Software Engineering
Compiled by: Rajashekar Tuppada
Document Reference Code: i3Q-SDTC-SE-0119

1. PURPOSE

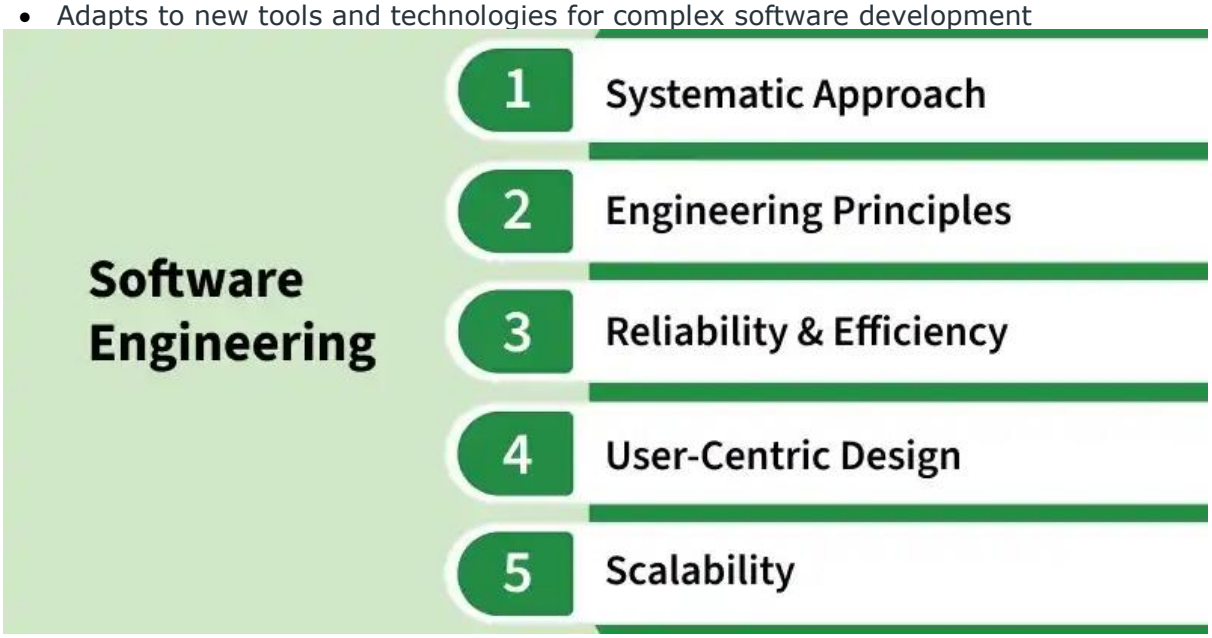
The Purpose of this document is to provide the Detailed Overview of

- Software Engineering
- Software Development Life Cycle (SDLC)
- Software Development Models
- Software Design and Quality
- Software Measurement and Metrics
- Testing Verification and Validation
- Practice Questions

2. SOFTWARE ENGINEERING

Software is a program or set of programs containing instructions that provide the desired functionality. Engineering is the process of designing and building something that serves a purpose efficiently and cost-effectively

- Uses requirements analysis, design, testing, and maintenance techniques
- Follows a well-defined software development lifecycle (SDLC)
- Improves development efficiency, cost management, and timely delivery
- Adapts to new tools and technologies for complex software development



2.1. Key Principles

- **Modularity**
  Divide software into small, independent parts that are easy to develop and test
- **Abstraction**
  Show only what is necessary and hide internal implementation details
- **Encapsulation**
  Keep data and related functions together and protect them from outside access
- **Maintenance**
  Regularly update software to fix bugs, improve features, and enhance security
- **Testing**
  Check the software to ensure it works correctly and meets requirements
- **Design Patterns**
  Use proven solutions for common software design problems

© i3Qube Software 2023 Confidential

- **Agile Methodologies**
  Develop software in small steps with frequent feedback and flexibility
- **Continuous Integration & Deployment (CI/CD)**
  Frequently merge code changes and automatically deploy updates

2.2. Dual Role of Software

Software plays a dual role in the industry by acting both as a product delivered to users and as a tool for building and running other systems

2.2.1. As a Product

- Delivers functionality to users
- Processes and manages information
- Runs on hardware systems

2.2.2. As a Vehicle for Delivering a Product

- Provides system functionality (e.g., payroll systems)
- Controls hardware/software (e.g., operating systems)
- Helps build other software (e.g., development tools)

2.3. Objectives

- **Maintainability**
  Software should be easy to update and improve when requirements change
- **Efficiency**
  Uses memory, CPU, and other resources without unnecessary waste
- **Reliability**
  Works consistently without failures over time
- **Correctness**
  Perform exactly as defined in the requirements
- **Reusability**
  Components can be used again in other projects
- **Testability**
  Easy to test and verify functionality
- **Portability**
  Runs smoothly across different systems or environments
- **Adaptability**
  Adjusts easily to new requirements or changes
- **Interoperability**
  Works well with other systems or software

2.4. Careers

A degree in software engineering and relevant experience can be utilized to explore several computing job choices. Software engineers have the opportunity to seek well-paying careers and professional progress, although their exact possibilities may vary depending on their particular school, industry and region

Following are the job choices in software engineering

- **Development Roles**
  Software Engineer, SDE, Front-End, Back-End
- **Testing Roles**
  QA Engineer, Software Test Engineer
- **Design Roles**
  Web Designer
- **Specialized Roles**
  Cloud Engineer, DevOps Engineer, Security Engineer

2.5. Tasks of Software Engineers

© i3Qube Software 2023 Confidential

The main responsibility of a software engineer is to develop useful computer programs and applications. Working in teams, you would complete various projects and develop solutions to satisfy certain customer or corporate demands. Some of the key responsibilities of software engineer are

- **Requirement Analysis**
  Collaborating with stakeholders to understand and gather the requirements to design and develop software solutions
- **Design and Development**
  Creating well-structured, maintainable code that meets the functional requirements and adheres to software design principles
- **Testing and Debugging**
  Writing and conducting unit tests, integration tests, and debugging code to ensure software is reliable and bug-free
- **Code Review**
  Participating in code reviews to improve code quality, ensure adherence to standards, and facilitate knowledge sharing among team members
- **Maintenance**
  Updating and maintaining existing software systems, fixing bugs, and improving performance or adding new features
- **Documentation**
  Writing clear documentation, including code comments, API documentation, and design documents to help other engineers and future developers understand the system

2.6. Skills Required

Following are some must have technical skills to become Software Engineers

- Coding and computer programming
- Programming (Java, Python, C++, etc.)
- Software testing
- Object-oriented design (OOD)
- Software development & Testing

Following are some must have soft skills to become Software Engineers

- Communication skills
- Problem-solving
- Teamwork
- Attention to detail

2.7. Program vs Software Product

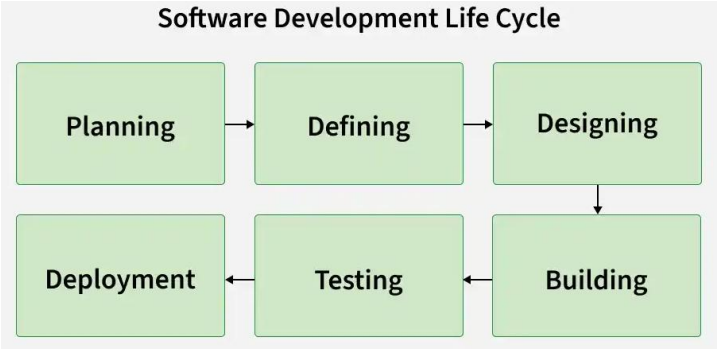
Program	Software Product
A set of instructions written to perform a specific task	A complete solution that includes the program along with documentation and licensing. Software Product = Program + Documentation + Licensing
Limited to a specific problem or task	Broader scope, designed to solve real-world problems and serve multiple users
Focuses mainly on coding	Follows a full development process including planning, design, development, testing, and maintenance
Little or no documentation	Proper and detailed documentation is provided

3. SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

The Software Development Life Cycle (SDLC) is a structured process used to plan, design, develop, test, deploy, and maintain software. It ensures a systematic workflow and helps align software development with business goals and user requirements

© i3Qube Software 2023 Confidential

- Provides a clear and organized framework for managing development phases
- Helps in early detection of defects, reducing overall cost and time
- Ensures high-quality software delivery that meets user expectations



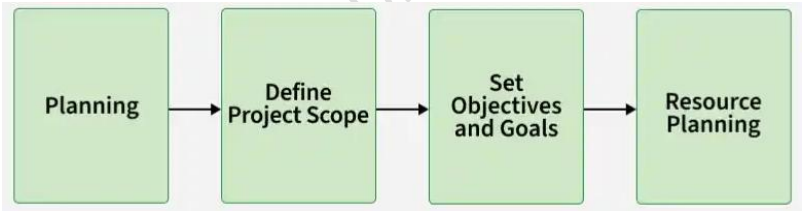
3.1. Stages of Software Development Life Cycle

The Software Development Life Cycle (SDLC) typically consists of six or seven stages, depending on the development model used

3.1.1. Stage 1: Planning & Feasibility Analysis

This stage determines whether the project is technically, financially, and operationally feasible

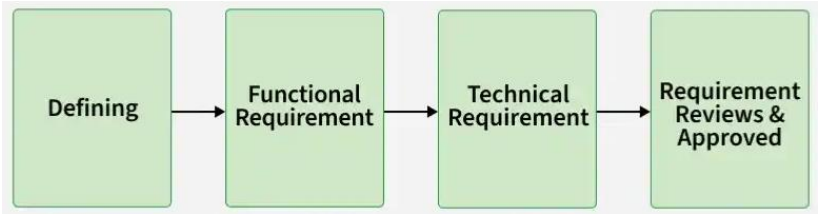
- **Activities**
  Feasibility analysis, cost estimation, scheduling, resource planning
- **Output**
  Project Plan, Feasibility Report
- **Key Roles**
  Project Managers, Senior Engineers, Stakeholders



