# SOFTWARE ENGINEERING FAT NOTES

## Contents



# MODULE 1

**An Introduction to Software Engineering**

Software engineering encompasses computer programs and their associated documentation, including requirements, design models, and user manuals. Software products may be developed for specific customers or for a general market.

**Types of Software Products**

- **Generic**: Developed for a wide range of customers (e.g., PC software like Excel or Word).
- **Bespoke (Custom)**: Developed for a single customer according to their specifications.

**Software Engineering**

Software engineering is an engineering discipline focused on all aspects of software development. Software engineers should adopt a systematic and organized approach, utilizing appropriate tools and techniques to develop software products. This field concerns theories, methods, and tools for professional software development.

**Characteristics of Projects**

A task is considered more ‘project-like’ if it exhibits the following characteristics:

- **Non-routine**
- **Planning involved**
- **Aiming at a specific target**
- **Work carried out for a customer**
- **Composed of several different phases**
- **Constrained by time and resources**
- **Large and/or complex**

**Software Characteristics**

- **Logical Nature**: Software is logical rather than physical.
- **Developed, Not Manufactured**: Software is engineered, not manufactured.

- **Durability**: Software doesn't wear out.
- **Custom Built**: Software continues to be tailored to specific needs.

**Software Process**

The software process consists of a set of activities required to develop a software system. The generic activities in all software processes are:

1. **Specification**: What the system should do and its development constraints.
2. **Development**: Production of the software system.
3. **Validation**: Checking that the software meets customer requirements.
4. **Evolution**: Changing the software in response to evolving demands.

**Costs of Software Engineering**

- Approximately **60%** of costs are development costs, while **40%** are testing costs.
- Costs vary based on the type of system being developed and the requirements related to attributes such as performance and reliability.
- Distribution of costs depends on the development model used.

**Categorizing Projects**

Distinguishing different types of projects is essential as varying tasks require different approaches:

- **Information Systems vs. Embedded Systems**
- **Product-based vs. Service-based**
- **Generic vs. Domain-specific**
- **Mission Critical vs. Life Critical**

**Attributes of Good Software**

Good software should deliver the required functionality and performance, and be:

- **Maintainable**: Must evolve to meet changing needs.
- **Dependable**: Trustworthy and reliable.
- **Efficient**: Optimal use of system resources.
- **Acceptable**: Understandable, usable, and compatible with other systems.

**Software Applications**

Different types of software applications include:

- **System Software**: Programs that service other programs.
- **Real-time Software**: Monitors/controls real-world events.
- **Business Software**: Used for business processing.
- **Engineering/Scientific Software**
- **Embedded Software**
- **Web-based Software**

**Examples of Software Applications**

- **Operating Systems**: iOS / Mac OS v14.5
- **Prediction Systems**: Stock market prediction
- **Monitoring Systems**: Weather monitoring system
- **Management Software**: Inventory management, resource management systems
- **E-commerce Applications**
- **Banking Applications**
- **Health Monitoring Software**

**Frequently Asked Questions about Software Engineering**

| Question | Answer |
|---|---|
| **What are the key challenges facing software engineering?** | Coping with increasing diversity, demands for reduced delivery times, and developing trustworthy software. |
| **What are the costs of software engineering?** | Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs. |
| **What are the best software engineering techniques and methods?** | Different techniques are suitable for different types of systems. For example, games should be developed using prototypes, while safety-critical systems require complete specifications. |
| **What differences has the web made to software engineering?** | The web has enabled the availability of software services and the development of highly distributed service-based systems, leading to advances in programming languages and software reuse. |

**Importance of Software Engineering**

As society increasingly relies on advanced software systems, the need for reliable and trustworthy systems that are produced economically and quickly becomes critical. Using software engineering methods is usually cheaper in the long run than treating software development as a personal programming project, as most costs arise from changes after the software has gone live.

**Software Lifecycle Phases**

The software lifecycle includes the following phases:

1. Requirement Analysis
2. Design
3. Development / Implementation
4. Testing
5. Deployment
6. Maintenance
7. Evolution

**Software Process Model Types**

Software process models include:

- **Linear Sequential Model**
- **Prototyping Model**

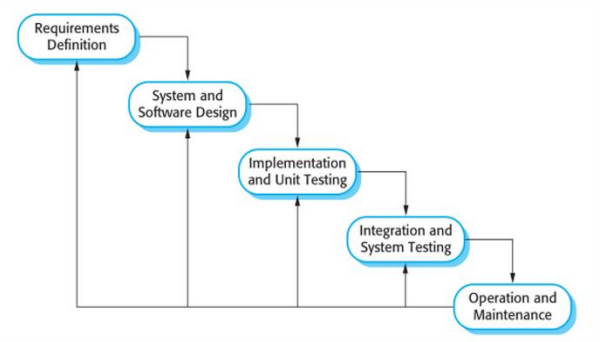
- **RAD Model**
- **Evolutionary Software Process Models**
    - Incremental
    - Spiral
    - Component-Based Development
    - Agile / Dynamic Models

**The Waterfall Model**

The Waterfall model is the classic lifecycle model that is widely known and understood. It is also referred to as the classical lifecycle model or sequential process model, introduced by Royce in 1970.

**Waterfall Model Phases**

1. Requirements analysis and definition
2. System and software design
3. Implementation and unit testing
4. Integration and system testing
5. Operation and maintenance

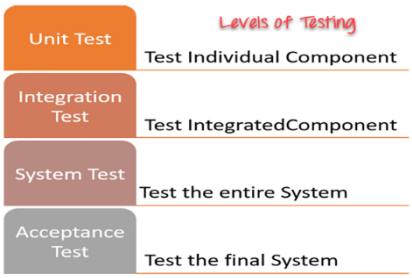



**Advantages**

1. Easy to understand and implement.
2. Widely used and known.
3. Reinforces good habits: define-before-design, design-before-code.
4. Identifies deliverables and milestones early.
5. Document-driven.
6. Works well on mature products and weak teams.

**Disadvantages**

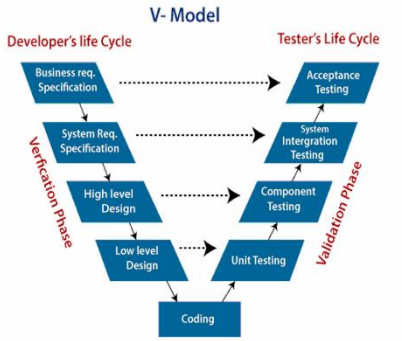
1. Idealized; doesn't match reality well.
2. Unrealistic expectations for accurate requirements early in the project.
3. Software is delivered late, delaying the discovery of serious errors.
4. Difficult and expensive to make changes to documents later.
5. Significant administrative overhead; costly for small teams and projects.




**Real-world Examples of the Waterfall Model**

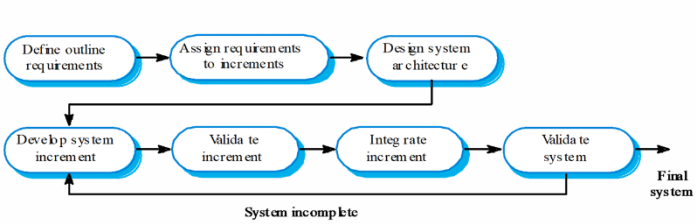
- **Consumer Goods Industry**: P&G implemented SAP ERP using the Waterfall Model to streamline global operations, integrating supply chain, manufacturing, and distribution processes.
- **Food and Beverage Industry**: Hershey's used the Waterfall Model for SAP ERP to enhance supply chain and production processes, improving inventory management and demand forecasting.

