Toppersnotes

Unleash the topper in you

UGC-NET

COMPUTER SCIENCE AND APPLICATIONS

National Testing Agency (NTA)

PAPER 2 || VOLUME – 2

SIERRA INNOVATIONS

UGC NET PAPER – 2

COMPUTER SCIENCE AND APPLICATIONS

VI

UNIT

Software Engineering

Software Process Models

Software process model and requirements :-

- **Software Engineering** is a specialized area within **Computer Science** that deals with the **systematic development, operation, and maintenance** of software.
- It involves **engineering principles** applied to software development to ensure the final product is:
    - Reliable
    - Efficient
    - Cost-effective
    - Maintainable
    - Delivered on time
- The development of a large software is a group activity or say software engineering activity.
- The components one writes may be modified by others; it may be used by others to construct different versions of system.
- software engineering has been defined by many authours differently. some of these definitions are:

1. As per the IEEE software engineering standards, software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.
2. Software engineering is a discipline whose aim is to produce error free software that satisfies the user's requirements and can be delivered on time within the budget.
3. Software engineering is an engineering discipline which is concerned with all aspects of software production.
4. Software engineering is a methodological and managerial discipline conerining the systematic production and maintenance of software products that are developed and maintained within anticipated and controlled time and cost limits.

Qualities of Software Product :-

- A software product is considered good or successful if it meets certain quality standards that ensure it is useful, reliable, and efficient for users.
- These qualities are often referred to as software quality attributes or non-functional requirements.
- The qualities are as follows:

1. <u>Functionality</u>:-

- A set of attributes that bear on the existence of a set of functions and their specified properties.
- The functions are those that satisfy stated or implied needs.

2. **Reliability :-**
   - The software should work **without failure** under expected conditions.
   - It should handle errors and **recover gracefully**.
   - Example: A banking app should never crash during money transfer.
3. **Usability :-**
   - The software should be **easy to use**, **understandable**, and have a good **user interface**.
   - Even non-technical users should be able to use it comfortably.
   - Example: Mobile apps like WhatsApp are easy for any user to operate.
4. **Efficiency:-**
   - A set of attributes that bear on the relationship between the level of performance
   - of the software and the amount of resources used, under stated conditions.
     - Time Behaviour
     - Resource Utilisation
     - Efficiency Compliance
5. **Maintainability:-**
   - It should be **easy to fix bugs**, update features, and make improvements.
   - The code should be **well-organized and documented** for future developers.
   - Example: If a tax rate changes, an accounting software should allow quick changes.
6. **Portability :-**
   - A set of attributes that bear on the ability of software to be transferred from one
   - environment to another.
     - Adaptability
     - Installability
     - Co-Existence
     - Replaceability
     - Portability Compliance
7. **Reusability**
   - Parts of the software (like functions, modules) should be usable in other projects.
   - Example: A login module can be reused in many different websites.

SDLC Models:-

Following are the SDLC Models.

1. Watefall
2. Prototype Model
3. V Model
4. RAD Model
5. Spiral Model

1. **Watefall Model :-**

   The **Waterfall Model** is the **earliest and simplest software development life cycle (SDLC) model**. It is a **linear sequential approach**, where each phase must be **completed fully** before the **next phase begins**.

Model does not work smoothly if there are some issues left at previous phase.

It's linear sequential behaviour do not allow to go back and redo some task.

