role of management in software can be characterized as:

- Every project may have a unique and distinct goal.
- Project is not routine activity or day-to-day operations.
- Project comes with a start time and end time.
- Project ends when its goal is achieved hence it is a temporary phase in the lifetime of an organization.
- Project needs adequate resources in terms of time, manpower, finance, material and knowledge-bank.

Role of matrices and measurements–

- In software projects, it is most important to measure the quality, cost and effectiveness of the project and the processes. Without measuring these, a project can't be completed successfully.

Metrics

- A Metric is a quantitative measure of the degree to which a system, system component, or process possesses a given attribute.
- Metrics can be defined as "STANDARDS OF MEASUREMENT". Software Metrics are used to measure the quality of the project. Simply, Metric is a unit used for describing an attribute. Metric is a scale for measurement.

Test metrics example:

- How many defects exist within the module?
- How many test cases are executed per person?
- What is the Test coverage %?

Measurement -

- Measurement is the quantitative indication of extent, amount, dimension, capacity, or size of some attribute of a product or process.
- Test measurement example: Total number of defects.

Problem Analysis–

- Problem analysis is the process of understanding real-world problems and user's needs and proposing solutions to meet those needs.
    - The goal of problem analysis is to gain a better understanding, before development begins, of the problem being solved.
- To identify the root cause, or the problem behind the problem, ask the people directly involved.
- Identifying the actors on the system is a key step in problem analysis.

Requirement Specification –

- A software requirements specification (SRS) is a document that captures complete description about how the system is expected to perform. It is usually signed off at the end of requirements engineering phase.

Qualities of SRS:

- Correct
- Unambiguous
- Complete
- Consistent
- Ranked for importance and/or stability
- Verifiable
- Modifiable
- Traceable

Types of Requirements:

The below diagram depicts the various types of requirements that are captured during SRS.

Monitoring and Control–

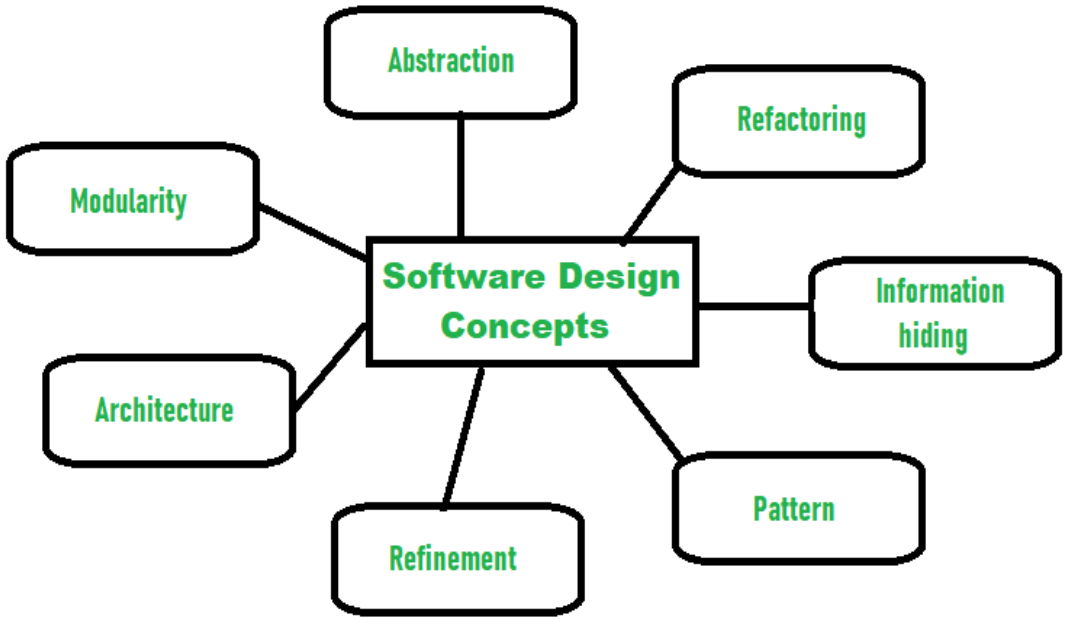
- The Monitoring and Controlling process oversees all the tasks and metrics necessary to ensure that the approved and authorized project is within scope, on time, and on budget so that the project proceeds with minimal risk. ... Monitoring and Controlling process is continuously performed throughout the life of the project.

Unit – 3

Software Design

Design Principles–

- Once the requirements document for the software to be developed is available, the software design phase begins. While the requirement specification activity deals entirely with the problem domain, design is the first phase of transforming the problem into a solution. In the design phase, the customer and business requirements and technical considerations all come together to formulate a product or a system.
- Developing design is a cumbersome process as most expansive errors are often introduced in this phase. Moreover, if these errors get unnoticed till later phases, it becomes more difficult to correct them. Therefore, a number of principles are followed while designing the software. These principles act as a framework for the designers to follow a good design practice.