## The V-Model



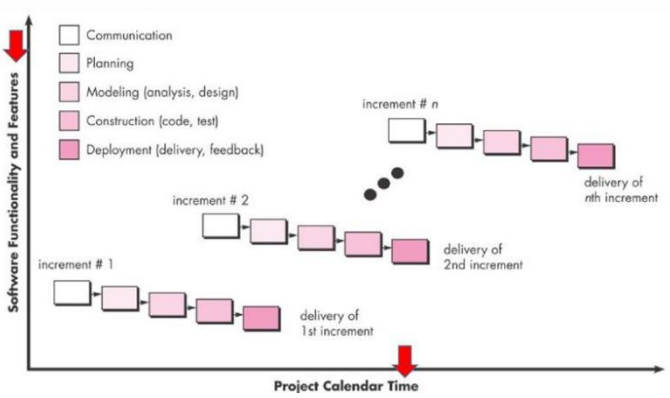

The V-Model can be viewed as an extension of the Waterfall Model, where the left side of the V focuses on refining requirements and the right side on validating the models through testing.

## Incremental Model



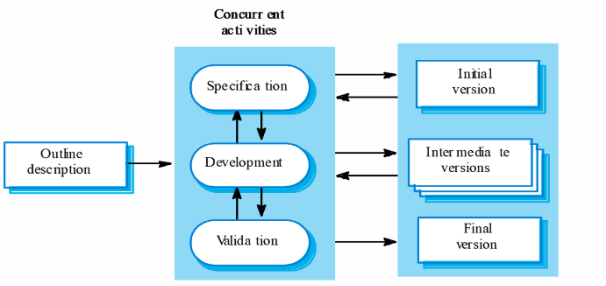

In the Incremental Model, development and delivery are broken down into increments, each delivering part of the required functionality. User requirements are prioritized, and the highest priority requirements are included in early increments.

## Advantages of Incremental Development



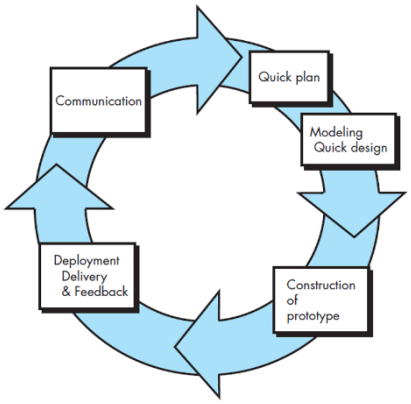

- Customer value is delivered with each increment.
- Early increments act as prototypes for later requirements.
- Lower risk of overall project failure.
- High-priority system services receive more testing.

## Evolutionary Development




Evolutionary development includes exploratory development and prototyping, aiming to work with customers to evolve a final system from an initial outline specification.

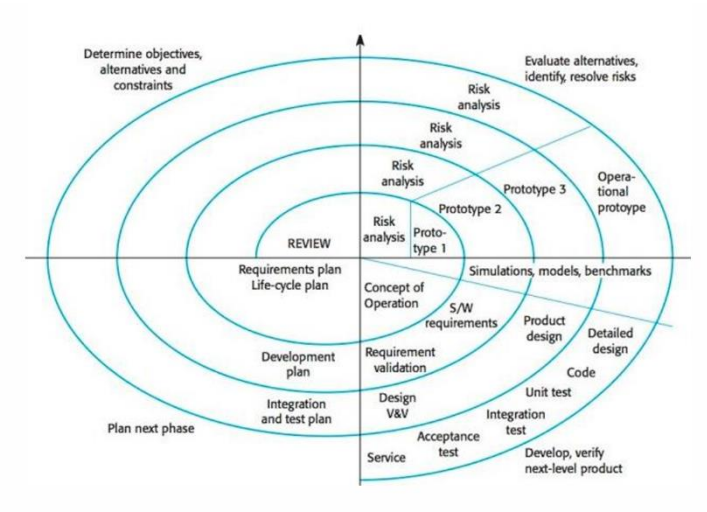
## Prototyping Model




- Customer is not clear with the idea
- Throwaway model
- Good for technical and requirement risks
- Increase in cost of development
- Costly
- No early lock-on requirements
- Reusability

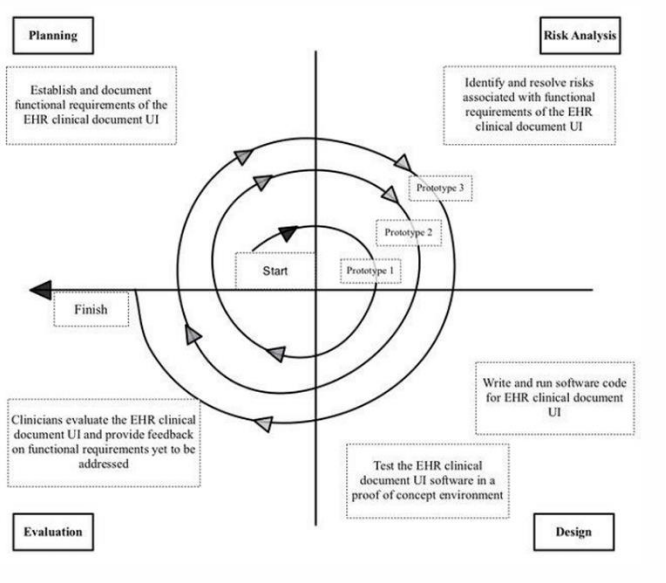
**Spiral Development**

The Spiral Model represents a process as a spiral with each loop representing a phase. It combines features of both the prototyping model and the waterfall model, with explicit risk assessment throughout the process.




**Steps of the Spiral Model**

1. Define the problem in detail.
2. Create a preliminary design.
3. Construct a prototype.
4. Evaluate the prototype and define requirements for the next iteration.
5. Iterate until the customer is satisfied with the prototype.
6. Construct the final system.
7. Conduct thorough evaluation and testing.




**Advantages of the Spiral Model**

- Risk reduction.
- Flexibility.
- Continuous client feedback.

**Challenges**

- Complexity.
- Expense.
- Time-consuming risk analysis.

**Component-Based Software Engineering**

This approach emphasizes systematic reuse, integrating systems from existing components or commercial-off-the-shelf (COTS) systems. It includes stages like component analysis, requirements modification, system design with reuse, and development/integration.

**1. Waterfall Model**

The **Waterfall Model** is a traditional linear approach where progress flows in one direction—downwards, like a waterfall. Each phase depends on the deliverables of the previous one, making it ideal for projects with clear, fixed requirements.

**Phases:**

1. **Requirement Analysis:** All project requirements are gathered and documented.
2. **System Design:** High-level and detailed design specifications are prepared.
3. **Implementation:** Developers write the code based on design specifications.
4. **Integration and Testing:** The system is tested for errors and bugs.

5. **Deployment:** The system is delivered to the customer.
6. **Maintenance:** Fixes and updates are made as needed post-deployment.

**Advantages:**

- Well-documented process and clear deliverables.
- Easy to manage due to its structured approach.
- Suitable for projects where requirements are stable.

**Disadvantages:**

- No room for feedback or change once a phase is completed.
- Testing happens late in the lifecycle, increasing the risk of undiscovered issues.
- Not ideal for complex or long-term projects with evolving needs.

**2. V-Model (Verification and Validation Model)**

The **V-Model** enhances the Waterfall model by aligning testing activities with development stages. Each development phase has a corresponding testing phase, ensuring that defects are identified early.

**Phases:**

- **Left side of the V (Development Stages):** Requirement analysis, system design, and coding.
- **Right side of the V (Testing Stages):** Unit testing, integration testing, system testing, and acceptance testing.

**Advantages:**

- Early testing ensures higher quality deliverables.
- Well-structured and straightforward to implement.
- Errors are caught early, reducing costs.