This model is most appropriate where-

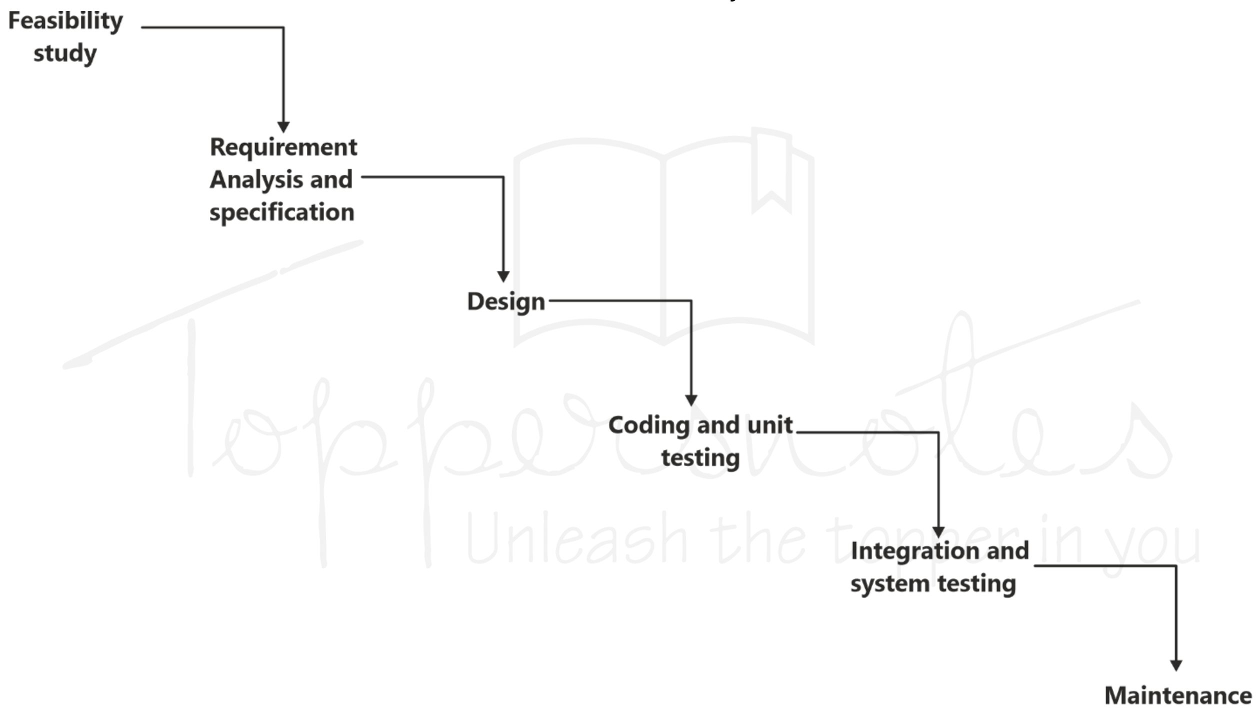
- Requirement are very well documented and fixed
- Product definition is stable.
- The project is short in duration.
- Technology is well known and understood.

The process flows **downwards like a waterfall**, hence the name.

<u>No Overlapping of Phases</u> :- In waterfall model phases are not overlap i.e. each phases are individually done.

<u>Considered a Theoretical Model</u> :- Waterfall model cannot be used in actual software development project. It is considered as theoretical way of developing software.

<u>Basis of All Other Models</u> :- All other models are essentially derived from waterfall model.


Feasibility study
Requirement Analysis and specification
Design
Coding and unit testing
Integration and system testing
Maintenance

1. **<u>Feasibility Study</u>**
   - This phase checks whether the project is **technically and financially possible** to develop or not.
   - It evaluates **cost, resources, and time** before starting.
2. **<u>Requirement Analysis and Specification</u>**
   - In this phase, the development team collects all the requirements from the client or stakeholders.
   - The main aim of this phase is to understand the exact requirements of the customer and to document them properly.
   - During this activity, the user requirement is systematically organize into a software requirement specification document (SRS).
3. **<u>Design</u>**
   - The goal of this phase is to convert the requirements into a complete software architecture, ready for coding.

- The software architecture is created in the Design Phase using the information in the SRS document.
- The system's architecture is planned here.
  - **High-Level Design (HLD):** System overview, modules
  - **Low-Level Design (LLD):** Internal logic of modules, data structure, etc.

4. **Coding and Unit Testing**
   - Developers write code based on the design and follow coding standards and guidelines set by the organization.
   - Each program module is tested individually (**Unit Testing**) to check for bugs.
   - The goal is to check whether the module works correctly and performs the intended function.
   - his helps identify bugs early before modules are integrated.
5. **Integration and System Testing**
   - All modules are combined and tested as a complete system.
   - Integration of different modules is done in a **planned and step-by-step manner**.
   - It is carried out **incrementally**, where in each step:
     - A **set of pre-planned modules** is added to the partially integrated system.
     - The combined system is tested after each integration step.
   - This continues until **all modules** are integrated and tested successfully.
   - Once all modules are integrated, the **entire system** is tested to verify that it meets all functional and performance requirements.
   - **System Testing** ensures the software meets functional and performance requirements.

System testing includes the following **three types of testing**:

i. **α (Alpha) Testing**:
   - Performed by the **development team**.
   - Done internally to identify bugs before public release.

ii. **β (Beta) Testing**:
   - Performed by a **selected group of end users** (friendly customers).
   - Used to collect feedback and detect issues in a real-world environment.

iii. **Acceptance Testing**:
   - Performed by the **customer** after delivery.
   - Determines whether the software should be **accepted or rejected**.