Some of the commonly followed design principles are as following

1. **Software design should correspond to the analysis model:** Often a design element corresponds to many requirements, therefore, we must know how the design model satisfies all the requirements represented by the analysis model.
2. **Choose the right programming paradigm:** A programming paradigm describes the structure of the software system. Depending on the nature and type of application, different programming paradigms such as procedure oriented, object-oriented, and prototyping paradigms can be used. The paradigm should be chosen keeping constraints in mind such as time, availability of resources and nature of user's requirements.
3. **Software design should be uniform and integrated:** Software design is considered uniform and integrated, if the interfaces are properly defined among the design components. For this, rules, format, and styles are established before the design team starts designing the software.
4. **Software design should be flexible:** Software design should be flexible enough to adapt changes easily. To achieve the flexibility, the basic design concepts such as abstraction, refinement, and modularity should be applied effectively.
5. **Software design should ensure minimal conceptual (semantic) errors:** The design team must ensure that major conceptual errors of design such as ambiguousness and inconsistency are addressed in advance before dealing with the syntactical errors present in the design model.
6. **Software design should be structured to degrade gently:** Software should be designed to handle unusual changes and circumstances, and if the need arises for termination, it must do so in a proper manner so that functionality of the software is not affected.
7. **Software design should represent correspondence between the software and real-world problem:** The software design should be structured in such away that it always relates with the real-world problem.
8. **Software reuse: Software engineers believe on the phrase:** 'do not reinvent the wheel'. Therefore, software components should be designed in such a way that they can be effectively reused to increase the productivity.
9. **Designing for testability:** A common practice that has been followed is to keep the testing phase separate from the design and implementation phases. That is, first the software is developed (designed and

implemented) and then handed over to the testers who subsequently determine whether the software is fit for distribution and subsequent use by the customer. However, it has become apparent that the process of separating testing is seriously flawed, as if any type

of design or implementation errors are found after implementation, then the entire or a substantial part of the software requires to be redone. Thus, the test engineers should be involved from the initial stages. For example, they should be involved with analysts to prepare tests for determining whether the user requirements are being met.

10. **Prototyping:** Prototyping should be used when the requirements are not completely defined in the beginning. The user interacts with the developer to expand and refine the requirements as the development proceeds. Using prototyping, a quick 'mock-up' of the system can be developed. This mock-up can be used as a effective means to give the users a feel of what the system will look like and demonstrate functions that will be included in the developed system. Prototyping also helps in reducing risks of designing software that is not in accordance with the customer's requirements.

Problem Partitioning–

- When solving a small problem, the entire problem can be tackled at once. The complexity of large problems and the limitations of human minds do not allow large problems to be treated as huge monoliths.
- For solving larger problems, the basic principle is the time-tested principle of “divide and conquer.” Clearly, dividing in such a manner that all the divisions have to be conquered together is not the intent of this wisdom. This principle, if elaborated, would mean, “Divide into smaller pieces, so that each piece can be conquered separately.”
- For software design, therefore, the goal is to divide the problem into manageably small pieces that can be solved separately. It is this restriction of being able to solve each part separately that makes dividing into pieces a complex task and that many methodologies for system design aim to address. The basic rationale behind this strategy is the belief that if the pieces of a problem are solvable separately, the cost of solving the entire problem is more than the sum of the cost of solving all the pieces.
- However, the different pieces cannot be entirely independent of each other, as they together form the system. The different pieces have to cooperate and communicate to solve the larger problem. This communication adds complexity, which arises due to partitioning and

may not have existed in the original problem. As the number of components increases, the cost of partitioning, together with the cost of this added complexity, may become more than the savings achieved by partitioning. It

is at this point that no further partitioning needs to be done. The designer has to make the judgment about when to stop partitioning.

Abstraction–

- Abstraction is the intellectual tool that allows us to deal with concepts apart from particular instances of those concepts. During requirements definition and design, abstraction permits separation of the conceptual aspects of a system from the (yet to be specified) implementation details. We can for example, specify the FIFO property of a queue or the LIFO property of a stock without concern for the representation scheme to be used in implementing the stack or queue. Similarly, we can specify the functional characteristics of the routines that manipulate data structures (e.g. NEW, PUSH, POP, TOP, EMPTY) without concern for the algorithmic details of the routines.
- Abstraction is a very powerful concept that is used in all-engineering disciplines. It is a tool that permits a designer to consider a component at an abstract level without worrying about the details of the implementation of the component. Any component or system provides some services to its environment. An abstraction of a component describes the external behavior of that component without bothering with the internal details that produce the behavior. Presumably, the abstract definition of a component is much simpler than the component itself.
- Abstraction is an indispensable part of the design process and is essential for problem partitioning. Partitioning, essentially, is the exercise of determining the components of a system. However, these components are not isolated from each other; they interact with each other, and the designer has to specify how a component interacts with other components. To decide how a component interacts with other components, the designer has to know, at the very least, the external behavior of other components. If the designer has to understand the details of the other components to determine their external behavior, we have defeated the purpose of partitioning-isolating a component from others. To allow the designer to concentrate on one component at a time, abstraction of other components is used.

- Abstraction is used for existing components as well as components that are being designed. Abstraction of existing components plays an important role in the maintenance phase. To modify a system, the first step is

  understanding what the system does and how. The process of comprehending an existing system involves identifying the abstractions of sub-systems and components from the details of their implementations. Using these abstractions, the behavior of the entire system can be understood. This also helps determine how modifying a component affects the system.