**Disadvantages:**

- Rigid structure, making it unsuitable for iterative processes.
- Assumes all requirements are known upfront.
- Testing relies heavily on documentation and predefined requirements.

**3. Incremental Model**

The **Incremental Model** breaks down development into smaller, manageable increments. Each increment delivers a part of the functionality, eventually combining to form the complete system.

**Process:**

1. Develop a basic version of the product with core functionality.
2. Add features or improvements in subsequent increments.
3. Test each increment before integrating it with the existing system.

**Advantages:**

- Customers can see and use parts of the system early.
- Adaptable to changing requirements.
- Reduces risks by addressing smaller portions at a time.

**Disadvantages:**

- Requires careful planning and coordination between increments.
- May lead to integration issues if increments are not compatible.
- Needs a modular system design for efficient implementation.

**4. Prototyping Model**

In the **Prototyping Model**, a prototype (a working model of the system) is built to help understand user needs better.

**Process:**

1. Collect preliminary requirements.
2. Build a prototype with limited functionality.
3. Allow users to test and provide feedback.
4. Refine the prototype based on feedback.
5. Finalize requirements and develop the actual system.

**Advantages:**

- Helps address unclear or poorly defined requirements.
- Early user involvement ensures a more user-friendly final product.
- Identifies potential risks and usability issues early.

**Disadvantages:**

- Users may become too focused on the prototype and expect the final product to match it exactly.
- Building prototypes can be time-consuming and expensive.
- May lead to excessive iterations, delaying project completion.

**5. Spiral Model**

The **Spiral Model** combines iterative development with a strong focus on risk assessment. It's visualized as a spiral with four quadrants:

**Phases (Each Spiral Cycle):**

1. **Planning:** Define objectives, alternatives, and constraints.
2. **Risk Analysis:** Identify and address risks in the current cycle.
3. **Engineering:** Develop and test the system incrementally.
4. **Evaluation:** Review progress with stakeholders and plan the next iteration.

**Advantages:**

- Excellent for large, complex projects with high risks.
- Flexibility to accommodate changes in requirements.
- Continuous customer involvement and feedback.

**Disadvantages:**

- High costs due to risk analysis and iterative development.

- Requires experienced teams and strong project management.
- Difficult to implement for small projects.

**6. Agile Model**

The **Agile Model** is a highly flexible and iterative approach that emphasizes collaboration, user feedback, and delivering small, functional parts of the system quickly.

**Principles:**

- Break the project into small, manageable pieces (sprints).
- Engage customers and stakeholders throughout development.
- Regularly review and adapt based on feedback.

**Advantages:**

- Highly adaptable to changing requirements.
- Involves end-users continuously, leading to a better final product.
- Delivers functional software quickly, even if not the full system.

**Disadvantages:**

- Requires a high level of coordination and collaboration.
- Scope creep can occur without strict control.
- Difficult to predict costs and timelines for large projects.

**When to Use Each Model**

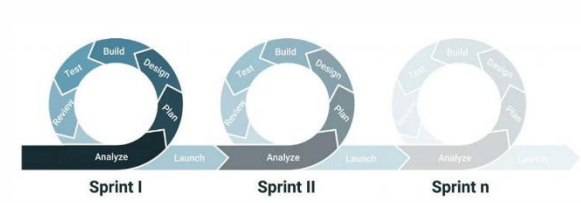
| Model | Best Used When |
|---|---|
| **Waterfall** | Requirements are fixed, clear, and well-documented. |
| **V-Model** | A high degree of testing is needed with defined requirements. |
| **Incremental** | Early delivery of parts of the system is needed. |
| **Prototyping** | Requirements are unclear or exploratory. |
| **Spiral** | Large projects with significant risks. |
| **Agile** | Projects with dynamic, evolving requirements. |

**Agile Methodology**

**Agile Methodology** is a flexible and iterative approach to software development that focuses on collaboration, customer feedback, and delivering functional software incrementally. It prioritizes adaptability and responsiveness to changes in requirements over rigid planning.

Agile is based on the principles outlined in the **Agile Manifesto**, which emphasizes:

1. **Individuals and interactions** over processes and tools.
2. **Working software** over comprehensive documentation.
3. **Customer collaboration** over contract negotiation.
4. **Responding to change** over following a fixed plan.



**Core Principles of Agile**

1. Deliver working software frequently (e.g., every 1-4 weeks).
2. Embrace changing requirements, even late in the development process.
3. Collaborate with stakeholders and customers throughout development.
4. Maintain a sustainable pace of work for the development team.
5. Focus on simplicity and delivering only what's necessary.
6. Regularly reflect and adjust processes to improve efficiency.

**Key Practices in Agile Methodology**

1. **Iterative Development:** Work is broken into small iterations or sprints, each producing a working increment of the software.
2. **Daily Standups:** Short, daily team meetings to discuss progress, obstacles, and plans.
3. **User Stories:** Requirements are written from the user's perspective to ensure they meet real-world needs.
4. **Retrospectives:** Teams reflect at the end of each sprint to identify and implement improvements.
5. **Backlog Management:** A prioritized list of features or tasks to be completed, updated regularly.
6. **Continuous Integration and Testing:** Code is frequently integrated and tested to ensure quality.

**Popular Agile Frameworks**

Several frameworks implement Agile principles. Some of the most popular are:

1. **Scrum:**
    - Involves roles such as Product Owner, Scrum Master, and Development Team.
    - Work is divided into time-boxed iterations called **sprints** (usually 2-4 weeks).
    - Includes ceremonies like Sprint Planning, Daily Standup, Sprint Review, and Sprint Retrospective.
2. **Kanban:**
    - Focuses on visualizing work and limiting work in progress (WIP) using a Kanban board.
    - Work items move through stages such as To Do, In Progress, and Done.
3. **Extreme Programming (XP):**
    - Emphasizes technical practices like pair programming, test-driven development (TDD), and continuous integration.
4. **Lean:**
    - Focuses on minimizing waste and maximizing value delivery.

**Advantages of Agile Methodology**

1. **Flexibility:** Easily adapts to changes in requirements.
2. **Customer Satisfaction:** Frequent deliveries and user involvement ensure the product meets their needs.
3. **Improved Quality:** Continuous testing and feedback improve the final product.
4. **Team Collaboration:** Encourages communication and collaboration among all stakeholders.
5. **Early Delivery:** Functional parts of the product are delivered early, even if the entire system isn’t complete.

**Disadvantages of Agile Methodology**

1. **Requires High Engagement:** Stakeholders and customers must be actively involved, which may not always be feasible.
2. **Unpredictable Costs:** Constant changes can make it difficult to estimate costs and timelines accurately.
3. **Scope Creep:** Without strict control, continuous additions can derail the project.
4. **Team Dependency:** Success depends heavily on the skills and collaboration of the team.
5. **Documentation Issues:** Prioritizing working software over documentation can lead to gaps in system knowledge.

**Agile Workflow**

Here's a typical Agile development workflow:

1. **Create a Backlog:** The Product Owner prepares a prioritized list of tasks or user stories.
2. **Sprint Planning:** The team selects tasks from the backlog for the sprint.
3. **Sprint Execution:** The team develops and tests the selected tasks within the sprint.
4. **Daily Standups:** The team discusses progress and blockers each day.
5. **Sprint Review:** At the end of the sprint, the team demonstrates the work completed to stakeholders.
6. **Retrospective:** The team reflects on what went well, what didn't, and how to improve.
7. **Repeat:** The process is repeated until the product is complete.

**When to Use Agile Methodology**

Agile is best suited for:

- Projects with evolving or unclear requirements.
- Dynamic environments where rapid delivery is needed.
- Projects where user feedback is critical for success.
- Teams that can work collaboratively and adapt quickly.