3.1.2. Stage 2: Software Requirement Specification (SRS)

In this stage, detailed functional and non-functional requirements are documented clearly and approved by stakeholders

- **Activities**
  Requirement gathering, validation, documentation
- **Output**
  Software Requirement Specification (SRS)
- **Key Roles**
  Business Analysts, Product Owners

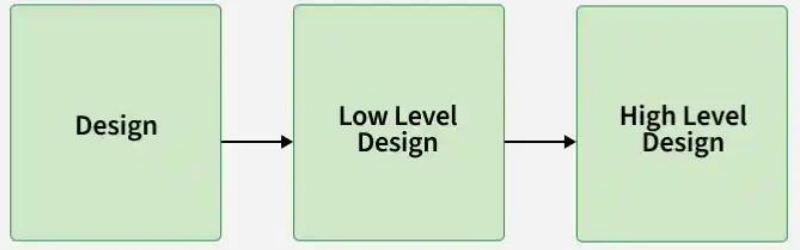


3.1.3. Stage 3: System Design

In this stage, the approved requirements are transformed into a technical blueprint for implementation

© i3Qube Software 2023 Confidential

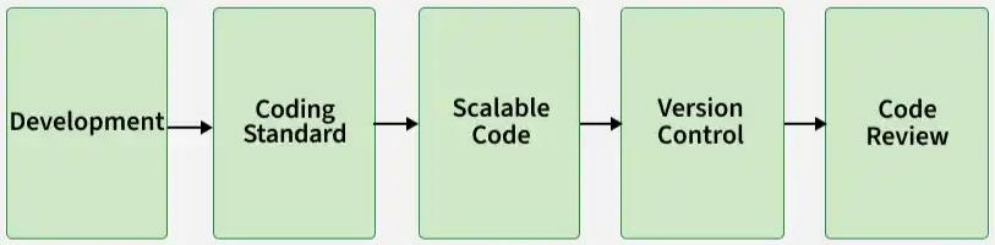
- **High-Level Design (HLD)**
  Defines system architecture, technology stack, database design, and major modules
- **Low-Level Design (LLD)**
  Specifies component logic, APIs, data structures, and workflows
- **Output**
  Design Document Specification (DDS)



3.1.4. Stage 4: Development (Coding)

Developers build the software based on the approved design

- **Activities**
  Coding, code reviews, unit testing, version control management
- **Tools**
  IDEs, version control systems, debuggers
- **Output**
  Source code, executable application
- **Key Roles**
  Frontend, Backend, Full Stack Developers



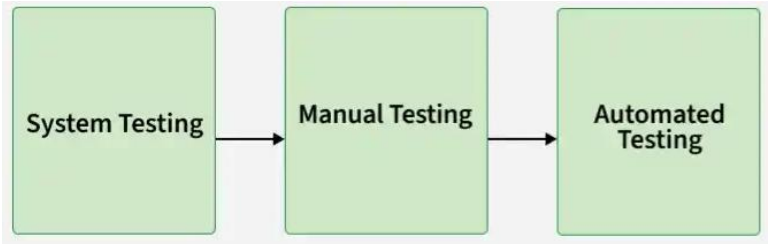
3.1.5. Stage 5: Testing

Testing ensures the software meets requirements and is free from defects before release

Types of Testing include

- **Unit Testing**
  Verifies individual components
- **Integration Testing**
  Ensures modules work together
- **System Testing**
  Validates the complete system
- **User Acceptance Testing (UAT)**
  Confirms business requirements are met
- **Output**
  Test cases, defect reports, and quality metrics.

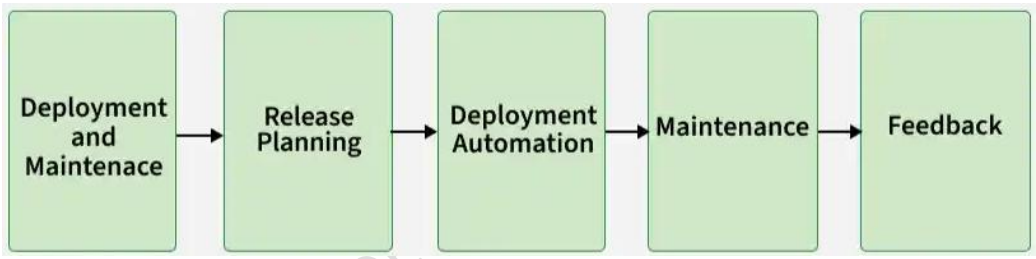
© i3Qube Software 2023 Confidential



3.1.6. Stage 6: Deployment

The tested software is released to users

- **Activities**
  Production setup, deployment, smoke testing
- **Modern Approach**
  Continuous Integration and Continuous Deployment (CI/CD) pipelines for faster and reliable releases
- **Output**
  Live application
- **Key Roles**
  DevOps Engineers, Release Managers



3.1.7. Stage 7: Maintenance

Post-deployment support ensures long-term usability

- **Activities**
  Bug fixes, performance tuning, updates, feature enhancements
- **Output**
  Patches, updates, new versions
- **Key Roles**
  Support Engineers, Developers

3.2. Software Development Life Cycle Models

Software Development Models are structured frameworks that guide the planning, execution, and delivery of software projects. They define the sequence of development stages, such as requirements, design, coding, testing, and deployment

- Waterfall Model
- Agile Model
- V-Model
- Spiral Model
- Incremental Model
- RAD Model

3.3. Embedding Security into SDLC

Security is integrated throughout the Software Development Life Cycle using a DevSecOps approach. It is built into every stage, from design to deployment, ensuring continuous protection. Embedding security into the SDLC reduces risks, improves software resilience, and enables the delivery of safer applications

- Vulnerabilities are identified and fixed early in the development process

© i3Qube Software 2023 Confidential

- Automated security checks are integrated into build and CI/CD pipelines
- Security becomes a shared responsibility across development, testing, and operations teams

3.4. Common SDLC Mistakes Teams Make

Even experienced teams misuse SDLC by focusing on form over substance

- **Treating documentation as the goal**
  Producing SRS and design documents without validating assumptions with real users
- **Late or inadequate testing**
  Testing is rushed or reduced due to time pressure, increasing post-release defects
- **Ignoring non-functional requirements**
  Performance, security, and scalability are considered too late
- **Poor communication between teams**
  Gaps between business, development, and QA lead to misaligned outcomes
- **Skipping feedback loops**
  Limited iteration and user feedback cause issues to surface only after deployment
- **Over engineering early**
  Designing overly complex solutions before validating core functionality

Avoiding these mistakes requires continuous collaboration, early validation, and a mindset that values outcomes over processes

3.5. Real Life Example of SDLC

Banking Application Development using SDLC

- **Planning & Analysis**
  Identify banking features such as account management, fund transfers, and security requirements
- **Design**
  Create UI designs, system architecture, databases, and technology stack
- **Development**
  Implement frontend interfaces, backend services and APIs
- **Testing**
  Conduct functional, performance, and security testing
- **Deployment & Maintenance**
  Release the application and continuously monitor, fix defects, and add enhancements

3.6. Reasons for Project Failure Despite Following SDLC

Following SDLC does not automatically guarantee project success. Many projects fail because teams treat SDLC as a checklist rather than a decision-making framework. SDLC provides structure, but success depends on how thoughtfully it is applied, not just whether it is followed

- **Poor requirement clarity**
  Requirements are documented but not deeply understood, leading to incorrect assumptions and rework
- **Weak stakeholder involvement**
  Limited feedback from users or business teams results in solutions that don't solve real problems
- **Rigid execution**
  Teams follow the process mechanically and resist adapting to changing business or technical realities

© i3Qube Software 2023 Confidential

- **Underestimated complexity**
  Risks related to scalability, integration, or performance are identified too late
- **Lack of ownership**
  Roles exist on paper, but accountability for outcomes is unclear

4. SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC) MODELS

Software development models are various processes or methods that are chosen for project development depending on the objectives and goals of the project. Many development life cycle models have been developed to achieve various essential objectives. Models specify the various steps of the process and the order in which they are executed

4.1. Software Modeling

Software modeling is the process of creating abstract representations of a software system. These models serve as blueprints that guide developers, designers, and stakeholders through the system's structure, behavior, and functionality

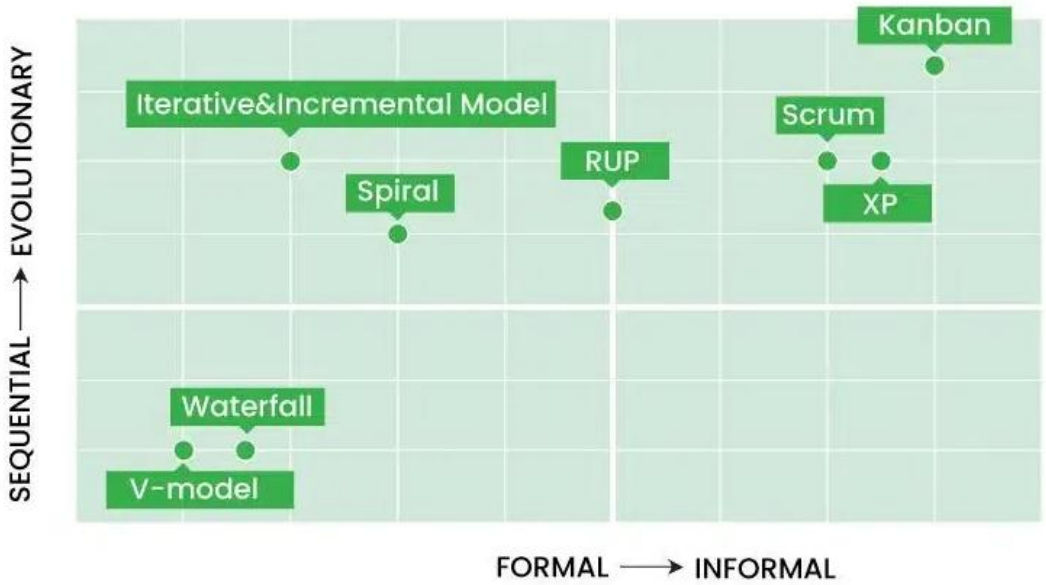
By using diagrams and various modeling languages, software modeling helps in visualizing and understanding the complex aspects of the software, making it easier to plan, develop, and manage the system