6. **Maintenance** After the software is deployed, it is maintained for:
   - Bug fixing
   - Performance improvement
   - Enhancements based on user feedback
   - Once a software product is developed and delivered to the customer, it enters the maintenance phase. Maintenance means keeping the software running properly, improving it, or adapting it to new needs.

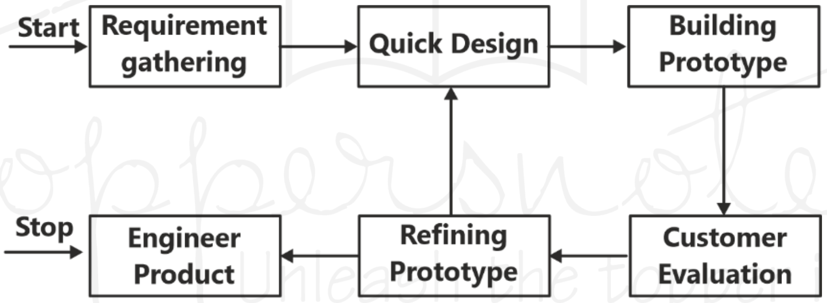
Types of Software Maintenance:

i. **Corrective Maintenance :-**
This is done to fix errors or bugs that were not detected during development or testing.
These may include logic errors, coding mistakes, or runtime problems that users report after deployment.

ii. **Perfective Maintenance:-**
This is done to enhance or improve the software's performance or features based on user feedback or new requirements.
It may involve adding new functionalities, improving UI/UX, or optimizing existing code.

iii. **Adaptive Maintenance**
This type involves modifying the software to adapt to new environments or technologies. For example, making the software compatible with a new operating system, hardware, or database.

2. **Prototype Model :-**
Prototyping Model is a software development model where a **working prototype** is developed **before the actual system** is built.
The **Prototyping Model** is a software development approach used when the exact requirements of the customer are not clearly known at the beginning of the project. This model is especially effective when user interaction is high or when the system's requirements are expected to evolve with feedback.



Phases of the Prototyping Model

1. **Requirement Gathering** Only initial and known requirements are collected to build the prototype.
2. **Quick Design** A preliminary design is created for the prototype with essential features and user interfaces.
3. **Build Prototype** A working model is developed with limited functionality.
   It may not include all internal logic or final architecture.
4. **Customer Evaluation** The prototype is shared with users to get feedback.
   Feedback is collected based on how well it meets the expectations.
5. **Refinining Prototype** Based on feedback, the prototype is improved and requirements are refined.
   Multiple iterations may occur until customer satisfaction is achieved.
6. **Engineer Product** After several iterations and final approval, the real system is developed.

Advantages:

- Early visibility of the system for the customer.
- Faster identification of missing or unclear requirements.
- Reduces the chances of system failure.
- High user involvement ensures better satisfaction.

3. **Spiral Model :-**

   The spiral model is similar to the incremental model, with more emphses placed on risk analysis. The spiral model has four phases: planning, risk analysis, engineering and evaluation.

   A software project repeatedly passes through these phase in iterations. In the baselines spirial , starting in the planning phases, requirements are gathered and risk is assessed. Each subsequent spirals builds on the baseline spiral.

Phases of Spiral Model:

1. **Planning Phase:**
   - Requirements are collected from the stakeholders.
   - Objectives, constraints, and alternatives are identified.
   - A preliminary project plan is created.
2. **Risk Analysis Phase:**
   - Potential risks (technical, management, cost-related) are identified.
   - Alternative solutions are analyzed to mitigate these risks.
   - Prototyping may be used to clarify uncertainties.
3. **Engineering Phase:**
   - Actual development and testing of the software occur.
   - Coding, unit testing, integration, and validation are done.
   - A new version of the product is created.
4. **Evaluation Phase:**
   - The current version is reviewed by stakeholders.
   - Feedback is collected to determine changes for the next iteration.
   - Planning for the next spiral begins.

Advantages :

- High amount of risk analysis.
- Good for large and mission critical projects.
- Software is producedd early in the software life cycle.

Disadvantage:

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

4. **RAD Model :-**

   RAD is a linear sequential software development process model that emphasizes an extremely short development cycle using a component-based construction approach.

   RAD is a way to create software quickly. It focuses on making a working program in a short time by using ready-made tools and reusable code. If the project's goals are clear and the scope is limited, RAD helps developers build a complete software system super fast.

   RAD focuses on reusing existing codes, leveraging tools like software development kits and graphical user interfaces (GUIs), and prioritizing visual development over extensive coding.