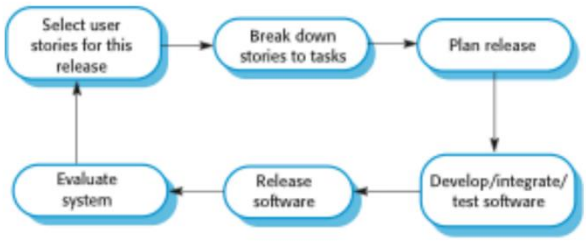
**Extreme Programming (XP)**

**Extreme Programming (XP)** is an Agile software development framework that focuses on improving software quality and responsiveness to changing customer requirements. XP is characterized by frequent releases in short development cycles, encouraging collaboration, simplicity, and adaptability.

**Core Values of XP**

1. **Communication:** Continuous and open communication among team members, customers, and stakeholders.
2. **Simplicity:** Focus on the simplest solution that works, avoiding unnecessary complexity.
3. **Feedback:** Regular feedback from customers and team members to ensure alignment with goals.
4. **Courage:** Encourages making changes when needed and addressing problems directly.
5. **Respect:** Promotes mutual respect among team members and stakeholders.

**Key Practices in Extreme Programming**




1. **Test-Driven Development (TDD):**
    - Write tests before writing code to define the desired behavior of the software.
    - Ensures that the code meets its requirements and prevents defects.
2. **Pair Programming:**
    - Two developers work together on the same code: one writes the code (driver), and the other reviews it (observer).
    - Improves code quality and facilitates knowledge sharing.
3. **Continuous Integration:**
    - Developers integrate code frequently (multiple times a day).
    - Automated tests are run after each integration to detect and fix issues early.
4. **Refactoring:**
    - Continuously improve the code's structure without changing its functionality.
    - Keeps the codebase clean and maintainable.
5. **Small Releases:**
    - Deliver small, functional parts of the system frequently.
    - Allows early user feedback and ensures incremental value delivery.
6. **Collective Ownership:**
    - All team members are responsible for the codebase.
    - Encourages collaboration and eliminates dependency on specific individuals.
7. **Onsite Customer:**
    - A customer representative is present with the team to clarify requirements and provide immediate feedback.
8. **Sustainable Pace (40-Hour Workweek):**
    - Teams work at a pace that can be sustained indefinitely without burnout.
9. **Coding Standards:**
    - All developers adhere to a set of consistent coding practices.
10. **Simple Design:**
    - Focuses on creating only what is necessary to meet the current requirements.

**XP Workflow**

1. **Planning:**
    - Involve the customer to define and prioritize user stories.
    - Break down the work into iterations, typically 1-3 weeks.
2. **Design:**

- Start with simple designs, ensuring they are flexible enough to adapt to future needs.

3. **Coding:**
    - Developers work in pairs, follow coding standards, and focus on delivering small, functional increments.
4. **Testing:**
    - Conduct automated and manual tests to ensure functionality and quality.
    - Perform tests continuously as part of the development process.
5. **Release:**
    - Deliver working software to the customer frequently, incorporating feedback for the next iteration.

**Advantages of XP**

1. **High Quality:** Rigorous testing and code reviews ensure fewer bugs and higher reliability.
2. **Adaptability:** Easily accommodates changing requirements due to iterative cycles.
3. **Customer Involvement:** Regular interaction with the customer ensures the software meets user needs.
4. **Knowledge Sharing:** Pair programming and collective ownership foster team collaboration.
5. **Faster Delivery:** Frequent releases deliver functional software earlier.

**Disadvantages of XP**

1. **Resource Intensive:** Requires high customer involvement and skilled developers.
2. **Less Predictable:** Difficult to estimate costs and timelines due to iterative changes.
3. **Not Suitable for Large Teams:** Works best for small to medium-sized teams.
4. **Pair Programming Challenges:** Some developers may find it challenging or inefficient.
5. **Overemphasis on Code:** May underprioritize documentation, potentially causing issues for long-term maintenance.

**When to Use XP**

- Projects with rapidly changing or unclear requirements.
- Small, co-located teams where collaboration is feasible.
- When high-quality software with frequent deliveries is required.
- Projects requiring active customer involvement.

**Comparison with Agile**

While XP is an Agile framework, it emphasizes technical practices like pair programming, TDD, and refactoring more than other frameworks like Scrum or Kanban. It is often considered the most **developer-centric Agile methodology**.

# MODULE 2

**Estimation**

1.Introduction

2.Post-Estimation Activities

3.Base Estimation

4.Decomposition

5.Empirical Models

6.Case Studies

7.Practical Applications

8.Conclusion

• In software development, estimation is determining the resources, time, and effort needed to complete a project.

• It helps teams set realistic expectations, plan efficiently, allocate resources effectively, and communicate project timelines to stakeholders.

**•Importance**

• **Project success:** Accurate estimation contributes to project success by ensuring that the team can meet deadlines, deliver on promises, and manage stakeholder expectations.

• **Resource optimization:** It helps in optimizing resource allocation, preventing overcommitment, and avoiding bottlenecks in the development process.

• **Client satisfaction:** Clients and stakeholders rely on accurate estimates for effective decision-making and to maintain a positive relationship with the development team

**Post-estimation - Refining Estimates for Better Accuracy**

• **Definition**: Post-estimation involves the process of reviewing and adjusting initial project estimates after they have been made. It's a crucial step in the project lifecycle to ensure accuracy and adaptability.

• **Purpose**:

- Reviewing and adjusting initial estimates.
- Factors:
- What influences post-estimation adjustments?

**Case Study 1: Post-Estimation Adjustments in a Real Project**

• **Context:** A software development project aiming to deliver a new feature for an e-commerce platform.

• **Post-Estimation Challenges:**

- *Challenge 1:* Unforeseen technical complexities arose during development.
- *Challenge 2:* User feedback prompted a need for additional functionalities.

• **Adjustments Made:**

- *Timeline Extension:* Originally estimated at 8 weeks, extended to 12 weeks to accommodate challenges.
- *Resource Reallocation:* Development team adjusted to include specialists addressing new requirements.

• **Outcome:**

- *Positive Impact:* Despite challenges, the feature was successfully delivered with improved functionality and met user expectations.

**Base Estimation - The Starting Point for Project Planning**

- **Definition**: What is the base estimation?
- **Significance**: How it sets the foundation for further discussions.
- **Adjustments**: Factors influencing adjustments to the base estimate.
- **Best Practices**: Guidelines for establishing a reliable base estimate.

**Case Study 2: Base Estimation Challenges and Solutions**

- **Context**: Initiating a software project with a complex feature set and evolving requirements.
- **Challenges Faced:**
  - *Challenge 1:* Initial estimates were based on incomplete information.
  - *Challenge 2:* Scope changes occurred due to evolving stakeholder expectations.
- **Solutions Implemented:**
  - *Iterative Estimation:* Adopted an iterative estimation approach to account for evolving requirements.
  - *Continuous Stakeholder Communication:* Maintained transparent communication to manage expectations.
- **Result:**
  - *Improved Planning:* Subsequent iterations of estimation aligned more closely with evolving project requirements.

**Decomposition - Breaking Down Complex Projects**

- **Definition:** Decomposition in software development is a technique that involves breaking down a complex project or task into smaller, more manageable components or pieces. These components are easier to understand, estimate, and manage, contributing to improved project planning and execution.
- **Techniques**: Overview of methods like Work Breakdown Structure (WBS).
- **WBS Example:**
  - Level 1: Project
  - Level 2: Phases (e.g., Planning, Development, Testing)
  - Level 3: Deliverables or Milestones
  - Level 3: Tasks or Activities
  - Level 4: Subtasks or Work Packages
- **Benefits**: improved Understanding, Efficient Resource Allocation, Risk Mitigation
- **Application**: How to apply decomposition in practice.