4.2. Top 8 Software Development Models

Choosing the right model is very important for the Development of a Software Product or application. Development and testing processes are carried out based on the model

Different companies, depending on the software application or product, choose the type of development model whichever is appropriate for their application. But these days 'Agile Methodology' is the most popular in the market

Let's look at the every model in brief one by one



4.2.1. Agile Model

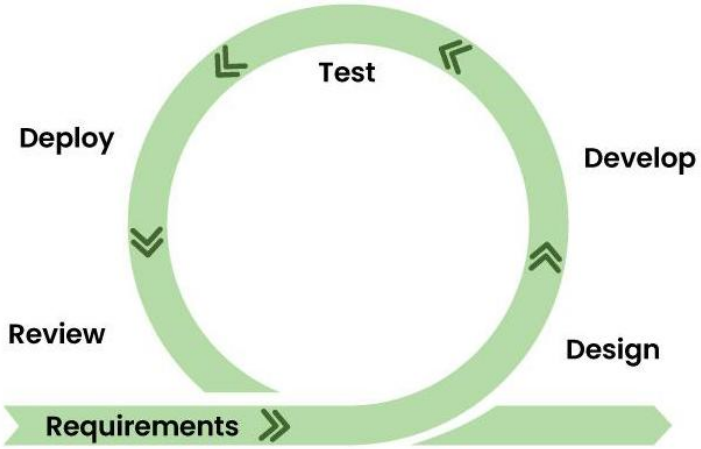
Agile Development Model is a combination of iterative and incremental models, that is, it is made up of iterative and incremental models

- In Agile model, focus is given to process adaptability and customer satisfaction
- In earlier times, iterative waterfall model was used to create software. But in today's time developers have to face many problems. The biggest problem is that in the middle of software

© i3Qube Software 2023 Confidential

development, the customer asks to make changes in the software. It takes a lot of time and money to make these changes. So to overcome all these shortcomings, the agile model was proposed in the 1990s

The Agile Model was created mainly to make changes in the middle of **software development** so that the software project can be completed quickly



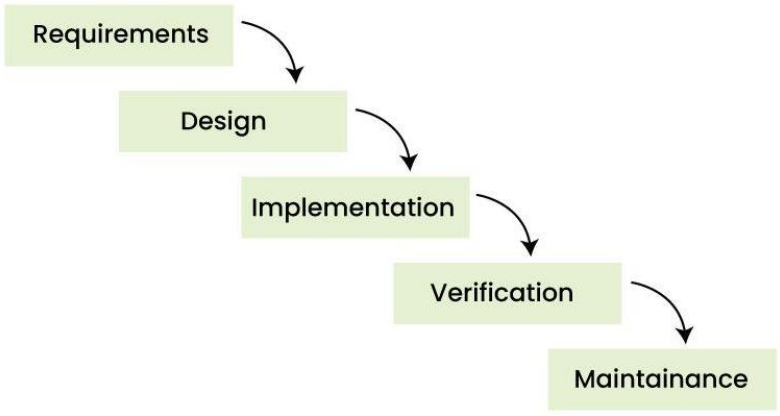
Agile Model

- In the agile model, the software product is divided into small incremental parts. In this, the smallest part is developed first and then the larger one
- And each incremental part is developed over iteration
- Each iteration is kept small so that it can be easily managed. And it can be completed in two-three weeks. Only one iteration is planned, developed and deployed at a time

4.2.2. Waterfall Model

Waterfall Model is a famous and good version of SDLC (System Development Life Cycle) for software engineering. The waterfall model is a linear and sequential model, which means that a development phase cannot begin until the previous phase is completed. We cannot overlap phases in waterfall model

Phases of Waterfall Model



Waterfall Model

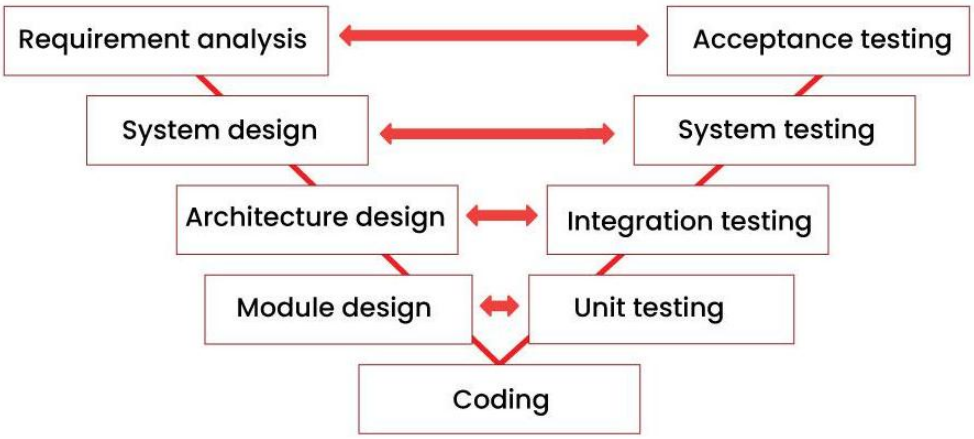
Similarly waterfall model also works, once one phase of development is completed then we move to the next phase but cannot go back to the previous phase. In the waterfall model, the output of one phase serves as the input for the other phase

© i3Qube Software 2023 Confidential

4.2.3. V-Model

V-Model is an SDLC model, it is also called Verification and Validation Model. V-Model is widely used in the **Software Development Process**, and it is considered a disciplined model. In V-Model, the execution of each process is sequential, that is, the new phase starts only after the previous phase ends

Phases of V-model

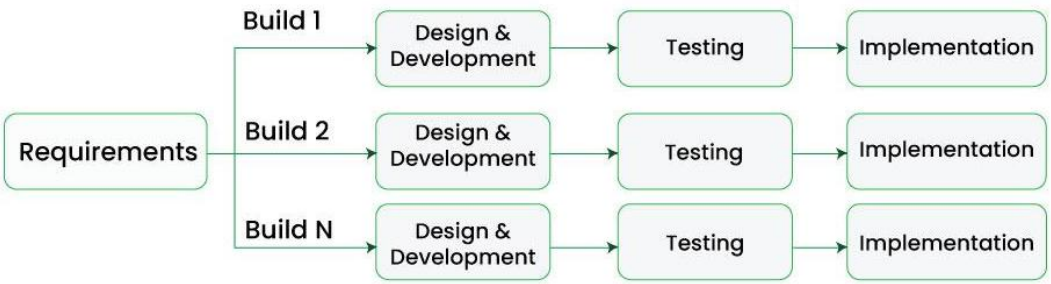


V-Model

- It is based on the association of testing phase with each development phase that is in V-Model with each development phase, its testing phase is also associated in a V-shape in other words both Software Development and testing activities take place at the same time
- So in this model, Verification Phase will be on one side, Validation Phase will be on the other side that is both the activities run simultaneously and both of them are connected to each other in V-Shape through Coding Phase, hence it is called V-Model
- **V-Design**
  In V-Design the left side represents the development activity, the right side represents the testing activity

4.2.4. Incremental Model

In Incremental Model, the **Software Development Process** is divided into several increments and the same phases are followed in each increment. In simple language, under this model a complex project is developed in many modules or builds



Incremental Model

- For example, we collect the customer's requirements, now instead of making the entire software at once, we first take some requirements and based on them create a module or function of the software and deliver it to the customer. Then we take some more requirements and based on them add another module to that software

© i3Qube Software 2023 Confidential

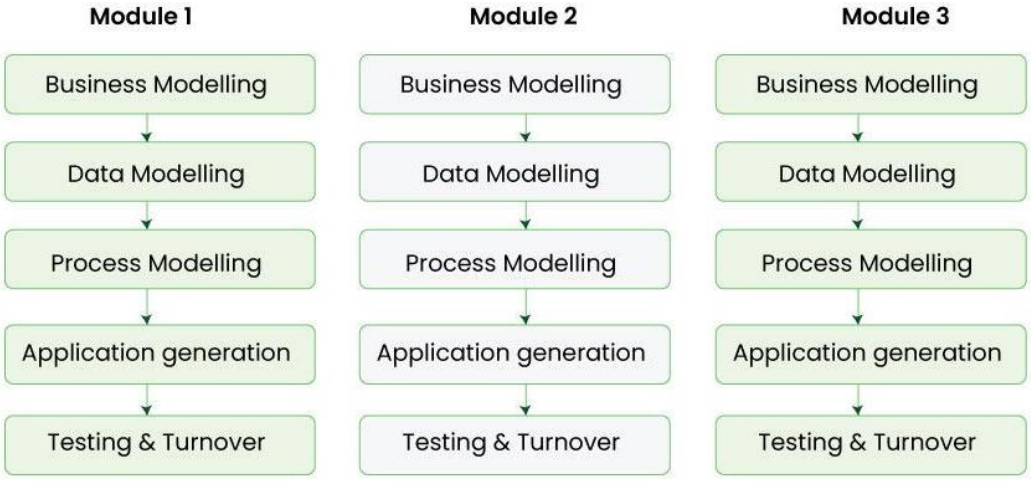
- Similarly, modules are added to the software in each increment until the complete system is created. However, the requirements for making a complex project in multiple iterations/parts should be clear
- If we understand the entire principle of Incremental methodology, then it starts by developing an initial implementation, then user feedback is taken on it, and it is developed through several versions until an accepted system is developed. Important functionalities of the software are developed in the initial iterations

Each subsequent release of a software module adds functions to the previous release. This process continues until the final software is obtained

4.2.5. RAD Model

RAD Model stands for rapid application development model. The methodology of RAD model is similar to that of incremental or waterfall model. It is used for small projects

The main objective of RAD model is to reuse code, components, tools, processes in project development