It is particularly effective when requirements are well-defined, and the project scope is constrained. Unlike traditional models that follow a rigid, sequential approach (e.g., Waterfall), RAD compresses the analysis, design, build, and test phases into short, iterative cycles.

The RAD approach focuses on:

- **Reusability**: Leveraging existing code and components to reduce development time.
- **Prototyping**: Creating iterative prototypes that are tested and refined based on user feedback.
- **User Involvement**: Engaging end-users throughout the process to ensure the product aligns with their needs.
- **Visual Development**: Prioritizing GUI development over complex, original coding, often using visual programming tools.
- **Object-Oriented Programming**: languages like C++ and Java, which support modularity and reuse.

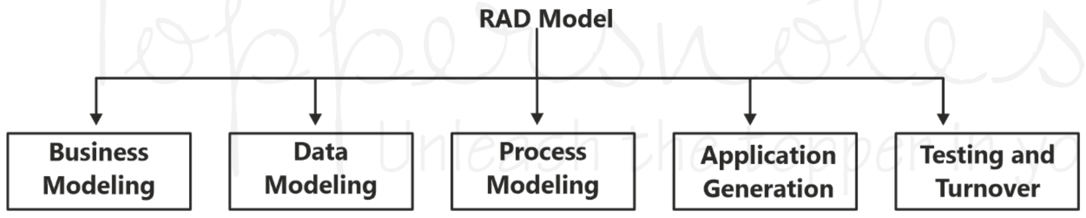
Key Features of RAD

RAD is designed to produce high-quality systems faster through:

- Gathering requirements using workshops or focus groups
- Prototyping and early reiterative user testing of designs
- The re-use of software components
- A rigidly paced schedule that defers design improvements to the next product version
- Less formality in reviews and other team communication

RAD Model Phases

The RAD model is structured into five distinct phases:



1. **Business Modeling**:
   The information flow among business functions is defined by answering questions like what information drives the business process, what information is generated, who generates it, where does the information go, who process it and so on.
2. **Data Modeling**:
   The information collected from business modeling is refined into a set of data objects (entities) that are needed to support the business. The attributes (character of each entity) are identified and the relation between these data objects (entities) is defined.
3. **Process Modeling**:
   The data object defined in the data modeling phase are transformed to achieve the information flow necessary to implement a business function. Processing descriptions are created for adding, modifying, deleting or retrieving a data object.
4. **Application Generation**:
   Automated tools are used to facilitate construction of the software; even they use the 4th GL techniques.

5. **Testing and Turnover**:
   Many of the programming components have already been tested since RAD emphasis reuse. This reduces overall testing time. But new components must be tested and all interfaces must be fully exercised.

Software Projects Management

Software Project Management (SPM) is the discipline of planning, organizing, leading, and controlling software projects. It ensures that software is developed on time, within budget, and meets quality standards.

Mainly there are three needs of SPM:

1. Time :- Complete the project within estimated time.
2. Cost :- Complete the project within estimated cost.
3. Quality :- Deliver high-quality software as per client requirements.

List of activities done:-

1. Project Planning and tracking
2. Project resource management
3. Scope Management
4. Estimation management
5. Project risk management
6. Scheduling management
7. Project communication management
8. Configuration management

Project Management Tools:-

- Project Management tools and techniques are methods, software, or frameworks used by project managers to plan, organize, monitor, and control software projects effectively.

1. **Gantt Chart :-** A Gantt Chart is a bar chart that shows the start and end dates of project tasks along a timeline. It is used when you want to see the entire landscope of projects. It helps us to view which task are dependent on another and which event is coming up.
2. **PERT (Program Evaluation and Review Technique) :-** PERT is used for estimating time required for activities when there is uncertainty.
   A network-based planning technique that focuses on event-based scheduling.
   It represent a network diagram concerning the number of nodes.

Time Estimates:

- **Optimistic Time (O)** – Minimum time if everything goes well
- **Most Likely Time (M)** – Normal time
- **Pessimistic Time (P)** – Maximum time if delays happen

$$TE = \frac{O + 4M + P}{6}$$

3. **CPM (Critical Path Method) :-**
   A network technique to find the longest path (critical path) in a project schedule.
   The critical path Identifies tasks that directly affect project completion time.
   It used to categorize activities which are required to complete a task and relationship between each activity involved to complete that task.

4. **Work Breakdown Structure (WBS)**
   WBS is a tree-structured breakdown of the entire project into smaller, manageable components or tasks.