Top Down & Button Up Design–

Top-down Design

- We know that a system is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their onset of sub-system and components and creates hierarchical structure in the system.
- Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.
- Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.
- Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown.

Bottom-up Design

- The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.
- Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.
- Both, top-down and bottom-up approaches are not practical

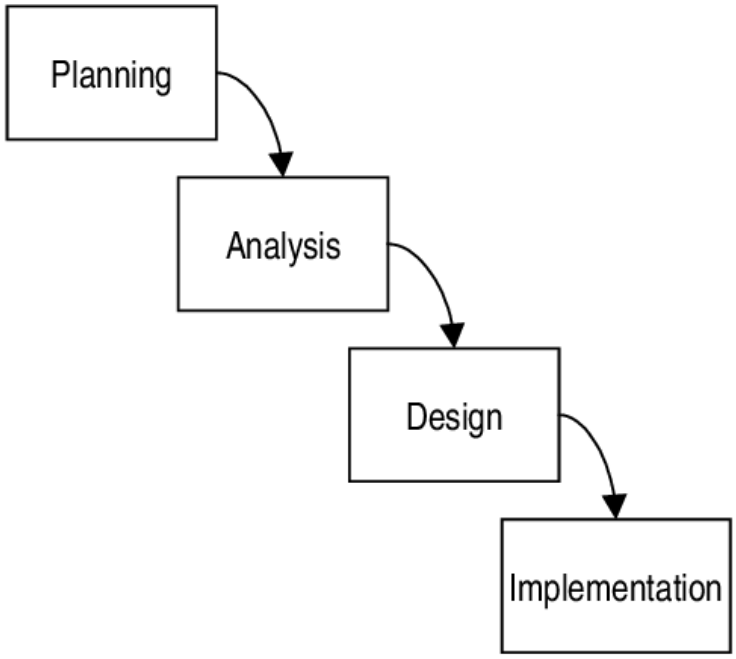
Structured Approach–

The structured design approach helps developers deal with the size and complexity of large scale projects. The structured approach is a process oriented approach, aiming to break a large complex project into a series of smaller, more manageable modules.

The Structured approach is usually associated with projects that have the following characteristics:

- **Long time periods:** The time involved in a structured approach is generally measured in months, or even years.
- **Large-scale projects:** The structured approach is usually applied to large projects which involve a huge amount of detail and coordination between various parts of the business and other systems.

Large budgets: Large projects will involve a large number of people and will employ them over a long time scale. The budget, consequently, will be large. New hardware and software is often required for completion of the project, elevating the cost of the project - though the amount spent upon personnel is usually the highest cost factor.
The steps of the structured approach are as follows:



Defining the problem: In this stage, the requirements for the software are gathered and analyzed, to produce a complete and unambiguous specification of what the software is required to do. The problem will need careful analysis so that the "true" issues involved are identified. For example, if your car won't start, your "problem" is "My car is not working". However, this is merely an analysis of the "symptoms". Deeper analysis may reveal that there is no petrol in the car, or the battery is flat or some other problem. Until the true cause of the problem is discovered, a useable solution cannot be developed. This stage is sometimes called Requirements Analysis.

Planning: During this phase, software architecture for the implementation of the requirements is designed and specified, identifying the components within the software and the relationships between the components. Inputs, Outputs and Processes required for the new system are identified and a basic plan in the form of flowcharts or pseudo code is developed. A "Top Down" approach would almost certainly be used. This involves breaking a larger and complex problem into smaller more manageable modules - each module being coded separately. Thus, a structure chart is an important document developed during this phase. This stage is sometimes called Architectural and Detailed design.

Building: This is the Code and Test phase, in which each component of the software is coded and tested to verify that it faithfully implements the detailed design outlined in the previous phase. Finally, once all of the modules have been coded and tested the software is integrated. Progressively larger groups of tested software components are integrated and tested until the software works as a whole. This process is sometimes called Software Integration.

Checking the solution: This involves Acceptance Testing, where tests are applied and witnessed to validate that the software faithfully implements the specified requirements. Real data may be used and this is compared to outputs from a previous system. The system may also be tested under full load. A program may work well for a small data set, but have problems when thousands or millions of records and transactions are required of it. Beta testing may be used during application software development. Users are also involved in this process to ensure that their

expectations of the system are fulfilled. In addition to the feedback that the development team receives, users also benefit by being able to use the system. In addition to checking that the program can process the data correctly, the user interface is evaluated for ease of use and intuitive design.

Modifying the solution: During the testing phase, problems or possible improvements (from beta testing) may be found. If problems are found, modifications may be made to solve the problems and another round of testing will occur.

Functional v/s Object Oriented Approach–

FOD: The basic abstractions, which are given to the user, are real world functions. OOD: The basic abstractions are not the real world functions but are the data abstraction where the real world entities are represented.

FOD: Functions are grouped together by which a higher level function is Page on obtained. an eg of this technique is SA/SD.
OOD: Functions are grouped together on the basis of the data they operate since the classes are associated with their methods.

FOD: In this approach the state information is often represented in a centralized shared memory.
OOD: In this approach the state information is not represented in a centralized memory but is implemented or distributed among the objects of the system.