RAD Model

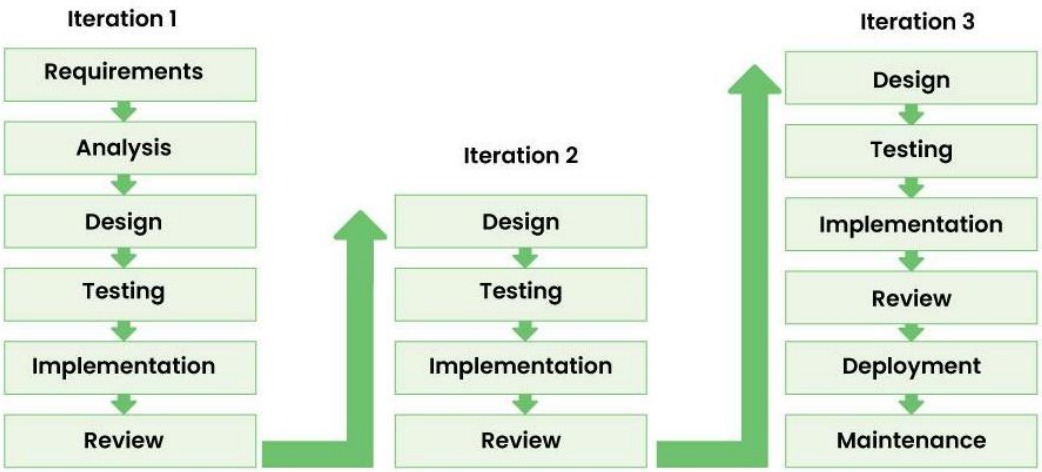
- If the project is large then it is divided into many small projects and these small projects are planned one by one and completed. In this way, by completing small projects, the large project gets ready quickly
- In RAD model, the project is completed within the given time and all the requirements are collected before starting the project. It is very fast and there are very less errors in it

4.2.6. Iterative Model

In Iterative Model we start developing the software with some requirements and when it is developed, it is reviewed. If there are requirements for changes in it, then we develop a new version of the software based on those requirements. This process repeats itself many times until we get our final product

Through this diagram you can understand the Interactive model

© i3Qube Software 2023 Confidential



Iterative Model

- So, in Iterative model a software is developed by following several iterations. Iteration means that we are repeating the Development Process again and again. For example, we develop the first version of the software following the SDLC process with some software requirements
- After the first version is developed, if there is a need to change the software , then a new version is developed with the second iteration. Now again we will see if the new version is enough, if not then we will make changes in it with the third iteration. The iteration will be repeated until the complete software is ready
- The basic concept of Iterative model is that the software should be developed through repeated cycles or what we also call iteration and only a small part of it should be developed at a time. This model was developed to overcome the drawbacks of the classical waterfall model

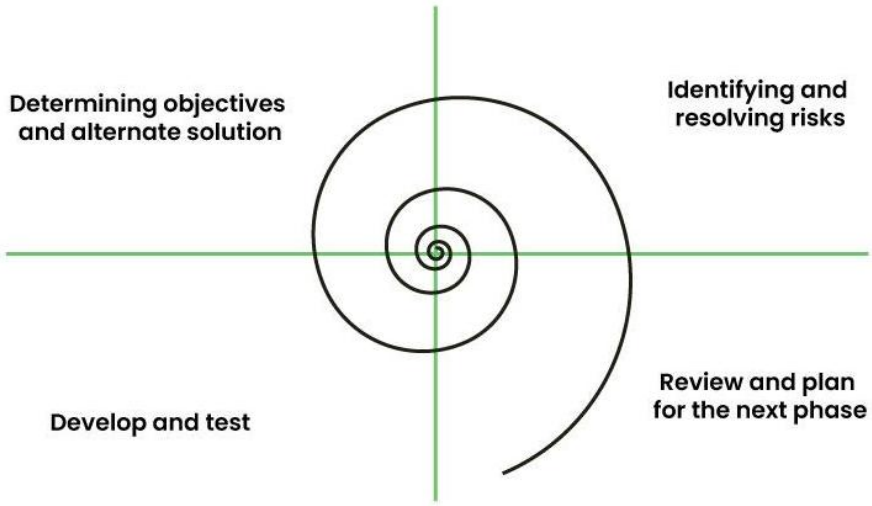
4.2.7. Spiral Model

Spiral Model is a software development process model. This model has characteristics of both iterative and waterfall models. This model is used in projects which are large and complex.

This model was named spiral because if we look at its figure, it looks like a spiral, in which a long curved line starts from the center point and makes many loops around it

The number of loops in the spiral is not decided in advance but it depends on the size of the project and the changing requirements of the user. We also call each loop of the spiral a phase of the software development process

© i3Qube Software 2023 Confidential



Spiral Model

In Spiral Model the entire process of software development is described in four phases which are repeated until the project is completed

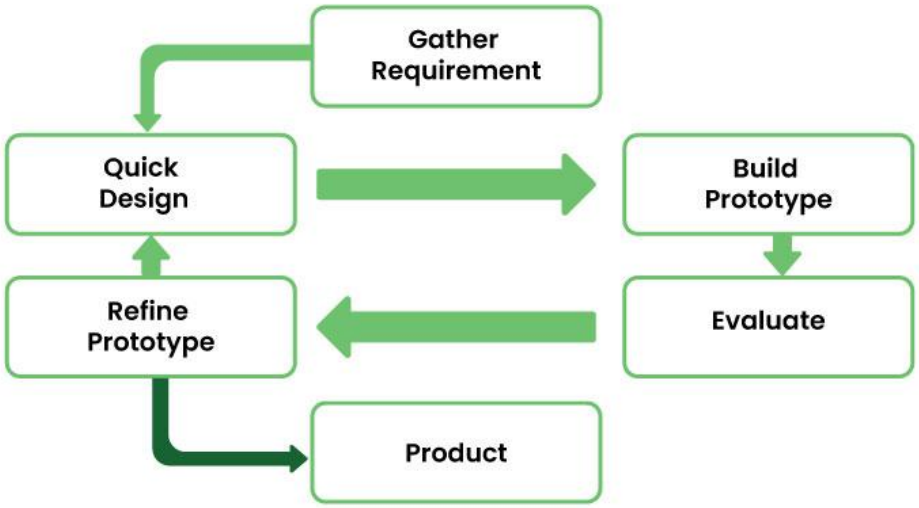
A software project goes through these loops again and again in iterations. After each iteration a more and more complete version of the software is developed. The most special thing about this model is that risks are identified in each phase and they are resolved through prototyping. This feature is also called Risk Handling

Since it also includes the approaches of other SDLC models, it is also called Meta Model. It was first developed by Barry Boehm in 1986

4.2.8. Prototype Model

Prototype Model is an activity in which prototypes of software applications are created. First a prototype is created and then the final product is manufactured based on that prototype

One problem in this model is that if the end users are not satisfied with the prototype model, then a new prototype model is created again, due to which this model consumes a lot of money and time



Prototype Model

- The prototype model was developed to overcome the shortcomings of the waterfall model
- This model is created when we do not know the requirements well

© i3Qube Software 2023 Confidential

- The specialty of this model is that this model can be used with other models as well as alone

4.3. Why Companies are shifting towards agile Software Development models

In earlier times, iterative waterfall model was used to create software. But in today's time developers have to face many problems.

The biggest problem is that in the middle of software development, the customer asks to make changes in the software.

It takes a lot of time and money to make these changes. The agile model was created mainly to make changes in the middle of Software Development so that the software project can be completed quickly

Agile focuses on customer-centric approach that constantly take feedback from customer and make changes in the middle of software developer according to their needs and expectation.

Agile Methodologies like scrum adopt iterative and incremental approach this accelerate the time-to-market for new features and product

4.4. Conclusion

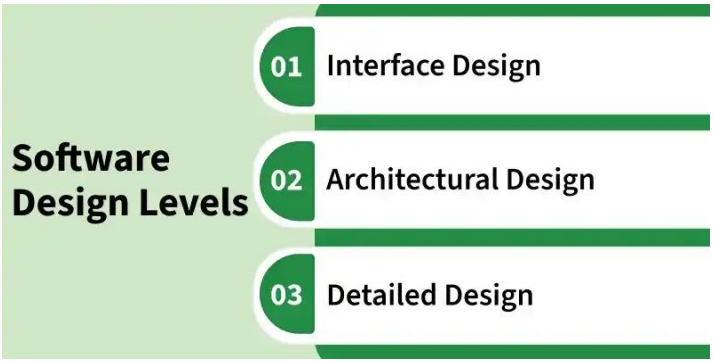
Choosing the right software development model is important for the Success of a Project**.** Each model offers unique advantages and is suitable for Different Types of Projects**.** Agile Methodologies, due to their flexibility and customer-centric approach, are increasingly popular in the industry. By adopting Agile, companies can accelerate time-to-market and better adapt to changing requirements, leading to higher customer satisfaction and Successful Project Outcomes

5. SOFTWARE DESIGN AND QUALITY

5.1. Software Design

Software Design processes the phase where developers plan how to turn a set of requirements into a working system like a blueprint for the software

- Break down complex requirements into smaller, manageable modules or components
- It bridges functional and non-functional requirements to coding, preventing costly rework
- Design the system architecture and define how different components interact and work together



Software Design Levels

5.1.1. Software Design Levels

The software design process can be divided into the following three levels or phases of design

1. **INTERFACE DESIGN**

   It is the specification of the interaction between a system and its environment. This phase proceeds at a high level of abstraction with respect to the inner workings of the system i.e, during

© i3Qube Software 2023 Confidential

interface design, the internal of the systems are completely ignored, and the system is treated as a black box.

Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts.

The design problem statement produced during the problem analysis step should identify the people, other systems, and devices which are collectively called agents

Interface design should include the following details

- Precise description of events in the environment, or messages from agents to which the system must respond
- Precise description of the events or messages that the system must produce
- Specification of the data, and the formats of the data coming into and going out of the system
- Specification of the ordering and timing relationships between incoming events or messages, and outgoing events or outputs

2. **ARCHITECTURAL DESIGN**

It is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them.