Software Requirements

Requirement Engineering Process

- **Requirement Engineering (RE)** is the process of **identifying, documenting, analyzing, validating, and managing the needs and expectations** of the users for a software system.
- It ensures that software is **developed correctly and meets the actual needs** of users, clients, and stakeholders.
- Phases of Requirement Engineering Process:

1. **Requirement Elicitation :** Requirements Elicitation is the **first and one of the most important phases** in the requirement engineering process. It involves **gathering information and understanding** the user's problems, goals, and expectations from the system to be developed.
   At the **start of any software project**, requirements are often **unclear, incomplete, or ambiguous**. The job of the analyst is to **explore the domain, communicate with stakeholders**, and acquire the **necessary domain-specific knowledge** to understand the system properly.
   Activities Involved:
   - **Identify stakeholders** (users, customers, developers).
   - **Conduct interviews**, surveys, group discussions.
   - Observe existing systems and workflows.
   - Use **prototypes, scenarios, or use-case diagrams** to help users express their needs
2. **Requirement Analysis :-** After the **elicitation phase**, the next step in the Software Requirement Engineering process is **Requirement Analysis**. This step involves analyzing the gathered requirements to get a **clear and structured understanding** of what exactly the customer needs, and what the software product must deliver.
   Categorize requirements as:
   - Functional
   - Non-functional
   - Domain-specific

   Check feasibility: Can it be implemented with available resources?
   Check consistency: No two requirements should conflict.

Key Activities in Requirement Analysis:

i. **Validation**
   Ensures that each requirement is **relevant to the project's goals** and aligns with business objectives.

ii. **Consistency Checking**
   Verifies that there are **no conflicting requirements** and that all requirements work together logically.

iii. **Feasibility Study**

Confirms that the required **technology, tools, time, and budget** are available to implement the system successfully.

3. **Requirement Documentation :**

Requirement Documentation is the **third phase** of the Requirement Engineering Process. After gathering and analyzing the requirements, the next step is to **formally document** them in a structured and organized format.

This is usually done by creating an **SRS (Software Requirement Specification)** document, which serves as a **blueprint** for software development.

- **SRS** stands for **Software Requirements Specification**.
- It is a **formal document** that provides a **complete description of the software system** to be developed.
- SRS serves as a **bridge between the customer and development team**, ensuring both have a clear, shared understanding of what the software should do.
- SRS could be written by the client or developer of a system.

Purpose of SRS:

- To **clearly define functional and non-functional requirements**
- To serve as a **contract** between clients and developers
- To help in **project planning, design, coding, and testing**
- To ensure **customer satisfaction** by preventing misunderstandings

Feature of Good SRS:-

1. Correctness
2. Completeness
3. Consistency
4. Unambiguousness
5. Modification
6. Ranking for importance and stability
7. Verifiability
8. Traceability
9. Design Independence
10. Testability
11. Understandable by the customer
12. Right level of abstraction

4. **Requirement Review :-**

Requirement Review is a **manual inspection and evaluation process** that ensures the **correctness, clarity, and completeness** of the requirement specifications. It involves collaboration between the **client and the development team** to check the documented requirements before software development begins.

Types of Requirement Reviews:

1. **Informal Review**
   - Conducted without a fixed schedule or structured format.
   - The contractor / development team meets with as many stakeholders as possible.
   - Discussions are free-flowing and help in **early identification** of major gaps or misunderstandings.
   - No formal documentation is required, but **important notes are recorded**.
2. **Formal Review**
   - **Well-planned and structured** meeting between clients and developers.
   - Involves a **formal review team** with defined roles and responsibilities.
   - Each requirement is examined for **consistency, feasibility, completeness, and clarity**.
   - **Formal review reports** are created to document:
     - Conflicts
     - Contradictions
     - Omissions
     - Suggestions for improvement

5. **Requirement Management :-**

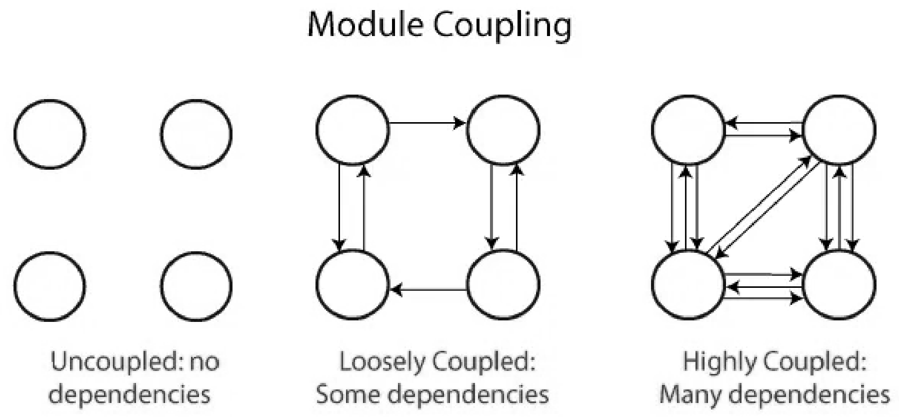
Requirement Management is the **continuous process** of documenting, analyzing, tracking, prioritizing, and updating the requirements throughout the **software development lifecycle**.