**Decomposition**

- Direct and indirect estimation are two methods used in various fields to estimate values or parameters.

- **Direct Estimation: Size oriented metrics (KLOC)**

  - **Definition**: Direct estimation involves obtaining a value or measurement of the desired parameter directly from the source or through direct observation. In this method, the measurement process is

straightforward, and the collected data represent the exact value being estimated.

• **Example**: If you want to estimate the height of a person, you can use a measuring tape or ruler to directly measure the person's height.

• **Characteristics**:

- *Precision*: Direct estimation can be highly precise, especially when using accurate measurement tools.
- *Simplicity*: The process is generally simple and involves minimal interpretation.
- *Size-oriented*

- **Indirect Estimation: Function-oriented metrics (FP)**

• **Definition**: Indirect estimation involves deriving the value of a parameter through a series of intermediate steps or by using related information rather than measuring it directly. This method relies on relationships or models between the observed data and the parameter of interest.

• **Example**: If you want to estimate the number of trees in a forest, you might use satellite imagery to measure the forest area and then apply a statistical model to estimate the tree density.

• **Characteristics**:

- *Relies on Models*: Indirect estimation often involves the use of statistical models or mathematical relationships.
- *Inference*: The estimate is inferred from observed data and assumptions.
- *Applicability*: Used when direct measurement is impractical, costly, or impossible.

**Case Study 3: Decomposition Success Stories**

• **Context**: Developing a comprehensive project management tool.

• **Decomposition Techniques Applied:**

- *Work Breakdown Structure (WBS):* Thoroughly defined and structured project tasks into smaller, manageable components.

• **Benefits Realized:**

- *Improved Understanding:* The team gained a clearer understanding of project scope and dependencies.
- *Efficient Resource Allocation:* Enhanced resource allocation based on the detailed breakdown.

• ***Impact:***

- *On-Time Delivery:* The project was delivered on time with minimized risks and efficient resource utilization.

**Empirical Models- Using Data for Predictions**

- **Definition**: Empirical models are mathematical models or statistical relationships derived from observed data, experimentation, or historical project data. These models are used to predict future outcomes, performance, or characteristics based on patterns identified in existing data.
- **Examples**: Introduction to models like COCOMO.
- **Application**: How empirical models leverage historical data.
- **Limitations**: Considerations when using empirical models.
  - Assumption of Similarity
  - Limited Generalization
  - Dynamic Nature of Technology
  - Dependency on Quality of Historical Data
  - Human Factor Considerations

**Case Study 4: Empirical Models in Action**

- **Context**:Developing a large-scale enterprise software solution.
- **Use of Empirical Models:**
  *COCOMO (Constructive Cost Model):* Utilized COCOMO for cost estimation based on historical project data.
- **Observations:**
  - *Accuracy of Estimates:* COCOMO provided accurate estimates based on historical data.
  - *Adaptability:* Easily adapted to changes in project scope and requirements.
- **Significance*:***
  - *Informed Decision-Making:* Empirical models facilitated informed decision-making by providing realistic estimations.

**Direct Estimation – Size Oriented Metrics**

- Effort = Size / Productivity
- Productivity = Size / Effort
- Size = Effort X Productivity
- Cost = Effort X Pay
- Duration = Effort / Team Size
- Team Size = Effort / Duration
- Effort = Duration X Team Size

**Example 1**

**Imagine you are a project manager overseeing a software development project with three modules (A, B, C) and a client component, each with distinct lines of code (LOC). The project details are as follows:**

**Module A: 6200 LOC**

**Module B: 9800 LOC**

**Module C: 8500 LOC**

**Client: 4800 LOC**

**• Additional Information:**

**Developer Productivity: 920 LOC per month**

**Developer Salary: $1600 per month**

**a) Calculate the total lines of code (LOC) for the entire project.**

**b) Using the productivity rate, estimate the total person-months required to complete the project.**

**c) Determine the total salary expenditure for the developers based on the estimated person-months and the**

**developer salary.**

**d) Discuss how variations in module sizes and developer productivity could impact the overall project timeline**

**and budget.**

**e) Consider the scenario where the developer salary is increased to $2000 per month. How would this change**

**affect the total salary expenditure for the project?**

*Person-month: It represents the total effort or work performed by one person in one month.*

**Solution**

- *Total Number of Lines of Code (LOC):* TotalLOC=6200+9800+8500+4800= 29300
- *Total Number of Person-Months:*

TotalPerson-Months= TotalLOC / Productivity = 29300 / 920 ≈31.85

- *Total Salary of Developers:*

TotalSalary=TotalPerson-Months×DeveloperSalary = 31.85×1600≈$50960

**Example - 2**

**Consider an application which contains 4 modules as follows:**

**Module 1 = 5400 LOC**

**Module 2 8500 LOC**

**Module 3 = 7300 LOC**

**Module 4 = 5500 LOC**

**Find the total expenditure if the productivity of the person is 870 LOC per month. Consider the salary of each developer is 1500 US dollars per month.**

- To find the total expenditure of the software development project, we need to calculate the total number of lines of code (LOC), the total number of person-months, and the total salary of the developers.
- The total number of LOC is the sum of the LOC of each module and the client, which is: **5400+8500+7300+5500=26700**
- The total number of person-months is the ratio of the total LOC and the productivity of the person, which is: **Effort = 26700/870 ≈30.69 pm**
- The total salary of the developers is the product of the total number of person-months and the salary of each developer, which is: **30.69×1500≈ $46035**

**Example 3**

**Consider a software application with four modules:**

**Module A: 18.2 KLOC**

**Module B: 12.8 KLOC**

**Module C: 25.6 KLOC**

**Module D: 9.3 KLOC**

**The cost of writing 1 KLOC is $60. If the productivity of the developer is 3.5 KLOC per month, find the cost of the application, the effort required to complete the project in a month, and predict the team size.**

65.9 KLOC X $60 = $ 3945

$$\text{Effort} = \frac{Size}{Productivity} = \frac{65.9\ KLOC}{3.5\ KLOC} \approx 18.828 \text{ pm}$$

$$\text{Size} = \frac{Effort}{Project\ Duration} = = \frac{18.828}{1} \approx 19 \text{ developers}$$

**Function Points - Introduction**

- **Definition**: Function Points (FP) is a software metric used to measure the size and complexity of a software application based on its functionality.
- **Purpose**: FP provides a standardized method to quantify the functionality that a software system delivers to users.

**Components of Function Points**

1. *External Inputs (EI):*
    i. Represents the count of unique external inputs that the application processes.
    ii. Examples: User inputs, data uploads, or external system interactions.
2. *External Outputs (EO):*
    i. Represents the count of unique outputs that the application generates for external entities.
    ii. Examples: Reports, user interface displays, or data exports.
3. *External Inquiries (EQ):*
    i. Represents the count of unique input/output combinations where the application processes data and sends a response.
    ii. Examples: Interactive queries or transactions involving both input and output.
4. *Internal Logical Files (ILF):*
    i. Represents the count of logical files that the application maintains.
    ii. Examples: Databases, data tables, or other data storage mechanisms.
5. *External Interface Files (EIF):*
    i. Represents the count of external interface files that the application uses.
    ii. Examples: Files shared with other systems or databases maintained by external entities.