In architectural design, the overall structure of the system is chosen, but the internal details of major components are ignored. Issues in architectural design includes

- Gross decomposition of the systems into major components
- Allocation of functional responsibilities to components
- Component Interfaces
- Component scaling and performance properties, resource consumption properties, reliability properties, and so forth
- Communication and interaction between components

3. **DETAILED DESIGN**

It is the specification of the internal elements of all major system components, their properties, relationships, processing and often their algorithms and the data structures

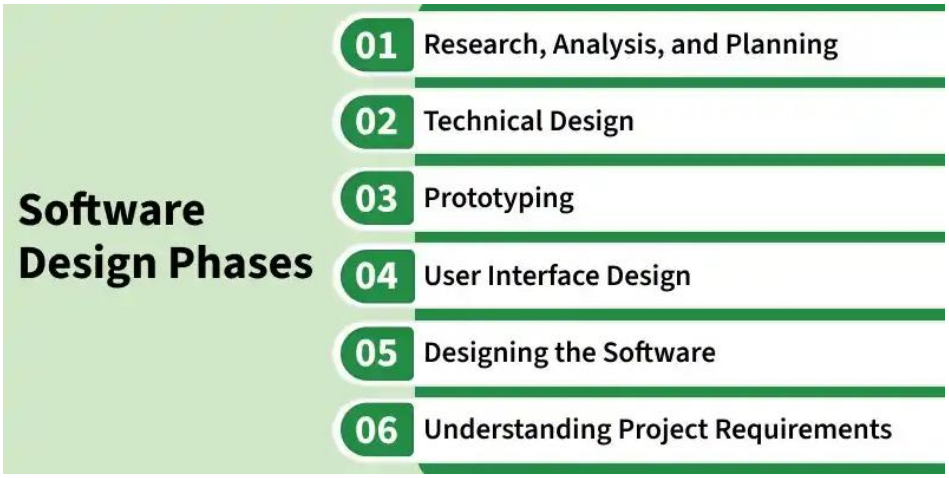
The detailed design may include

- Decomposition of major system components into program units
- Allocation of functional responsibilities to units
- User interfaces
- Unit states and state changes
- Data and control interaction between units
- Data packaging and implementation, including issues of scope and visibility of program elements
- Algorithms and data structures

5.1.2. Software Design Phases

The Actual Software Design process travel through the one end to another end of making and Software well with completion of requirements and the phases which included in the same that are bellow

© i3Qube Software 2023 Confidential



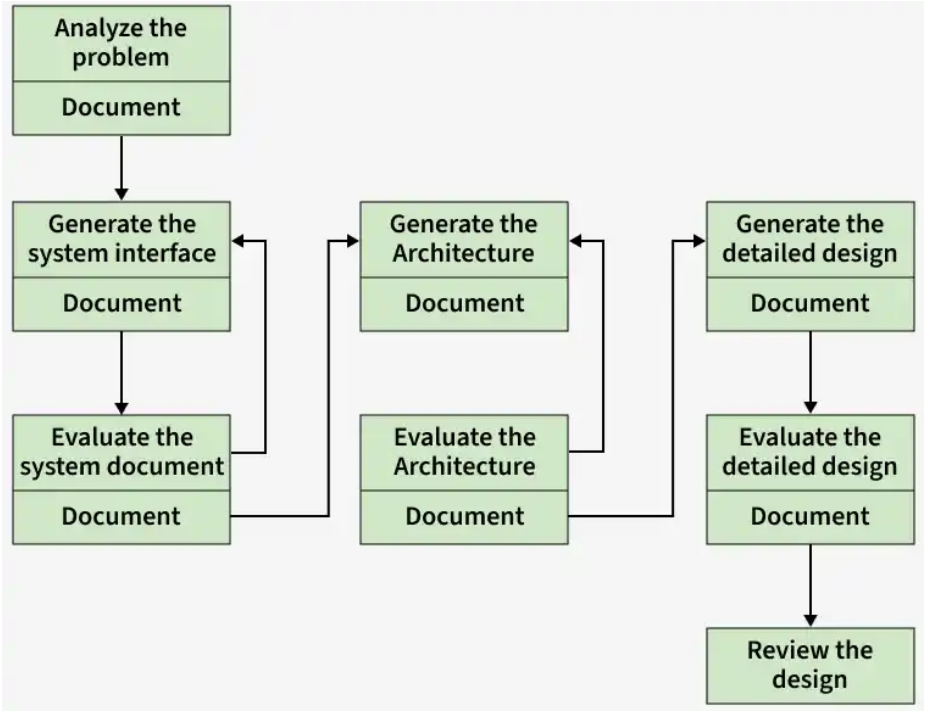
Software Design Phases

1. **Understanding Project Requirements**
   Before jumping into the design, the first step is to make sure the team understands what the users need, what the business goals are, and any potential challenges. This understanding sets the foundation for creating a design that meets both user and business needs
2. **Research, Analysis, and Planning**
   During this phase, the team gathers data through methods like interviews, surveys, and focus groups. This helps get a clearer picture of what the users want and allows the team to design with the user in mind, ensuring the software will truly meet their needs
3. **Designing the Software**
   In the design phase, the team starts creating visual elements like wireframes, user stories, and flow diagrams to map out how the system will work. Based on the feedback, prototypes are created and fine-tuned to make sure the design is moving in the right direction
4. **Technical Design**
   After gathering feedback, the team gets into more details with the technical design. This phase involves creating a thorough technical document that outlines exactly how the software will be implemented, including specific components and how they will work together
5. **User Interface Design**
   The focus here is on making sure the software is easy to use. UI designers work on the visual elements, navigation, and overall user experience to ensure the interface is intuitive and friendly
6. **Prototyping**
   Prototypes are created to visualize the design and functionality before the full development process begins. These can range from simple wireframes (low fidelity) to fully interactive models (high fidelity) depending on the stage of development

5.1.3. Elements of Software Design

Good software design is built on several core element that work together to create effective system

© i3Qube Software 2023 Confidential



Software Design Process

1. **Architecture**
   This is the conceptual model that defines the structure, behavior, and views of a system. We can use flowcharts to represent and illustrate the architecture. A solid architecture verify the system is flexible, stable, and easy to maintain over time
2. **Modules**
   Modules as the building blocks of the system. Each one handles a specific task or feature. Breaking a system into smaller modules makes it easier to develop, test, and maintain the system. These are components that handle one specific task in a system. A combination of the modules makes up the system
3. **Components**
   This provides a particular function or group of related functions. They are made up of modules. Organizing the system into components helps keep the code clean and makes the system more adaptable
4. **Interfaces**
   These are smaller units within modules that focus on specific functions. This is the shared boundary across which the components of a system exchange information and relate
5. **Data**
   Data is at the heart of any system. It's all about how information is stored, accessed, and shared. This is the management of the information and data flow

5.1.4. How Software Design Fits into SDLC

Software Design comes when the project requirement are done and when we are about to start the Development process, Here is How Software Design fits into the Software Development Life Cycle

- **Design the System Architecture**
  Create the overall structure of the software, including modules, components, and their interactions
- **Prepare Design Artefacts**

© i3Qube Software 2023 Confidential

Develop wireframes, UI/UX designs, data flow diagrams (DFDs), and database designs to visualize the system

- **Define Component Functionality**
  Specify how each component will work together to ensure smooth functionality and efficient data flow
- **Begin Software Development**
  Once the design is finalized and approved, the implementation (coding) phase starts, where developers build the actual software

5.1.5. Tools for Software Design

There are many tools that make the process easier and more efficient, whether you're working on wireframes, prototypes, or documentation. Here are some popular tools

1. Figma
2. Balsamiq
3. Axure RP
4. Sketch
5. InVision Studio

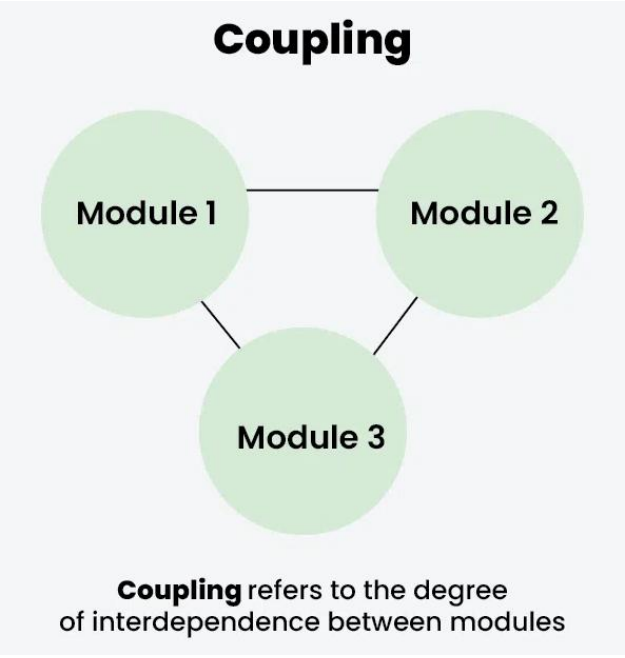
5.2. Coupling and Cohesion

The purpose of the Design phase in the Software Development Life Cycle is to produce a solution to a problem given in the SRS(Software Requirement Specification) document. The output of the design phase is a Software Design Document (SDD)

Coupling and Cohesion are two key concepts in software engineering that are used to measure the quality of a software system's design

5.2.1. Coupling and Cohesion

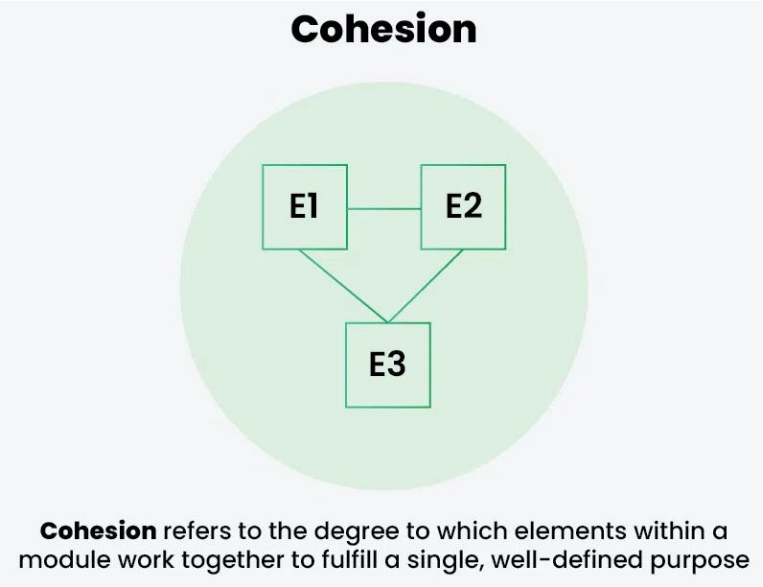
Coupling refers to the degree of interdependence between software modules. High coupling means that modules are closely connected and changes in one module may affect other modules. Low coupling means that modules are independent, and changes in one module have little impact on other modules