Software Design

Coupling

- Coupling means how **strongly two modules are connected** to each other. It shows the **level of dependency** between modules in a program. If two modules share a lot of data, they are **tightly coupled**, which is not good. We aim for **loose coupling**, where modules are more **independent**. This makes the software easier to **change, test, and reuse**. Lower coupling leads to better **modularity** and system quality.

  Good software will have low coupling.


Module Coupling
Uncoupled: no dependencies
Loosely Coupled: Some dependencies
Highly Coupled: Many dependencies

Types of Coupling :-

- Coupling tells us how much one module depends on another in a software system. Less dependency (loose coupling) is better for flexibility and easier maintenance.

1. **Content Coupling**
   Content coupling occurs when a module modifies the internal state or behavior of another module.. This might involve jumping into the middle of another module's code or altering its internal data. It creates strong dependencies, making maintenance very difficult. This is considered the worst form of coupling and should be avoided in good software design.
2. **Common Coupling**
   Common coupling happens when two or more modules share global data or common storage. Changes made to the shared data by one module can unexpectedly affect the others. This creates a high level of dependency and makes debugging and maintaining the system more complex. It is generally discouraged in structured programming.
3. **External Coupling**
   External coupling exists when modules depend on external data formats, communication protocols, or device interfaces. A change in these external elements requires modifications in all dependent modules. Although better than content and common coupling, it still introduces risks during integration and future changes in external systems.
4. **Control Coupling**
   Control coupling occurs when one module influences the behavior of another by passing control information like flags or command variables. This causes one module to depend on the logic of another, reducing independence. It also makes testing and understanding the module more difficult due to the tight logical link between them.
5. **Stamp Coupling**
   Stamp coupling (also known as data structure coupling) happens when modules share a complex data structure but only use part of it. This exposes unnecessary details and causes unwanted dependencies. It is preferable to pass only the necessary elements instead of the whole structure to improve modularity.
6. **Data Coupling**
   Data coupling is when modules share data strictly through parameters, and each parameter is a simple, atomic data item like an integer, character, or float. This type of coupling is considered good because it ensures modules remain independent and focused. It also improves reusability and simplifies testing.
7. **Message Coupling**
   Message coupling is the loosest and most desirable form of coupling. Here, modules communicate only through message passing without depending on internal details or shared structures. It is common in object-oriented programming and promotes maximum modularity and low interdependence, making systems easier to maintain and scale.

Cohesion

- Cohesion refers to the relationship among elements within a module. If a module performs only one specific task, it is considered more cohesive(High cohesion) and is less error-prone than moudles that perform multiple tasks . High cohesion indicates that the components inside a module are closely related, often sharing communication or resources. This leads to better modularity, as tasks are handled internally with minimal dependency on other modules. A good software design ensures strong cohesion within modules and loose coupling between them.

Types of Cohesion

1. **Functional Cohesion**

   A module is functionally cohesive when every part of it contributes to a single, well-defined task. All elements work together to achieve one specific functionality. For example, a payroll module that handles salary calculation, deductions, and payslip generation is functionally cohesive. This is the strongest and most desirable form of cohesion. It ensures maximum modularity, minimal errors, and better maintainability of the software system.

2. **Sequential Cohesion**

   Sequential cohesion exists when the output of one operation in a module serves as the input for the next. This creates a strong connection between the components. For instance, reading data from a file and then processing that data in the same module is an example of sequential cohesion. This improves the clarity and flow of the module. It is stronger than communicational cohesion and is a good practice in structured programming.

3. **Communicational Cohesion**

   This type of cohesion is present when all components of a module operate on the same data or share the same input or output. They may perform different tasks, but they are related by the data they use. For example, all functions working on the same database record form a communicationally cohesive module. It ensures a better structure than procedural or temporal cohesion. Modules with communicational cohesion are more understandable and maintainable.