FOD approach is mainly used for computation sensitive application,
OOD: whereas OOD approach is mainly used for evolving system which mimics a business process or business case.

In FOD - we decompose in function/procedure level
OOD: - we decompose in class level

FOD: Top down Approach

OOD: Bottom up approach

FOD: It views system as Black Box that performs high level function and later decomposes it detailed function so to be mapped to modules.
OOD: Object-oriented design is the discipline of defining the objects and their interactions to solve a problem that was identified and documented during object-oriented analysis.

FOD: Begins by considering the use case diagrams and Scenarios. OOD: Begins by identifying objects and classes

Design Specification and Verification–

Design Specification

- A software design description (a.k.a. software design document or SDD), also Software Design Specification is a written description of a software product, that a software designer writes in order to give a software development team overall guidance to the architecture of the software project.

Design Verification

- Verification Testing can be defined as a method of confirmation by examining and providing evidence that the design output meets the design input specifications. An essential process during any product development that ensures the designed product is same as the intended use.
- Design input is any physical and performance requirement that is used as the basis for designing purpose. Design output is the result of each design phase and at the end of total design effort. The final design output is a basis for device master record.

Monitoring and Control–

- The objective of project monitoring and control is to review the project performance and progress, risks, issues, stakeholder's commitments and involvement against project plans. Monitoring and status review at different levels of management on different stages of the project

better management control and early detection of project issues and caring out appropriate corrective action to address those issues.

Cohesiveness–

- The literary meanings of word "cohesion" are consistency and organization of different units. In computer science and software engineering, cohesion refers to the level of strength and unity with which different components of a software program are inter-related with each other.

There are seven types of cohesion, namely –

Co-incidental cohesion - It is unplanned and random cohesion, which might be the result of breaking the program into smaller modules for the sake of modularization. Because it is unplanned, it may serve confusion to the programmers and is generally not-accepted.

Logical cohesion - When logically categorized elements are put together into a module, it is called logical cohesion.

Temporal Cohesion - When elements of module are organized such that they are processed at a similar point in time, it is called temporal cohesion.

Procedural cohesion - When elements of module are grouped together, which are executed sequentially in order to perform a task, it is called procedural cohesion.

Communicational cohesion - When elements of module are grouped together, which are executed sequentially and work on same data (information), it is called communicational cohesion.

Sequential cohesion - When elements of module are grouped because the output of one element serves as input to another and so on, it is called sequential cohesion.

Functional cohesion - It is considered to be the highest degree of cohesion, and it is highly expected. Elements of module in functional cohesion

are grouped because they all contribute to a single well-defined function. It can also be reused.

Coupling–

- Coupling is a measure that defines the level of inter-dependability among modules of a program. It tells at what level the modules interfere and interact with each other. The lower the coupling, the better the program.

There are five levels of coupling, namely -

Content coupling - When a module can directly access or modify or refer to the content of another module, it is called content level coupling.

Common coupling- When multiple modules have read and write access to some global data, it is called common or global coupling.

Control coupling- Two modules are called control-coupled if one of them decides the function of the other module or changes its flow of execution.

Stamp coupling- When multiple modules share common data structure and work on different part of it, it is called stamp coupling.

Data coupling- Data coupling is when two modules interact with each other by means of passing data (as parameter). If a module passes data structure as parameter, then the receiving module should use all its components.

Ideally, no coupling is considered to be the best.

Forth Generation Techniques–

- The term fourth generation technique (4GT) encompasses a broad array of software tools that have one thing in common: each enables the software engineer to specify some characteristic of software at a high level.
- The tool then automatically generates source code based on the developer's specification. There is little debate that the higher the

level at which software can be specified to a machine, the faster a program can be built.

- 4GT paradigm for software engineering focuses on the ability to specify software using specialized language forms or a graphic notation that describes the problem to be solved in terms that the customer can understand

4GT TOOLS

• Currently, a software development environment that supports the 4GT paradigm includes some or all of the following tools:

Nonprocedural languages for database query, report generation, data manipulation, screen interaction and definition, Code generation; high-level graphics capability; spreadsheet capability, and automated generation of HTML and similar languages used for Web-site creation using advanced software tools.

Requirements gathering

• Like other paradigms, 4GT begins with a requirements gathering step. • Ideally, the customer would describe requirements and these would be directly translated into an operational prototype. But this is unworkable. • The customer may be unsure of what is required, may be ambiguous in specifying facts that are known, and may be unable or unwilling to specify information in a manner that a 4GT tool can consume.

Design

• For small applications, it may be possible to move directly from the requirements gathering step to implementation using a nonprocedural fourth generation language(4GL) or a model composed of a network of graphical icons. • However, for larger efforts, it is necessary to develop a design strategy for the system, even if a 4GL is to be used.

Implementation

• Implementation using a 4GL enables the software developer to represent desired results in a manner that leads to automatic generation of code to create those results. • Obviously, a data

structure with relevant information must exist and be readily accessible by the 4GL.

• To transform a 4GT implementation into a product, the developer must conduct thorough testing, develop meaningful documentation, and perform all other solution integration activities that are required in other software engineering paradigms. • In addition, the 4GT developed software must be built in a manner that enables maintenance to be performed expeditiously.