**Function point Calculation**

| Measurement Parameters | Low | Average | Complex |
|---|---|---|---|
| **Input** | 3 | 4 | 6 |
| **Output** | 4 | 5 | 7 |
| **Inquiries** | 3 | 4 | 6 |
| **Files** | 7 | 10 | 15 |
| **External Interface** | 5 | 7 | 10 |

**Example –unadjustablefp**

| Measurement Parameters | Simple |
|---|---|
| **Input** | 30 |
| **Output** | 24 |
| **Inquiries** | 14 |

| | |
|---|---|
| **Files** | 5 |
| **External Interface** | 5 |

**Calculation of Function Points – Adjustable Error Factor**

- *Weights and Complexity Factors:* Explain the concept of weighting each component based on complexity (Low, Average, High).
- The CAF ranges from 0.65 to 1.35
- Formulas

$$FP = (\sum Weights \times Complexity\ Factors) \times Adjustment\ Factor$$

$$AF = 0.65 + 0.01 * \sum_{i=1}^{14} fi$$

**Complexity Adjustment Factor**

- CAF is a value that adjusts the unadjusted function point (UFP) based on the general functionality of the application.
- The CAF ranges from 0.65 to 1.35, depending on the degree of influence of 14 general system characteristics, such as data communications, distributed functions, performance, reusability, etc.
- Each characteristic has a degree of influence ranging from 0 (no influence) to 5 (strong influence).
- The CAF is calculated by adding 0.65 to the sum of the degrees of influence divided by 100.
- For example, if all the characteristics have a degree of influence of 3, which is the average value, then the CAF would be 0.65 + 0.01 ( 14 * 3) = 1.07.

**Example**

**Consider the project with the following functional units by assuming complexity adjustment factors and weighing factors as average.**

**Number of user inputs = 50**

**Number of user outputs = 40**

**Number of user inquiries = 30**

**Number of user files = 06**

**Number of external interfaces = 03**

**Compute the functional point.**

Calculation of Function Points – Adjustable Error Factor

$$UFP = (\sum Weights \times Complexity\ Factors) \times Adjustment\ Factor$$

= (50 * 4) + (40* 5) + (30 * 4) + (6 * 10) + (3 * 7) = 601

$$CAF = 0.65 + 0.01 * \sum_{i=1}^{14} fi$$

$$CAF = 0.65 + 0.01 * 14 * 3 = 1.07$$

FP = 601 * 1.07 = **602.07**

**Benefits of Function Points**

- *Standardization*: FP provides a standardized measure that facilitates consistent comparisons across different projects.
- *Estimation Accuracy:* Contributes to more accurate project estimation by considering the functional complexity.

**Limitations and Considerations**

- *Subjectivity*: Function points involve some subjectivity, especially in assigning complexity factors.
- *Adaptability*: It may not be suitable for all, especially those with highly dynamic requirements.

**COCOMO Model**

- The Constructive Cost Model (COCOMO) was developed by Dr. Barry Boehm in 1981.
- **Purpose**: COCOMO serves as a tool for estimating the resources required for software development projects.
- The primary objectives include:
  - *Effort Estimation:* Predicting the amount of effort (in person-months) required for a project.
  - *Cost Estimation:* Estimating the financial resources needed for the project.
  - *Schedule Estimation*: Predicting the project's timeline or duration.

**COCOMO Components**

Basic COCOMO Equation

- Effort (E)

$$E = a * (KLOC)^b$$

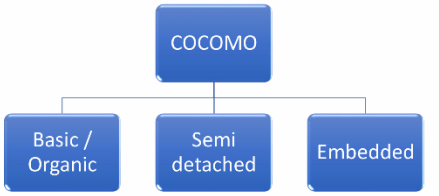
- *E*: Effort in person-months.
- *KLOC*: Size of the software in thousand lines of code.
- *a* and *b*: Constants determined based on project characteristics.

- Scheduled Time (D)

$$D = c * (E)^d$$

- *E*: Effort in person-months.
- *c* and *d*: Constants determined based on project characteristics.
- Person (P)

$$P = E/D$$




| PROJECT TYPE | a | b | c | d |
|---|---|---|---|---|
| Organic | 2.4 | 1.05 | 2.5 | 0.38 |
| Semidetached | 3 | 1.12 | 2.5 | 0.35 |
| Embedded | 3.6 | 1.2 | 2.5 | 0.32 |

**Assume that the mentioned 'e-Basket' application is of BASIC type software product that has 90,000 estimated lines of source code. The average salary of software engineers is Rs. 60,000/- per month. Using the COCOMO model, determine the cost, effort and time required to build the above product**

- Effort = 270.49
- Duration = 20.99 ~ 21
- Person = 270.49/21 = 13
- Sal = 780000

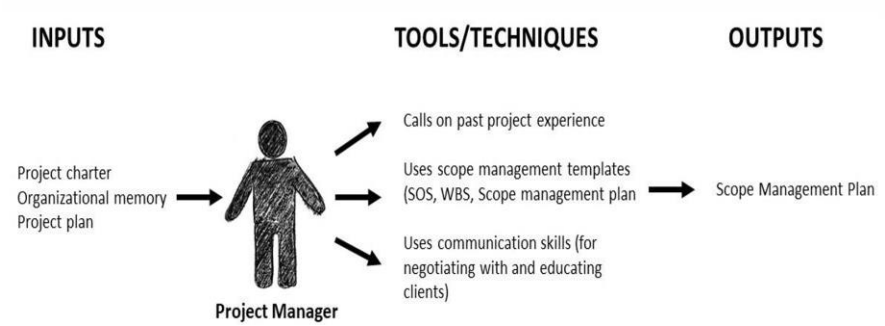
**Introduction To Software Project Management**

- Software Process Management (SPM) optimizes the software development lifecycle through structured methodologies, quality assurance, and a commitment to continuous improvement.
- The main goal is to ensure quality and continuous improvement for successful project delivery.
- A Project Manager is a professional responsible for planning, executing, and closing projects.
  1. Role
  2. Planning

3. Execution
4. Team Leadership
5. Communication
6. Risk Management
7. Resource Allocation
8. Monitoring and Control
9. Quality Assurance
10. Budget Management
11. Stakeholder Engagement
12. Closure and Evaluation

**Planning**

- Software planning is a critical phase in the software development lifecycle that involves comprehensive preparation for a successful project.
- *Significance*: It helps mitigate risks, ensures project alignment with organizational goals, and sets the stage for effective execution.
- Planning Objectives
    - To reduce uncertainty.
    - To bring cooperation and coordination.
    - To provide clarity in operation.
    - To anticipate unpredictable contingencies.
    - To achieve the predetermined goals.
    - To reduce competition.




**Scope**

- Scope refers to the combined objectives and requirements needed to complete a project.
- The scope of a project allows managers to estimate costs and the time required to finish the project.
- Project scope is the part of project planning that involves determining and documenting a list of specific project goals, deliverables, features, tasks, deadlines, and ultimately costs.
- **Steps in Scope Definition**
    1. Identify the project needs
    2. Confirm the objectives and goals of the Project
    3. Project Scope description
    4. Expectations and acceptance
    5. Identify constraints
    6. Identify necessary changes
- **Scope Project Requirements**
    - Functional Requirements
    - Non-Functional Requirements
    - Technical Requirements
    - Business Requirements
    - User Requirements
    - Regulatory requirements
- **An Example of Requirements**
- The following represents one possible example of each type of requirement as they would be applied to a bank's external ATM.

- *ATM functional requirement:* The system will enable the user to select whether or not to produce a hard-copy transaction receipt before completing a transaction.
- *ATM non-functional requirement:* All displays will be in white, 14-point Arial text on black background.
- *ATM technical requirement*: The ATM system will connect seamlessly to the existing customer's database.
- *ATM user requirement*: The system will complete a standard withdrawal from a personal account, from login to cash, in less than two minutes.
- *ATM business requirement:* By providing superior service to our retail customers, Monumental Bank's ATM network will allow us to increase associated service fee revenue by 10% annually on an ongoing basis.
- *ATM regulatory requirement:* All ATMs will connect to standard utility power sources within their civic jurisdiction, and be supplied with an uninterrupted power source approved by the company.