4. **Procedural Cohesion**

   Procedural cohesion is found in a module when all its operations follow a specific sequence of execution to perform an objective. Even if the individual operations are unrelated, they are carried out one after another in a fixed order. For example, checking file permission and then opening the file in a single module indicates procedural cohesion. The sequence binds them together, improving control flow but not necessarily the logical relation. This is moderately good cohesion.

5. **Temporal Cohesion**

   When a module contains tasks that are executed around the same time during program execution, it shows temporal cohesion. This usually happens when tasks are grouped because they are triggered by the same event. For instance, closing files, logging errors, and notifying users during an exception are all placed in an exception handler module. These activities are related by time, not by their function or data. Temporal cohesion is better than logical, but still not very strong.

6. **Logical Cohesion**

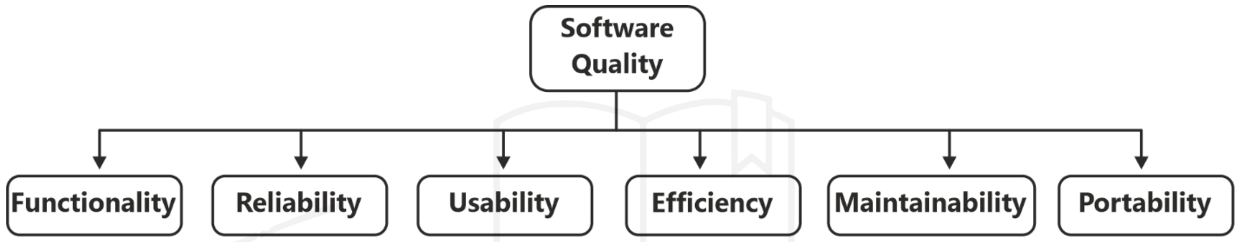
   Logical cohesion exists when elements in a module perform similar kinds of operations, even if they are unrelated in function. The grouping is based on a logical category, such as input or output handling. For example, handling both keyboard and mouse input in the same module represents logical cohesion. Although related in logic, their internal functionalities differ. This type of cohesion is better than coincidental but still not ideal for high-quality design.

7. **Coincidental Cohesion**

A module is said to have coincidental cohesion when it performs a set of unrelated or loosely related tasks. These tasks are not connected by data flow or control flow, and they do not contribute to a single clear purpose. It usually results from poor design where functions are randomly grouped. Such modules are hard to understand, test, and maintain. The lack of relationship makes the code error-prone and confusing. This is considered the worst form of cohesion.

Software quality

- Software Quality Attributes are the characteristics or properties that determine the overall quality of a software product. These attributes help in evaluating how well the software performs, how easy it is to use and maintain, and how robust and reliable it is.
- These attributes are also known as non-functional requirements and play a crucial role in software architecture and engineering.



- Here are the main Software Quality Attributes:

1. **Functionality :-** The Capability to provide functions which meet stated and implied needs when the software is used.
   It allows us to draw conclusions about how well the software delivers the required features and services expected by the user or system.
   It is used for assessing, controlling, and predicting the extent to which a software product satisfies its functional requirements.
   it include:
   - suitability
   - Accuracy
   - Interoperability
   - Security
2. **Reliability :-** The capability to maintain a specified level of performance.
   It allows us to draw conclusions about how well the software performs consistently and maintains the required level of system performance under specific conditions, even in the presence of faults or unexpected situations.
   it include:
   - Maturity
   - Fault Tolerance
   - Recoverability

3. **<u>Usability</u> :-** The capability to be understood, learned, and used.

   It refers to how easy and comfortable it is for users to interact with the software, especially first-time or non-technical users. High usability ensures that the software is user-friendly and minimizes the learning curve.

   Usability in software quality focuses on making the software user-friendly, helping users accomplish tasks with ease and confidence, thereby directly impacting the overall quality and acceptability of the product.

   Key Attributes of Usability:

   - Understandability
   - Learnability
   - Operability
   - Attractiveness

4. **<u>Efficiency</u> :-** The capability to provide appropriate performance relative to the amount of resources used.

   It focuses on how well the software performs when used under certain constraints and conditions, such as high workload, limited hardware, or real-time responses.

   Main Sub-Attributes of Efficiency:

   - Time Behavior : Refers to how quickly the system responds to user actions or events.
   - Resource Utilization: Describes the amount of resources (like memory, CPU, disk space, and bandwidth) the software consumes during execution.