Advantages and disadvantages

• Proponents claim dramatic reduction in software development time and greatly improved productivity for people who build software. • Opponents claim that current 4GT tools are not all that much easier to use than programming languages. • the resultant source code produced by such tools is "inefficient," and that the maintainability of large software systems developed using 4GT is open to question

Current state of 4GT approaches

• The use of 4GT is a viable approach for many different application areas. Coupled with computer-aided software engineering tools and code generators, 4GT offers a credible solution to many software problems.

• Data collected from companies that use 4GT indicates that the time required producing software is greatly reduced for small and intermediate applications and that the amount of design and analysis for small applications is also reduced.

• However, the use of 4GT for large software development efforts demands as much or more analysis, design, and testing (software engineering activities) to achieve substantial time savings that result from the elimination of coding.

Functional Independence–

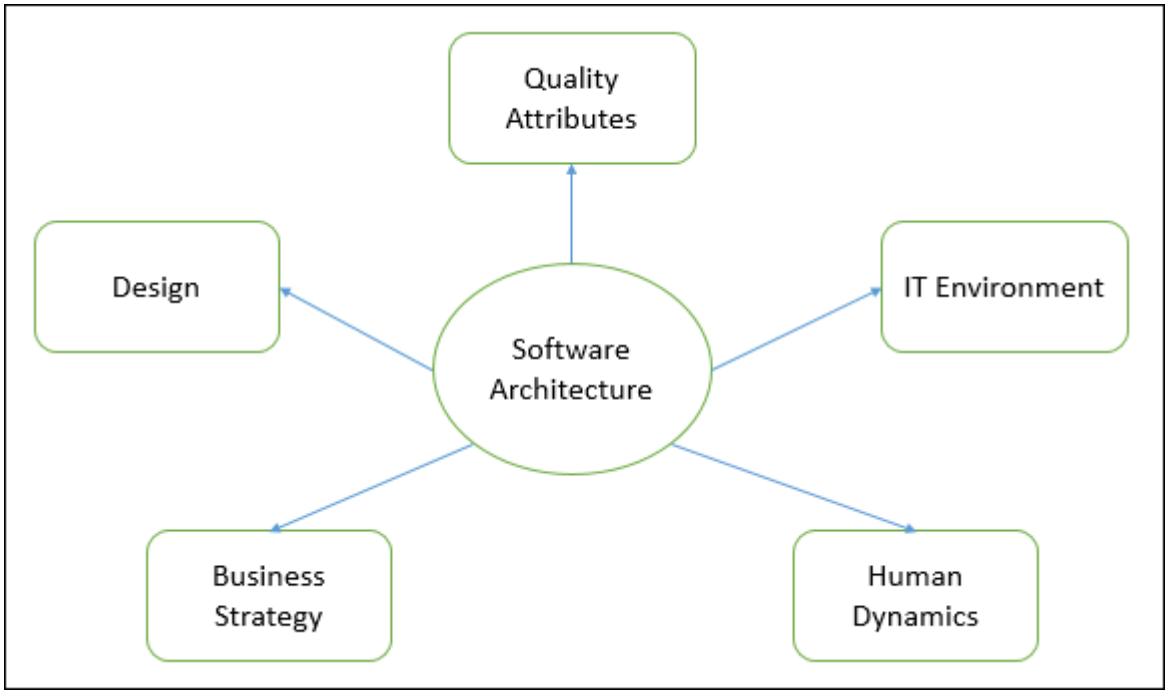
- The concept of functional Independence is a direct outgrowth of modularity and the concepts of abstraction and information hiding.

- Design software so that each module addresses a specific sub-function of requirements and has a simple interface when viewed from other parts of the program structure.

- Functional independence is a key to good design, and design is the key to software quality.

Software Architecture–

- The architecture of a system describes its major components, their relationships (structures), and how they interact with each other. Software architecture and design includes several contributory factors such as Business strategy, quality attributes, human dynamics, design, and IT environment.
  - We can segregate Software Architecture and Design into two distinct phases: Software Architecture and Software Design. In Architecture, nonfunctional decisions are cast and separated by the functional requirements. In Design, functional requirements are accomplished.



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

Unit – 4

Coding

Top Down & Bottom Up Programming–

- All designs contain hierarchies, as creating a hierarchy is a natural way to manage complexity. Most design methodologies for software also produce hierarchies. The hierarchy may be of functional modules, as is the case with the structured design methodology where the hierarchy of modules is represented by the structure chart. Or, the hierarchy may be an object hierarchy as is produced by object-oriented design methods and, frequently, represented by object diagrams. The question at coding time is: given the hierarchy of modules produced by design, in what order should the modules be built-starting from the top level or starting from the bottom level?
- In a top-down implementation, the implementation starts from the top of the hierarchy and proceeds to the lower levels. First, the main module is implemented, then its subordinates are implemented, and their subordinates, and so on. In a bottom-up implementation, the process is the reverse. The development starts with implementing the modules at the bottom of the hierarchy and proceeds through the higher levels until it reaches the top.
- Top-down and bottom-up implementation should not be confused with top-down and bottom-up design. Here, the design is being implemented, and if the design is fairly detailed and complete, its implementation can proceed in either the top-down or the bottom-up manner, even if the design was produced in a top-down manner. Which of the two is used, mostly affects testing.