**Milestones**

- Milestones are used as signal posts
- **for**: a project's start or end date, a need for external review or input, a need for budget checks, submission of a major deliverable, and much more.
- **Milestone Examples**
  - Start and end dates for project phases
  - Key deliveries
  - Client and stakeholder approvals
  - Important meetings and presentations
  - Key dates or outages that may impact your timeline

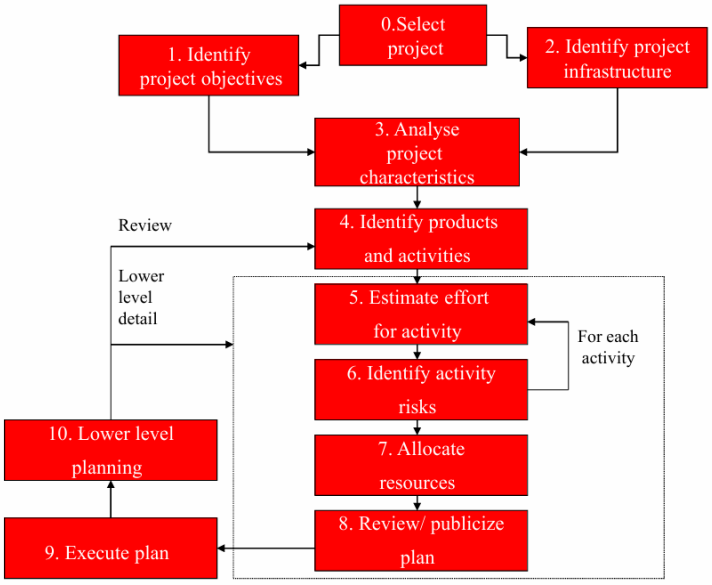
**Deliverables**

- A deliverable is any unique and verifiable product, result, or capability to perform a service that must be produced to complete a process, phase, or project.

**Milestones Vs. Deliverables**

- The main difference between deliverables and milestones is that milestones don't require a product to be delivered to the customer, client, or project sponsor.
- A milestone can be any threshold during which a project transitions to another phase
- For example, the completion of the foundation of the house is a milestone, but does not require any submission to the client, customer, or project sponsor, hence it is not a deliverable.

**'Step Wise' - Planning**




**Roles and Responsibilities of Project Manager**

- Define the scope of the project
- Identify stakeholders, decision-makers, and escalation procedures
- Develop a detailed task list (work breakdown structures)

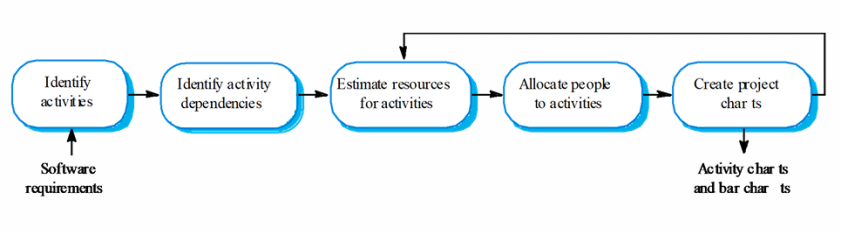
- Estimate time requirements
- Develop initial project management flow chart
- Identify required resources and budget
- Evaluate project requirements
- Identify and evaluate risks
- Prepare a contingency plan
- Identify interdependencies
- Identify and track critical milestones
- Participate in project phase review
- Secure needed resources
- Manage the change control process
- Report project status

**Skills required for Project Manager**

- Communication skills: listening, persuading
- Organizational skills: planning, goal-setting, analyzing
- Team Building skills: empathy, motivation, esprit de corps
- Leadership skills: set examples, be energetic, have vision (big picture), delegate, be positive
- Coping skills: persistence flexibility, creativity, patience,
- Technological skills: experience, project knowledge

**Project scheduling**

- Split the project into tasks and estimate the time and resources required to complete each task.
- Organize tasks concurrently to make optimal use of the workforce.
- Minimize task dependencies to avoid delays caused by one task waiting for another to complete.
- Dependent on project manager's intuition and experience.
- **The project scheduling process**




**Scheduling problems**

- Estimating the difficulty of problems and hence the cost of developing a solution is hard.
- Productivity is not proportional to the number of people working on a task.
- Adding people to a late project makes it later because of communication overheads.
- The unexpected always happens. Always allow contingency in planning.

**Case Study: E-Commerce Website Development**

1. ***Project Initiation:***

- **Goals and Objectives:**
  - Goal: Develop a user-friendly e-commerce website for online retail.
  - Objectives: Enable users to browse products, add items to the cart, and complete secure transactions.
- **Stakeholder Analysis:**
  - Stakeholders: Marketing, Development Team, Sales, Customers.
  - Analysis: Understand the expectations and interests of each stakeholder group.
- **Feasibility Study:**
  - Assess the feasibility of the project in terms of technology, budget, and market demand.

2. ***Scope Definition:***

- **Defining Project Scope:**
  - Scope: Develop a website with product listings, shopping cart functionality, user authentication, and a secure payment gateway.

    - Feature Identification: Features: Product pages, user registration, shopping cart, checkout, payment processing.
- **Limitations**:
    - Limitations: Initial launch to focus on desktop and mobile browsers. Future versions may include mobile apps.

***3. Requirement Analysis***

- **User Requirements:**
    - Gather requirements through interviews and surveys.
    - Examples: Intuitive product search, secure user accounts, responsive design.
- **Functional vs. Non-functional Requirements:**
    - Functional: Product search, user registration.
    - Non-functional: Page load time, system security.
- **Prioritization:**
    - High priority: Product pages, shopping cart, checkout.
    - Medium priority: User registration, mobile responsiveness.
    - Low priority: Social media integration.

***4. Estimation Techniques***

- **Effort, Time, and Resource Estimation:**
    - Estimate development effort using historical data and expert judgment.
    - Time estimation: Six months for development.
    - Resource estimation: Three developers, one designer, one QA.
- **Estimation Tools and Models:**
    - Use project management software for detailed task estimation and tracking.
    - Consider the PERT (Program Evaluation Review Technique) model.

***5. Project Scheduling***

- **Creating a Project Schedule:**
    - Milestones: Initial design, prototype completion, beta testing, launch.
    - Deadlines: Regular sprints with specific feature completion dates.
- **Project Management Tools:**
    - Utilize tools like Jira for task tracking and collaboration.
    - Regularly update the project schedule based on progress and feedback.

**Summary**

- A software project should be well planned to meet the requirements of the project
- Scope should define the boundary conditions
- Milestones are to monitored and accomplished within the timeline
- Deliverables should reflect client requirements and without errors.

**RISK MANAGEMENT**

**Risk**

- The opportunity to be vulnerable to the negative effects of future events
- Possible future problems
- An unpredictable occurrence or situation that has a positive or negative impact on the objectives of a project, whether it happens.

**Risk Management**

- Risk management is about identifying threats and designing strategies to mitigate their impact on a project.
- Reactive vs. proactive Risk Management
    - **Reactive**- Dealing with risks as they arise
        - Identifying risks during or after their occurrence.
        - Addressing issues as they become problems.
        - Late discovery of a critical software bug during testing.
    - **Proactive**- Anticipating and addressing risks before they become issues.
        - Identifying potential risks early in the project.
        - Implementing mitigation strategies beforehand.