5. **<u>Maintainability</u> :-** Maintainability is the capability of a software product to be modified easily and effectively, whether the purpose is to:

   - Fix bugs (corrections),
   - Improve performance or features (enhancements),
   - Adapt to a new environment (platform or OS changes), or
   - Meet updated requirements.
   - It ensures that the software can be safely and quickly changed without introducing new defects.

6. **<u>Portability</u> :-** Portability is the capability of the software to be work properly in different environment without requiring any changes to the software code**.**

   In simple terms, portable software can "move" and "run" across systems easily.

   <u>Key Sub-Attributes of Portability</u>:

   - Adaptability
   - Installability
   - Replaceability

Estimation and Scheduling of Software Projects

<u>**COCOMO (Constructive Cost Model) :-**</u>

- The **Constructive Cost Model (COCOMO)** is an **algorithmic software cost estimation model** developed by **Barry W. Boehm** in 1981.
- It is used to estimate:
  - **Effort** (in person-months)
  - **Development time** (schedule)
  - **Project cost**
- **COCOMO (Constructive Cost Model)** is a **cost estimation model** for software projects.
- It is often used as a systematic process for **reliably predicting** various parameters associated with software development, such as:
  - **Size** of the software (in KLOC)
  - **Effort** required (in person-months)
  - **Development Cost**
  - **Time** to complete the project
  - **Staffing needs**
  - **Quality** and **schedule** expectations

<u>**Three Levels of COCOMO:-**</u>

Level	**Description**	**Accuracy**
1. Basic COCOMO	Gives a **quick and rough estimate** based only on software size (KLOC).	Low
2. Intermediate COCOMO	Considers **Cost Drivers** (project attributes like complexity, experience, tools).	Medium
3. Detailed COCOMO	Includes everything from Intermediate + factors for Low **each phase** of development.	High

1. <u>**Basic COCOMO -**</u>

   Basic COCOMO (Constructive Cost Model) is a simple and widely used **software cost estimation model** developed by **Barry W. Boehm**.

   It estimates the **effort, cost, and development time** based on the size of the software in **KLOC (Thousands of Lines of Code)**.

   It is best suited for **early-phase, quick estimates** and assumes a traditional waterfall development process.

   COCOMO applies to three classes of software projects:

i. Organic Projects:

"small" teams with "good" experience working with "less than rigid" requirements.

ii. Semi-Detached Projects:

"medium" teams with mixed experience working with a mix of rigid and less than rigid requirements.

iii. Embedded Projects:

developed within a set of "tight" constraints. It is also combination of organic and semidetached projects. (hardware, software, operational, ...).

Basic COCOMO Equations:

1. Effort Applied (E):
   $E = a \times (\text{KLOC})^b$ [person-months]
2. Development Time (D):
   $D = c \times (E)^d$ [months]
3. People Required (P):
   $P = \frac{E}{D}$ [personnel]

Where:

- **KLOC** = Estimated number of lines of code (in thousands)
- **a, b, c, d** = Constants depending on project type (Organic, Semi-Detached, Embedded)

2. Intermediate COCOMO:-

Intermediate COCOMO is an extension of the **Basic COCOMO** model.

It improves estimation accuracy by including not only the size of the software , but also a set of **"Cost Drivers"** that affect the effort required.

These cost drivers are based on **subjective assessments** of various **product, hardware, personnel, and project attributes.**

Effort (E) = a × $(\text{KLOC})^b$ × EAF

Where:

- **EAF** = Effort Adjustment Factor (based on cost drivers)
- **a, b** = Constants (vary based on project type)
- **KLOC** = Size of the code (in thousands of lines)

Intermediate COCOMO uses **4 categories of cost drivers**, each with multiple factors.

1. Product Attributes:
2. Hardware Attributes:
3. Personnel Attributes:
4. Project Attributes:

Intermediate COCOMO provides **more realistic and accurate estimates** than Basic COCOMO.

3. <u>Detailed COCOMO:-</u>

Detailed COCOMO is the most comprehensive level of the Constructive Cost Model. It builds upon the **Intermediate COCOMO** model by including **phase-wise effort estimation** throughout the **software development life cycle**.

This model not only considers the **program size** and **cost drivers**, but also evaluates the **impact of each cost driver on each development phase** like planning, design, coding, and testing.

Effort (E) = a × $(\text{KLOC})^b$ × EAF_i

Where:

- **KLOC** = Size of software (in thousands of lines of code)
- **EAF<sub>i</sub>** = Effort Adjustment Factor for each phase
- **a, b** = Constants based on project type (Organic, Semi-Detached, Embedded)