Structured Programming–

- Structured coding practices translate a structured design into well structured code. PDL statements come in four different categories: sequence, selection (IF-THEN-ELSE, CASE), iteration (WHITE, REPEAT-UNTIL, FOR), and parallelism. Data statements included structure definitions and

monitor. Programming languages may have special purpose statements: patter matching in SNOBOL; process creation and generation of variates for some probability distributions in simulation languages such as SIMULA67, and creating appending, or querying a database file in DBase (Reg. Trademark). Even special purpose languages have at least the first three types of statements.

- The goal of the coding effort is to translate the design into a set of Single Entry-Single-Exit (SESE) modules. We can explain this by representing a program as a directed graph where every statement is a node and, possible transfers of control between statements is indicated through arcs between nodes. Such a control flow graph shows one input arc, one output arc and for all nodes in the graph a path starts at the input arc, goes to the output arc, and passes through that node.

Information Hiding–

- A software solution to a problem always contains data structures that are meant to represent information in the problem domain. That is, when software is developed to solve a problem, the software uses some data structures to capture the information in the problem domain. With the problem information represented internally as data structures, the required functionality of the problem domain, which is in terms of information in that domain, can be implemented as software operations on the data structures. Hence, any software solution to a problem contains data structures that represent information in the problem domain.
- If the information hiding principle is used, the data structure need not be directly used and manipulated by other modules. All modules, other than the access functions, access the data structure through the access functions.
- Information hiding can reduce the coupling between modules and make the system more maintainable.

Programming Style and internal documentation–

Programming Style

- Why is programming style important? A well written program is more easily read and understood both by the author and by others who work with that program. Not even the author will long remember his precise thoughts on a program. The program itself should help the reader to understand what it does quickly because only a small fraction of the developers if any, are maintaining the program they wrote. Others will, and they must be able to understand what the program does. Bad programming style makes program difficult to understand, hard to modify, and impossible to maintain over a long period of time, even by the person who coded it originally.

A good programming style is characterized by the following: simplicity

- readability
- good documentation
- changeability
- predictability
- consistency in input and output
- module independence
- Good structure.

Internal Documentation

- This is the phase, which provides help to the programmer for further review of the software and existing systems. In the coding phase, the output document is the code itself. However, some amount of internal documentation in the code can be extremely useful in enhancing the understandability of programs. Internal documentation of programs is done by the use of comments.
- All languages provide a means for writing comments in programs. Comments are textual statements that are meant for the program reader and are not executed. Comments, if properly written and kept consistent with the code, can be invaluable during maintenance.

Testing: Testing Principles–

Principles of Software Testing:

Testing Shows Presence of Defects:

- Testing shows the presence of defects in the software. The goal of testing is to make the software fail. Sufficient testing reduces the presence of defects. In case testers are unable to find defects after repeated regression testing doesn't mean that the software is bug-free.

Exhaustive Testing is Impossible:

- Testing all the functionalities using all valid and invalid inputs and preconditions is known as Exhaustive testing.

Early Testing:

- Defects detected in early phases of SDLC are less expensive to fix. So conducting early testing reduces the cost of fixing defects.

Defect Clustering:

- Defect Clustering in software testing means that a small module or functionality contains most of the bugs or it has the most operational failures.

Pesticide Paradox:

- Pesticide Paradox in software testing is the process of repeating the same test cases again and again, eventually, the same test cases will no longer find new bugs. So to overcome this Pesticide Paradox, it is necessary to review the test cases regularly and add or update them to find more defects.

Testing is Context Dependent:

- Testing approach depends on the context of the software we develop. We do test the software differently in different contexts. For example, online banking application requires a different approach of testing compared to an e-commerce site.

Absence of Error – Fallacy:

- 99% of bug-free software may still be unusable, if wrong requirements were incorporated into the software and the software is not addressing the business needs.

Levels of Testing–

Levels of testing include different methodologies that can be used while conducting software testing. The main levels of software testing are −

- Functional Testing
- Non-functional Testing

Functional Testing–

- This is a type of black-box testing that is based on the specifications of the software that is to be tested. The application is tested by providing input and then the results are examined that need to conform to the functionality it was intended for. Functional testing of a software is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

There are five steps that are involved while testing an application for functionality.

Steps Description

I The determination of the functionality that the intended application is meant to perform.

II	The creation of test data based on the specifications of the

III The output based on the test data and the specifications of the application.

IV The writing of test scenarios and the execution of test cases.

V	The comparison of actual and expected results based on the executed test cases.

An effective testing practice will see the above steps applied to the testing policies of every organization and hence it will make sure that the organization maintains the strictest of standards when it comes to software quality.

Structural Testing–

- Structural or White-box testing is the detailed investigation of internal logic and structure of the code. White-box testing is also called glass testing or open-box testing. In order to perform white-box testing on an application, a tester needs to know the internal workings of the code.
  - The tester needs to have a look inside the source code and find out which unit/chunk of the code is behaving inappropriately.

The following table lists the advantages and disadvantages of white-box testing.