Coupling

© i3Qube Software 2023 Confidential

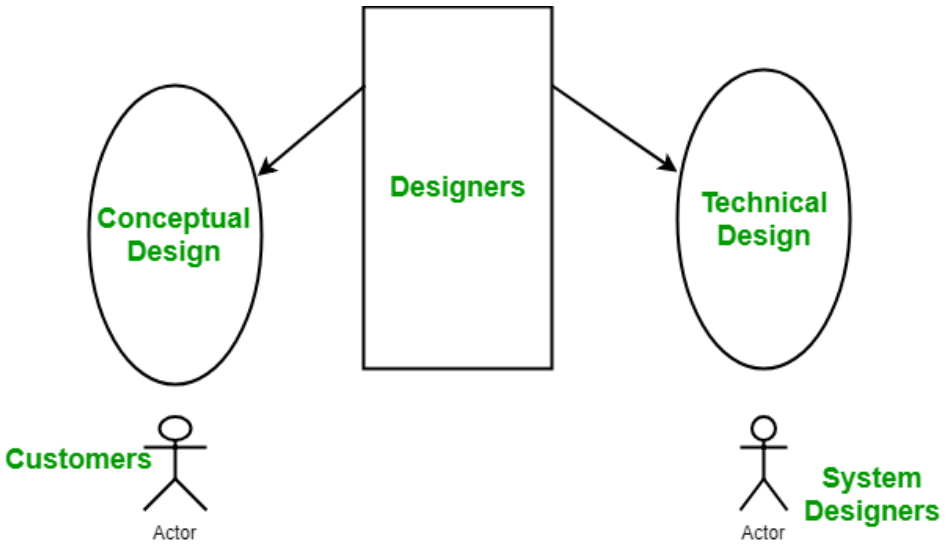
Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose. High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes



Cohesion

Both coupling and cohesion are important factors in determining the maintainability, scalability, and reliability of a software system. High coupling and low cohesion can make a system difficult to change and test, while low coupling and high cohesion make a system easier to maintain and improve

Basically, design is a two-part iterative process. The first part is Conceptual Design which tells the customer what the system will do. Second is Technical Design which allows the system builders to understand the actual hardware and software needed to solve a customer's problem



© i3Qube Software 2023 Confidential

5.2.2. Conceptual Design of the System

- Written in simple language i.e. customer understandable language
- Detailed explanation about system characteristics
- Describes the functionality of the system
- It is independent of implementation
- Linked with requirement document

5.2.3. Technical Design of the System

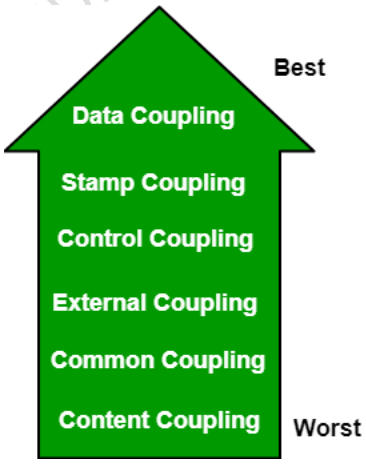
- Hardware component and design
- Functionality and hierarchy of software components
- Software architecture
- Network architecture
- Data structure and flow of data
- I/O component of the system
- Shows interface

Modularization is the process of dividing a software system into multiple independent modules where each module works independently. There are many advantages of Modularization in software engineering. Some of these are given below

- Easy to understand the system
- System maintenance is easy
- A module can be used many times as their requirements. No need to write it again and again

5.2.4. Types of Coupling

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



Following are the types of Coupling

- **Data Coupling**
  If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent of each other and communicate through data. Module communications don't contain tramp data. Example-customer billing system
- **Stamp Coupling**
  In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice was made by the insightful designer, not a lazy programmer
- **Control Coupling**

© i3Qube Software 2023 Confidential

If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument

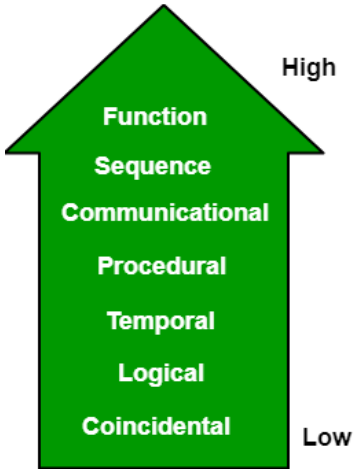
- **External Coupling**
  In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc
- **Common Coupling**
  The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses, and reduced maintainability
- **Content Coupling**
  In a content coupling, one module can modify the data of another module, or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided
- **Temporal Coupling**
  Temporal coupling occurs when two modules depend on the timing or order of events, such as one module needing to execute before another. This type of coupling can result in design issues and difficulties in testing and maintenance
- **Sequential Coupling**
  Sequential coupling occurs when the output of one module is used as the input of another module, creating a chain or sequence of dependencies. This type of coupling can be difficult to maintain and modify
- **Communicational Coupling**
  Communicational coupling occurs when two or more modules share a common communication mechanism, such as a shared message queue or database. This type of coupling can lead to performance issues and difficulty in debugging
- **Functional Coupling**
  Functional coupling occurs when two modules depend on each other's functionality, such as one module calling a function from another module. This type of coupling can result in tightly-coupled code that is difficult to modify and maintain
- **Data-Structured Coupling**
  Data-structured coupling occurs when two or more modules share a common data structure, such as a database table or data file. This type of coupling can lead to difficulty in maintaining the integrity of the data structure and can result in performance issues
- **Interaction Coupling**
  Interaction coupling occurs due to the methods of a class invoking methods of other classes. Like with functions, the worst form of coupling here is if methods directly access internal parts of other methods. Coupling is lowest if methods communicate directly through parameters
- **Component Coupling**
  Component coupling refers to the interaction between two classes where a class has variables of the other class. Three clear situations exist as to how this can happen. A class C can be component coupled

© i3Qube Software 2023 Confidential

with another class C1, if C has an instance variable of type C1, or C has a method whose parameter is of type C1,or if C has a method which has a local variable of type C1. It should be clear that whenever there is component coupling, there is likely to be interaction coupling

5.2.5. Types of Cohesion

Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion



Following are the Types of Cohesion

- **Functional Cohesion**
  Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation
- **Sequential Cohesion**
  An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages
- **Communicational Cohesion**
  Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer
- **Procedural Cohesion**
  Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA
- **Temporal Cohesion**
  The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time
- **Logical Cohesion**
  The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different

© i3Qube Software 2023 Confidential

- **Coincidental Cohesion**
  The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex- print next line and reverse the characters of a string in a single component
- **Procedural Cohesion**
  This type of cohesion occurs when elements or tasks are grouped together in a module based on their sequence of execution, such as a module that performs a set of related procedures in a specific order. Procedural cohesion can be found in structured programming languages
- **Communicational Cohesion**
  Communicational cohesion occurs when elements or tasks are grouped together in a module based on their interactions with each other, such as a module that handles all interactions with a specific external system or module. This type of cohesion can be found in object-oriented programming languages
- **Temporal Cohesion**
  Temporal cohesion occurs when elements or tasks are grouped together in a module based on their timing or frequency of execution, such as a module that handles all periodic or scheduled tasks in a system. Temporal cohesion is commonly used in real-time and embedded systems
- **Informational Cohesion**
  Informational cohesion occurs when elements or tasks are grouped together in a module based on their relationship to a specific data structure or object, such as a module that operates on a specific data type or object. Informational cohesion is commonly used in object-oriented programming
- **Functional Cohesion**
  This type of cohesion occurs when all elements or tasks in a module contribute to a single well-defined function or purpose, and there is little or no coupling between the elements. Functional cohesion is considered the most desirable type of cohesion as it leads to more maintainable and reusable code
- **Layer Cohesion**
  Layer cohesion occurs when elements or tasks in a module are grouped together based on their level of abstraction or responsibility, such as a module that handles only low-level hardware interactions or a module that handles only high-level business logic. Layer cohesion is commonly used in large-scale software systems to organize code into manageable layers

5.2.6. Advantages of Low Coupling

- Improved maintainability: Low coupling reduces the impact of changes in one module on other modules, making it easier to modify or replace individual components without affecting the entire system
- Enhanced modularity: Low coupling allows modules to be developed and tested in isolation, improving the modularity and reusability of code
- Better scalability: Low coupling facilitates the addition of new modules and the removal of existing ones, making it easier to scale the system as needed

5.2.7. Advantages of High Cohesion

© i3Qube Software 2023 Confidential

- **Improved readability and understand-ability**
  High cohesion results in clear, focused modules with a single, well-defined purpose, making it easier for developers to understand the code and make changes
- **Better error isolation**
  High cohesion reduces the likelihood that a change in one part of a module will affect other parts, making it easier to
- **Improved reliability**
  High cohesion leads to modules that are less prone to errors and that function more consistently
- leading to an overall improvement in the reliability of the system

5.2.8. Disadvantages of High Coupling

- **Increased complexity**
  High coupling increases the interdependence between modules, making the system more complex and difficult to understand
- **Reduced flexibility**
  High coupling makes it more difficult to modify or replace individual components without affecting the entire system
- **Decreased modularity**
  High coupling makes it more difficult to develop and test modules in isolation, reducing the modularity and reusability of code

5.2.9. Disadvantages of Low Cohesion

- **Increased code duplication**
  Low cohesion can lead to the duplication of code, as elements that belong together are split into separate modules
- **Reduced functionality**
  Low cohesion can result in modules that lack a clear purpose and contain elements that don't belong together, reducing their functionality and making them harder to maintain
- **Difficulty in understanding the module**
  Low cohesion can make it harder for developers to understand the purpose and behavior of a module, leading to errors and a lack of clarity

5.2.10. Conclusion

In conclusion, it's good for software to have low coupling and high cohesion. Low coupling means the different parts of the software don't rely too much on each other, which makes it safer to make changes without causing unexpected problems.