Advantages	Disadvantages
As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively.	Due to the fact that a skilled tester is needed to perform white-box testing, the costs are increased.
It helps in optimizing the code.	Sometimes it is impossible to look into every nook and corner to find out hidden errors that may create problems, as many paths will go untested.
Extra lines of code can be removed which can bring in hidden defects.	Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing.

Test Plane–

- A test plan outlines the strategy that will be used to test an application, the resources that will be used, the test environment in which testing will be performed, and the limitations of the testing and the schedule of testing activities. Typically the Quality Assurance Team Lead will be responsible for writing a Test Plan.

A test plan includes the following −

- Introduction to the Test Plan document
- Assumptions while testing the application
- List of test cases included in testing the application
- List of features to be tested
- What sort of approach to use while testing the software
- List of deliverables that need to be tested
- The resources allocated for testing the application
- Any risks involved during the testing process
- A schedule of tasks and milestones to be achieved

Test case Specification–

- Test cases involve a set of steps, conditions, and inputs that can be used while performing testing tasks. The main intent of this activity is to ensure whether a software passes or fails in terms of its functionality and other aspects. There are many types of test cases such as functional, negative, error, logical test cases, physical test cases, UI test cases, etc.

- Furthermore, test cases are written to keep track of the testing coverage of a software. Generally, there are no formal templates that can be used during test case writing. However, the following components are always available and included in every test case −
- Test case ID
- Product module
- Product version
- Revision history
- Purpose

- Assumptions
- Pre-conditions
- Steps
- Expected outcome
- Actual outcome
- Post-conditions

Many test cases can be derived from a single test scenario. In addition, sometimes multiple test cases are written for a single software which are collectively known as test suites.

Reliability assessment–

- Software Reliability is the probability of failure-free software operation for a specified period of time in a specified environment. Software Reliability is also an important factor affecting system reliability. ... Measurement in software is still in its infancy.
- Software Reliability Assessment by Statistical Analysis of Operational Experience. Statistical testing represents a well-founded approach to the estimation of software reliability. ... The execution of a test case does not influence the outcome of further test cases.

Software testing strategies–

A healthy software testing or QA strategy requires tests at all technology stack levels to ensure that every part, as well as the entire system, works correctly.

Leave time for fixing: Setting aside time for testing is pointless if there is no time set aside for fixing. Once problems are discovered, developers required time to fix them and the company needs time to retest the fixes as well. With a time and plan for both, then testing is not very useful.

Discourage passing the buck: The same way that testers could fall short in their reports, developers could also fall short in their effort to comprehend the reports. One way of minimizing back and forth conversations between developers and testers are having a culture that will encourage them to hop on the phone or have desk-side chat to get to the bottom of things. Testing and fixing are all about collaboration. Although it is important that developers should not waste time on a wild goose chase, it is equally important that bugs are not just shuffled back and forth.

Manual testing has to be exploratory: A lot of teams prefer to script manual testing so testers follow a set of steps and work their way through a set of tasks that are predefined for software testing. This misses the point of manual testing. If something could be written down or scripted in exact terms, it could be automated and belongs in the automated test suite. Real-world use of the software will not be scripted, thus testers must be free to probe and break things without a script.

Encourage clarity: Reporting bugs and asking for more information could create unnecessary overhead costs. A good bug report could save time through avoiding miscommunication or a need for more communication. In the same way, a bad bug report could lead to a fast dismissal by a developer. These could create problems. Anyone reporting bugs should make it a point to create bug reports that are informative. However, it is also integral for a developer to out of the way to effectively communicate as well.

Test often: The same as all other forms of testing, manual testing will work best when it occurs often throughout the development project, in general, weekly or bi-weekly. This helps in preventing huge backlogs of problems from building up and crushing morale. Frequent testing is considered the best approach.

- Testing and fixing software could be tricky, subtle and political even. Nevertheless, as long as one is able to anticipate and recognize common issues, things could be kept running smoothly.

Verification and Validation–

- These two terms are very confusing for most people, who use them interchangeably. The following table highlights the differences between verification and validation.

Sr.No. Verification Validation

Verification	Validation
It includes checking documents, design, codes and programs.	It includes testing and validating the actual product.
Verification is the static testing.	Validation is the dynamic testing.
It does *not* include the execution of the code.	It includes the execution of the code.
Methods used in verification are reviews, walkthroughs, inspections and desk-checking.	Methods used in validation are Black Box Testing, White Box Testing and non-functional testing.
It checks whether the software conforms to specifications or not.	It checks whether the software meets the requirements and expectations of a customer or not.
It can find the bugs in the early stage	It can only find the bugs that could not

Unit Testing–

- This type of testing is performed by developers before the setup is handed over to the testing team to formally execute the test cases. Unit testing is performed by the respective developers on the individual units of source code assigned areas. The developers use test data that is different from the test data of the quality assurance team.
- The goal of unit testing is to isolate each part of the program and show that individual parts are correct in terms of requirements and functionality.

Limitations of Unit Testing

- Testing cannot catch each and every bug in an application. It is impossible to evaluate every execution path in every software application. The same is the case with unit testing.
- There is a limit to the number of scenarios and test data that a developer can use to verify a source code. After having exhausted all the options, there is no choice but to stop unit testing and merge the code segment with other units.

Integration Testing–

- Integration testing is defined as the testing of combined parts of an application to determine if they function correctly. Integration testing can be done in two ways: Bottom-up integration testing and Top-down integration testing.

Sr.No.	Integration Testing Method
1	Bottom-up integration This testing begins with unit testing, followed by tests of progressively higher-level combinations of units called modules or builds.
2	Top-down integration In this testing, the highest-level modules are tested first and progressively, lower-level modules are tested thereafter.

In a comprehensive software development environment, bottom-up testing is usually done first, followed by top-down testing. The process concludes with multiple tests of the complete application, preferably in scenarios designed to mimic actual situations.

Alpha and Beta Testing–

Alpha Testing

This test is the first stage of testing and will be performed amongst the teams (developer and QA teams). Unit testing, integration testing and system testing when combined together is known as alpha testing. During this phase, the following aspects will be tested in the application −

- Spelling Mistakes
- Broken Links
- Directions
- The Application will be tested on machines with the lowest specification to test loading times and any latency problems.

Beta Testing

This test is performed after alpha testing has been successfully performed. In beta testing, a sample of the intended audience tests the application. Beta testing is also known as pre-release testing. Beta test versions of software are ideally distributed to a wide audience on the Web, partly to give the program a "real-world" test and partly to provide a preview of the next release. In this phase, the audience will be testing the following −

- Users will install, run the application and send their feedback to the project team.
- Typographical errors, confusing application flow, and even crashes.

- Getting the feedback, the project team can fix the problems before releasing the software to the actual users.
- The more issues you fix that solve real user problems, the higher the quality of your application will be.
- Having a higher-quality application when you release it to the general public will increase customer satisfaction.

System testing and debugging–

Testing − It involves idenƟfying bug/error/defect in a soŌware without correcƟng it. Normally professionals with a quality assurance background are involved in bug's identification. Testing is performed in the testing phase.

Debugging − It involves idenƟfying, isolaƟng, and fixing the problems/bugs. Developers who code the software conduct debugging upon encountering an error in the code. Debugging is a part of White Box Testing or Unit Testing. Debugging can be performed in the development phase while conducting Unit Testing or in phases while fixing the reported bugs.

Unit – 5

Software Project Management

- Software is said to be an intangible product. Software development is a kind of all new stream in world business and there's very little experience in building software products. Most software products are tailor made to fit client's requirements. The most important is that the underlying technology changes and advances so frequently and rapidly that experience of one product may not be applied to the other one.

The Management Spectrum:

The People–

- Act as project leader
- Liaison with stakeholders
- Managing human resources
- Setting up reporting hierarchy etc.

Software Project Manager

A software project manager is a person who undertakes the responsibility of executing the software project. Software project manager is thoroughly aware of all the phases of SDLC that the software would go through. Project manager may never directly involve in producing the end product but he controls and manages the activities involved in production.

The Process -

- Every project may has a unique and distinct goal.
- Project is not routine activity or day-to-day operations.
- Project comes with a start time and end time.
- Project ends when its goal is achieved hence it is a temporary phase in the lifetime of an organization.
- Project needs adequate resources in terms of time, manpower, finance, material and knowledge-bank.

The Project-

- A Software Project is the complete procedure of software development from requirement gathering to testing and maintenance, carried out according to the execution methodologies, in a specified period of time to achieve intended software product.

Cost estimation –

- This might be considered as the most difficult of all because it depends on more elements than any of the previous ones. For estimating project cost, it is required to consider -
- Size of software
    - Software quality
    - Hardware
    - Additional software or tools, licenses etc.
    - Skilled personnel with task-specific skills
    - Travel involved
    - Communication
    - Training and support

Project scheduling-

- Project Scheduling in a project refers to roadmap of all activities to be done with specified order and within time slot allotted to each activity. Project managers tend to define various tasks, and project milestones and they arrange them keeping various factors in mind.
- They look for tasks lie in critical path in the schedule, which are necessary to complete in specific manner (because of task interdependency) and strictly within the time allocated. Arrangement of tasks which lies out of critical path are less likely to impact over all schedule of the project.

For scheduling a project, it is necessary to -

- Break down the project tasks into smaller, manageable form
- Find out various tasks and correlate them

- Estimate time frame required for each task
- Divide time into work-units

Assign adequate number of work-units for each task

- Calculate total time required for the project from start to finish

Software configuration management-

- Configuration management is a process of tracking and controlling the changes in software in terms of the requirements, design, functions and development of the product.
- IEEE defines it as “the process of identifying and defining the items in the system, controlling the change of these items throughout their life cycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items”.
- Generally, once the SRS is finalized there is less chance of requirement of changes from user. If they occur, the changes are addressed only with prior approval of higher management, as there is a possibility of cost and time overrun.

Structured v/s Unstructured maintenance-

- Unstructured maintenance wades straight into the source code and makes changes based on that alone

- Structured maintenance examines and modifies the original design, and then reworks the code to match it

- Clearly structured maintenance is a more reliable and (usually) a more efficient process

- Unfortunately, it's not always possible.