High cohesion means each part of the software has a clear purpose and sticks to it, making the code easier to work with and reuse. Following these principles helps make software stronger, more adaptable, and easier to grow

5.3. Software Quality

Software quality is the degree to which a product meets specified requirements and user expectations. It is categorised by functional suitability and structural quality (how well it performs, scales, and maintains)

- It measures how reliable, efficient, and user-friendly a software product is
- Functionally correct software can still be low quality if it has poor usability or user experience
- High-quality software meets functional requirements, providing good usability and performance

© i3Qube Software 2023 Confidential



Factors of Software Quality

Though "fitness of purpose" could be a satisfactory definition of quality for some products like an automobile, a table fan, a grinding machine, etc. for code products, "fitness of purpose" isn't a completely satisfactory definition of quality

5.3.1. Factors of Software Quality

The modern read of high-quality associates with software many quality factors like the following

1. **Portability**
   A software is claimed to be transportable, if it may be simply created to figure in several package environments, in several machines, with alternative code products, etc
2. **Usability**
   A software has smart usability if completely different classes of users (i.e. knowledgeable and novice users) will simply invoke the functions of the products
3. **Reusability**
   A software has smart reusability if completely different modules of the products will simply be reused to develop new products
4. **Correctness**
   Software is correct if completely different needs as laid out in the SRS document are properly enforced
5. **Maintainability**
   A software is reparable, if errors may be simply corrected as and once they show up, new functions may be simply added to the products, and therefore the functionalities of the products may be simply changed, etc
6. **Reliability**
   Software is more reliable if it has fewer failures. Since software engineers do not deliberately plan for their software to fail, reliability depends on the number and type of mistakes they make. Designers can improve reliability by ensuring the software is easy to implement and change, by testing it thoroughly, and also by ensuring that if failures occur, the system can handle them or can recover easily
7. **Efficiency**
   The more efficient software is, the less it uses of CPU-time, memory, disk space, network bandwidth, and other resources. This is important to customers in order to reduce their costs of running the

© i3Qube Software 2023 Confidential

software, although with today's powerful computers, CPU time, memory and disk usage are less of a concern than in years gone by

5.3.2. Software Quality Management System

Software Quality Management System contains the methods that are used by the authorities to develop products having the desired quality

- **Managerial Structure**
  Quality System is responsible for managing the structure as a whole. Every Organization has a managerial structure
- **Individual Responsibilities**
  Each individual present in the organization must have some responsibilities that should be reviewed by the top management and each individual present in the system must take this seriously

Quality System Activities

The activities which each quality system must have been

- Project Auditing
- Review of the quality system
- It helps in the development of methods and guidelines

5.3.3. Evolution of Quality Management System

Quality Systems are basically evolved over the past some years. The evolution of a Quality Management System is a four-step process



Evolution of Quality Management System

1. **Inspection**
   Product inspection task provided an instrument for quality control (QC)
2. **Quality Control**
   The main task of quality control is to detect defective devices, and it also helps in finding the cause that leads to the defect. It also helps in the correction of bugs
3. **Quality Assurance**
   Quality Assurance helps an organization in making good quality products. It also helps in improving the quality of the product by passing the products through security checks

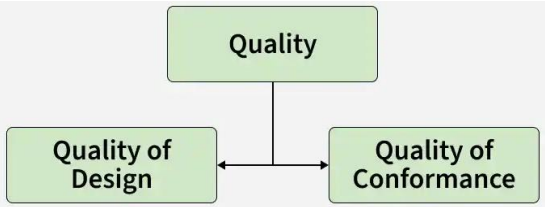
© i3Qube Software 2023 Confidential

4. **Total Quality Management (TQM)**
   Total Quality Management(TQM) checks and assures that all the procedures must be continuously improved regularly through process measurements

5.4. Software Quality Assurance

Software Quality Assurance (SQA) is a systematic approach used to ensure that software development processes and work products follow defined quality standards and procedures. It focuses on preventing defects, improving development processes, and maintaining software quality throughout the software development life cycle (SDLC)

- Helps prevent defects by identifying process issues early
- Supports quality activities throughout the software development life cycle



Two Primary Dimensions of Quality

5.4.1. Quality in Software

Software quality refers to the degree to which a software product satisfies specified requirements and meets the needs of its users. It includes both the quality of the development process and the quality of the final software product

1. **Quality of Design**
   - Ensures that requirements and specifications correctly reflect customer needs
   - Decided before development begins
   - Focuses on planning, architecture, and system design
2. **Quality of Conformance**
   - Measures how well the final product matches the design specifications
   - Evaluated during and after development
   - Focuses on implementation, testing, and delivery

5.4.2. Focus of Software Quality Assurance (SQA)

SQA focuses on establishing and maintaining processes that help achieve the required level of software quality. Important quality characteristics include



Software Quality Assurance Characteristics

© i3Qube Software 2023 Confidential

- **Functional Suitability**
  The software provides the functions needed to meet specified requirements
- **Performance Efficiency**
  The software performs efficiently while using appropriate system resources
- **Compatibility**
  The software can work and interact effectively with other systems or environments
- **Usability**
  The software is easy for users to learn, understand, and operate
- **Reliability**
  The software performs consistently and dependably under specified conditions
- **Security**
  The software protects data and resources from unauthorized access and other security threats
- **Maintainability**
  The software can be easily analyzed, modified, corrected, and maintained
- **Portability**
  The software can be transferred and adapted to different environments with reasonable effort

5.4.3. Process of SQA

SQA activities are performed throughout the SDLC to establish, monitor, and improve software quality.

- **SQA Planning**
  Defines quality objectives, applicable standards, procedures, responsibilities, and quality activities
- **Requirements Review**
  Reviews requirements for correctness, completeness, consistency, and testability
- **Process Definition**
  Establishes development, documentation, coding, review, and testing procedures
- **Process Monitoring**
  Monitors project activities to ensure that defined processes and standards are followed
- **Reviews and Inspections**
  Examines requirements, designs, code, and other work products to identify issues early
- **Auditing**
  Evaluates processes and work products against defined standards and procedures
- **Verification and Testing**
  Checks whether software work products and the final product satisfy specified requirements
- **Defect Management**
  Records, tracks, analyzes, and verifies defects throughout their lifecycle
- **Process Improvement**

© i3Qube Software 2023 Confidential

Analyzes process performance and applies improvements to increase efficiency and quality

5.4.4. Process of SQA

Several standards and models help organizations establish, manage, and evaluate software quality and development processes

- **ISO 9001**
  Provides requirements for a quality management system that can be applied to organizations and their processes
- **ISO/IEC 25010**
  Defines a software product quality model with characteristics used to specify, measure, and evaluate software quality. The current edition is ISO/IEC 25010:2023
- **ISO/IEC/IEEE 12207**
  Defines software life cycle processes for activities such as development, operation, maintenance, and support
- **ISO/IEC/IEEE 29119**
  Provides internationally agreed standards for software testing, including testing concepts, processes, and test documentation
- **IEEE 730-2026**
  Defines requirements for initiating, planning, controlling, and executing Software Quality Assurance processes for software development and maintenance projects
- **IEEE 1012**
  Defines processes and requirements for software and system verification and validation
- **CMMI**
  Provides a framework for improving organizational processes and assessing process maturity and capability

5.4.5. Responsibilities of SQA Team

- Define and maintain quality processes and guidelines
- Monitor compliance with established standards and procedures
- Conduct reviews, inspections, and audits
- Identify process risks and recommend corrective actions
- Analyze recurring defects and support root cause analysis
- Monitor quality metrics and project quality activities
- Support quality-related training and awareness
- Promote continuous process improvement

5.4.6. Tools Used in Software Quality Assurance

Different tools support SQA activities such as testing, defect management, code analysis, version control, and continuous integration

- **Test Management Tools**
  Manage test cases, test execution, and test results. Examples: TestRail, Zephyr
- **Defect Tracking Tools**
  Record, track, and manage software defects. Examples: Jira, Bugzilla
- **Test Automation Tools**
  Automate software tests to improve efficiency and repeatability. Examples: Selenium, OpenText UFT
- **Static Analysis Tools**
  Analyze source code without executing it to identify defects, vulnerabilities, and coding issues. Examples: SonarQube, Checkstyle

© i3Qube Software 2023 Confidential

- **Performance Testing Tools**
  Evaluate performance, scalability, and system behavior under different workloads. Examples: JMeter, LoadRunner
- **Configuration Management Tools**
  Manage source code versions and changes. Examples: Git, SVN
- **Code Review Tools**
  Support collaborative code reviews and help maintain coding standards. Examples: GitHub, Gerrit
- **Continuous Integration Tools**
  Automate build, integration, and testing activities. Examples: Jenkins, GitLab CI/CD

5.4.7. Limitations

SQA improves software quality, but it cannot eliminate all risks or guarantee defect-free software

- SQA can reduce defects but cannot guarantee completely error-free software
- Reviews, audits, documentation, and monitoring require additional time and effort
- Implementing quality processes, tools, and skilled resources can increase project costs
- Effective SQA depends on appropriate processes, skilled professionals, and organizational support
- Teams may resist changes or additional quality processes.
- Complex systems make it difficult to identify every possible defect, risk, or failure condition

5.4.8. Software Quality Assurance vs Software Testing

Software Quality Assurance and Software Testing are related but differ in their focus and purpose

Aspect	Software Quality Assurance (SQA)	Software Testing
Focus	Process-oriented	Product-oriented
Objective	Prevent defects by improving development processes	Detect defects in the software
Scope	Covers quality activities throughout the SDLC	Performed at different stages of the SDLC
Nature	Primarily preventive	Primarily detection-oriented
Responsibility	Involves the development team, QA professionals, and other stakeholders	Mainly performed by testers, with developers also involved in testing
Activity Type	Process definition, reviews, audits, standards, and process improvement	Test planning, test execution, defect reporting, and verification
Goal	Improve the quality of software processes and products	Verify that software meets specified requirements and identify defects

© i3Qube Software 2023 Confidential

5.5. Boehm's Software Quality Model

In 1978, B.W. Boehm introduced his software quality model, which defines software quality through a hierarchical structure of attributes and metrics. This model is similar to the McCall Quality Model but encompasses a wider range of characteristics, including hardware performance-related ones. Boehm's model categorizes quality attributes into three levels: primary uses (high-level characteristics), intermediate constructs (mid-level characteristics), and primitive constructs (basic characteristics)

5.5.1. What is Boehm's Software Quality Model?

In 1978, B.W. Boehm introduced his software quality model. The model represents a hierarchical quality model similar to the McCall Quality Model to define software quality using a predefined set of attributes and metrics, each of which contributes to the overall quality of software. The difference between Boehm's and McCall's Models is that McCall's Quality Model primarily focuses on precise measurement of high-level characteristics, whereas **Boehm's Quality Model** is based on a wider range of characteristics

Example

Characteristics of hardware performance, that are missing in **McCall's Model**. Boehm's model has three levels for quality attributes. These levels are divided based on their characteristics. These levels are primary uses (high-level characteristics), intermediate constructs(mid-level characteristics), and primitive constructs(primitive characteristics)

5.5.2. Primary Uses of Boehm's Model

The highest level of Boehm's model has the following three primary uses, as stated as below

1. **As is the utility**
   The extent to which, we can use software as-is
2. **Maintainability**
   Effort required to detect and fix an error during maintenance
3. **Portability**
   Effort required to change the software to fit in a new environment

5.5.3. Quality Factors Associated with Boehm's Model

The next level of Boehm's hierarchical model consists of seven quality factors associated with three primary uses, stated below

1. **Portability**
   Effort required to change the software to fit in a new environment
2. **Reliability**
   The extent to which software performs according to requirements
3. **Efficiency**
   Amount of hardware resources and code required to execute a function
4. **Usability (Human Engineering)**
   Extent of effort required to learn, operate and understand functions of the software
5. **Testability**
   Effort required to verify that software performs its intended functions
6. **Understandability**
   Effort required for a user to recognize a logical concept and its applicability
7. **Modifiability**

© i3Qube Software 2023 Confidential

Effort required to modify software during the maintenance phase

5.5.4. Boehm's Primitive Constructs for Software Quality Characteristics

Boehm further classified software quality characteristics into more fundamental components called Primitive Constructs

These include

- Device Independence
- Accuracy
- Completeness
- Consistency
- Device Efficiency
- Accessibility
- Communicativeness
- Self-descriptiveness
- Legibility
- Structuredness
- Conciseness
- Augment-ability

For example, the characteristic of Testability is broken down into accessibility, communicativeness, structuredness, and self-descriptiveness.

5.5.5. Conclusion

Boehm's software quality model provides a comprehensive framework for evaluating software quality by breaking down attributes into detailed primitive constructs.

This allows for a thorough analysis of software quality, addressing various aspects such as portability, reliability, and efficiency. By including more characteristics than McCall's model,

Boehm's approach offers a broader and deeper understanding of software quality, making it a valuable tool for software development and maintenance

5.5.6. Questions for Practice

1. **COCOMO Model is used for [UGC-NET 2008]**

   A. product quality estimation
   B. product complexity estimation
   C. product cost estimation
   D. all of the above

   Solution: Correct Answer is C.

   Explanation: COCOMO Model is the model which was developed by B.W. Boehm that estimates the procedural cost model of software

2. **In a Software Project, COCOMO (Cost Constructive Model) is used to estimate [GATE CS 2004]**

   A. Effort and Duration based on the size of the Software
   B. Size and Duration based on the effort of the Software
   C. Effort and Cost based on the duration of the Software
   D. Size, Effort, and Duration based on the cost of the Software

   Solution: Correct Answer is A.

For More, refer to COCOMO Model.

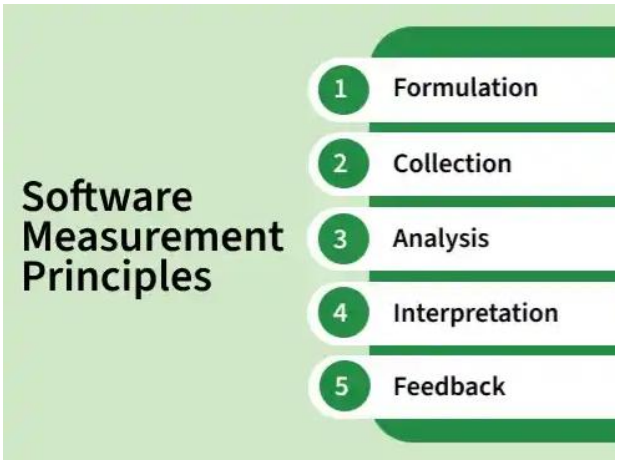
6. SOFTWARE MEASUREMENT AND METRICES

6.1. Software Measurement and Metrics

© i3Qube Software 2023 Confidential

Software measurement is a manifestation of the size, quantity, amount, or dimension of a particular attribute of a product or process

- Software measurement is the quantitative assessment of a characteristic of a software product or process
- It is an important part of software engineering, and its process is defined by ISO standards



Software Measurement Principles

6.1.1. Software Measurement Principles

The software measurement process can be characterized by five activities

1. **Formulation**
   The derivation of software measures and metrics appropriate for the representation of the software that is being considered
2. **Collection**
   The mechanism used to accumulate data required to derive the formulated metrics
3. **Analysis**
   The computation of metrics and the application of mathematical tools
4. **Interpretation**
   The evaluation of metrics results in insight into the quality of the representation
5. **Feedback**
   Recommendation derived from the interpretation of product metrics transmitted to the software team

6.1.2. Need for Software Measurement

- Create the quality of the current product or process
- Anticipate future qualities of the product or process
- Enhance the quality of a product or process
- Regulate the state of the project concerning budget and schedule
- Enable data-driven decision-making in project planning and control
- Identify bottlenecks and areas for improvement to drive process improvement activities
- Ensure that industry standards and regulations are followed
- Give software products and processes a quantitative basis for evaluation
- Enable the ongoing improvement of software development practices

6.1.3. Classification of Software Measurement

There are 2 types of software measurement

© i3Qube Software 2023 Confidential

1. **Direct Measurement**
   In direct measurement, the product, process, or thing is measured directly using a standard scale
2. **Indirect Measurement**
   In indirect measurement, the quantity or quality to be measured is measured using related parameters i.e. by use of reference

6.1.4. Software Metrics

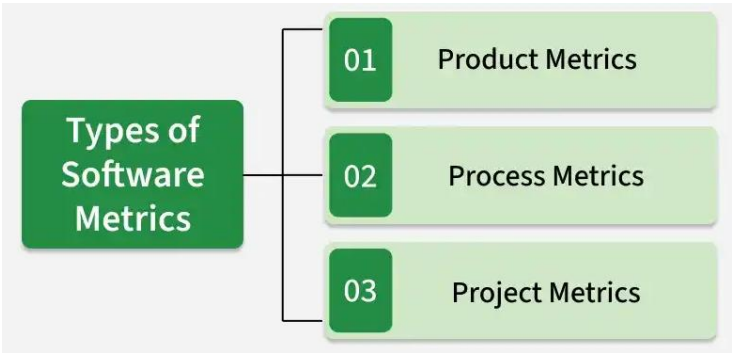
A metric is a measurement of the level at which any impute belongs to a system product or process. Software metrics are a quantifiable or countable assessment of the attributes of a software product. There are 4 functions related to software metrics

1. Planning
2. Organizing
3. Controlling
4. Improving

6.1.5. Characteristics of Software Metrics

- **Quantitative**
  Metrics must possess a quantitative nature. It means metrics can be expressed in numerical values
- **Understandable**
  Metric computation should be easily understood, and the method of computing metrics should be clearly defined
- **Applicability**
  Metrics should be applicable in the initial phases of the development of the software
- **Repeatable**
  When measured repeatedly, the metric values should be the same and consistent
- **Economical**
  The computation of metrics should be economical
- **Language Independent**
  Metrics should not depend on any programming language

6.1.6. Types of Software Metrics



Types of Software Metrics

1. **Product Metrics**
   Product metrics are used to evaluate the state of the product, tracing risks and undercover prospective problem areas. The ability of the team to control quality is evaluated. Examples include lines of code, cyclomatic complexity, code coverage, defect density, and code maintainability index

© i3Qube Software 2023 Confidential

2. **Process Metrics**
   Process metrics pay particular attention to enhancing the long-term process of the team or organization. These metrics are used to optimize the development process and maintenance activities of software. Examples include effort variance, schedule variance, defect injection rate, and lead time

3. **Project Metrics**
   The project metrics describes the characteristic and execution of a project. Examples include effort estimation accuracy, schedule deviation, cost variance, and productivity. Usually measures-

   - Number of software developer
   - Staffing patterns over the life cycle of software
   - Cost and schedule
   - Productivity

6.1.7. Advantages

- Reduction in cost or budget and managing the workloads and teams
- Reduction in overall time to produce the product
- It helps to identify the particular area for improvising and also to increase the product quality
- It helps to determine the complexity of the code and to test the code with resources
- It helps in providing effective planning, controlling and managing of the entire product

6.1.8. Disadvantages

- It is expensive and difficult to implement the metrics in some cases
- Performance of the entire team or an individual from the team can't be determined. Only the performance of the product is determined
- Sometimes the quality of the product is not met with the expectation
- It leads to measure the unwanted data which is wastage of time
- Measuring the incorrect data leads to make wrong decision making

6.2. People Metrics and Process Metrics
6.3. Halstead's Software Metrics
6.4. Cyclomatic Complexity
6.5. Functional Point (FP) Analysis
6.6. Lines of Code

7. TESTING VERIFICATION AND VALIDATION
8. PRACTICE QUESTIONS

© i3Qube Software 2023 Confidential