**Reactive and Proactive Risk Management**

| Aspect | Reactive Risk Management | Proactive Risk Management |
|---|---|---|
| **Identification Timing** | After occurrence | Before occurrence |
| **Focus** | Issue resolution | Issue prevention |
| **Impact on Projects** | More disruptions | Smoother project flow |
| **Cost and Time Efficiency** | Higher costs and delays | Cost and time savings |

**Types of Risks**

- A chance (risk) is a likelihood that there will be some adverse circumstances
  - Product/Project risk
  - Process/ Technical risk
  - Business risk

1. **Technical Risks**
   a. **Example**: Integration Challenges
   b. **Scenario**: Integrating a new cloud-based customer relationship management (CRM) system with an existing on-premises database.
   c. **Risk**: The complexity of synchronizing data between the cloud and on-premises systems may lead to integration challenges.
2. **Schedule Risks**
   a. **Example**: Unforeseen Delays in Development
   b. **Scenario**: A software project to develop a mobile app experiences delays due to unforeseen technical complexities.
   c. **Risk**: The project timeline is at risk of being extended, impacting the planned release date.
3. **Budget Risks**
   a. **Example**: Changes in Project Scope
   b. **Scenario**: The client requests additional features not initially included in the project scope.
   c. **Risk**: The budget may exceed the initial estimate due to additional development efforts and resources required for the new features.
4. **Requirement Risks**
   a. **Example**: Frequent Changes in Client Requirements
   b. **Scenario**: A client frequently changes their mind about the design and functionality of a software application.
   c. **Risk**: Unclear and changing requirements may result in rework, increased development time, and client dissatisfaction.
5. **Resource Risks**
   a. **Example**: Key Team Member Leaving the Project
   b. **Scenario**: A key developer with specialized skills decides to leave the project unexpectedly.
   c. **Risk**: The loss of expertise may impact project progress, and finding a suitable replacement may take time.
6. **Quality Risks**
   a. **Example**: Inadequate Testing
   b. **Scenario**: Due to time constraints, thorough testing is skipped in the final stages of a software release.
   c. **Risk**: Undiscovered defects and issues may lead to poor software quality and customer dissatisfaction.
7. **Communication Risks**
   a. **Example**: Poor Communication with Clients
   b. **Scenario**: Inadequate communication with clients about project progress and changes.
   c. **Risk**: Misunderstandings may arise, and client expectations may not align with the delivered product.
8. **Market Risks**
   a. **Example**: Shifts in User Preferences

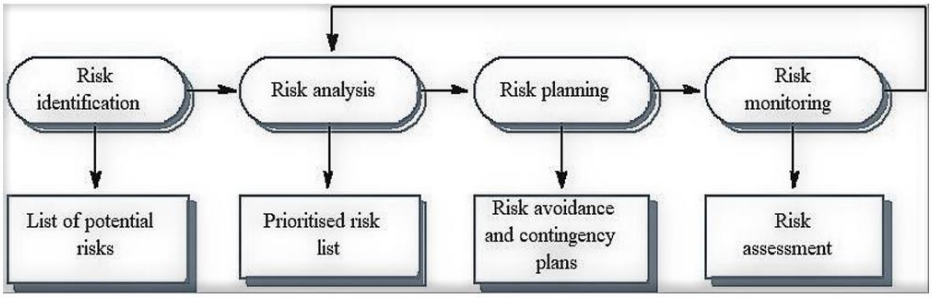
b. **Scenario**: A software company develops a product based on current user preferences, but market trends change.
c. **Risk**: The product may become less attractive to users, impacting market share and competitiveness.

9. **Legal and Compliance Risks**
    a. **Example**: Intellectual Property Infringement
    b. **Scenario**: Using third-party code without proper licensing or authorization.
    c. **Risk**: Legal issues may arise, leading to potential lawsuits and damage to the company's reputation.
10. **External Risks**
    a. **Example**: Natural Disasters
    b. **Scenario**: A development centre located in an earthquake-prone region.
    c. **Risk**: Natural disasters such as earthquakes may disrupt operations and impact project timelines.

**Risk Management Process**

In Risk management,

- **Risk identification**– what are the risks to a project?
- **Risk analysis**– which ones are really serious?
- **Risk planning**– what shall we do to minimise the risk?
- **Risk monitoring**– has the planning worked?




1. **Risk Identification**
    a. Howto identify risks?--
        i. Use of checklists
        ii. Brainstorming
        iii. Interviewing
2. **Risks Analysis/Prioritization----**
    a. Assessing the probability and severity of every risk.
    b. Probability: high or very high, moderate, very low, low
    c. Risk effects: serious, catastrophic, tolerable or insignificant.
    d. Risk probability may be qualitative and quantitative
    e. Risk exposure (RE)
        i. ***RE= (potential damage) x (probability of occurrence)***

**Risk types and examples**

| Risk | Probability | Effects |
|---|---|---|
| Organizational financial problems force reductions in the project budget. | Low | Catastrophic |
| It is impossible to recruit staff with the skills required for the project. | High | Catastrophic |
| Key staff are ill at critical times in the project | Moderate | Serious |
| Faults in reusable software components have to be repaired before these components are reused. | Moderate | Serious |
| Changes to requirements that require major design rework are proposed. | Moderate | Serious |
| The organization is restructured so that different management are responsible for the project. | High | Serious |
| The database used in the system cannot process as many transactions per second as expected. | Moderate | Serious |

| Risk | Probability | Effects |
|---|---|---|
| The time required to develop the software is underestimated | High | Serious |
| Software tools cannot be integrated. | High | Tolerable |
| Customers fail to understand the impact of requirements changes | Moderate | Tolerable |
| Required training for staff is not available. | Moderate | Tolerable |
| The rate of defect repair is underestimated | Moderate | Tolerable |
| The size of the software is underestimated | High | Tolerable |
| Code generated by code generation tools is inefficient | Moderate | Insignificant |

3. **Risk Prioritization**
    - Risk exposure (***Re***) is a weight associated to each possible risk factor that represents the importance of particular risk. The weight or the risk exposure for a particular risk indicates as product of the impact of the
      risk and the occurrences of the risk.

$$R_e(i) = Impact(i) * Occurence(i), \ \forall i = 1..n$$

$i$ represents number of risks ranging from 1 to n
*Impact* is the value denotes the effect of risks
*Occurrence* is the frequency of risks that taking place

4. **Risk Monitoring**
    a. Review each defined risk regularly to assess whether it is becoming less or more likely or not.
    b. Assess even how the consequences of the risk have changed.
    c. Increasing main risk should be addressed at progress meetings with management.

**Risk Control vs. Risk Mitigation**

- Implementing measures to prevent, reduce, or transfer identified risks.
- Example: Enhancing cybersecurity protocols to control the risk of data breaches and cyberattacks.
- Risk Mitigation: Taking actions to reduce the likelihood or impact of identified risks.
- Example: Diversifying investment portfolios to mitigate financial risks associated with market volatility.

**Mitigation Strategies - Examples**

- **Avoidance**: Changing project scope to eliminate a high-risk feature
- **Reduction**: Enhancing employee training to reduce the likelihood of errors
- **Transfer**: Purchasing insurance to transfer financial risks
- **Acceptance**: Acknowledging and budgeting for known risks

**Summary**

- For the success of a project, good project management is essential.
- Many types of risks cause problems for management
- Significant activities such as planning, estimating and scheduling should be monitored.
- Planning and estimating are iterative processes until meeting the requirements of the project

# MODULE 5

**Validation And Verification**

**Test plan(procedures)**

- A test plan is a **detailed document** which describes software testing areas and activities. It outlines the **test strategy, objectives, test schedule, required resources** (human resources, software, and hardware), **test estimation** and **test deliverables**.
- The test plan is a **template** for conducting software **testing activities** as a defined process that is **fully monitored** and **controlled** by the **testing manager**. The test plan is prepared by the **Test Lead** (60%), **Test Manager** (20%), and by the **test engineer** (20%).

**Types of Test Plan**

There are **three types** of the test plan

1. **Master** Test Plan
2. **Phase** Test Plan
3. Testing **Type Specific** Test Plans

**Master Test Plan**

- Master Test Plan is a type of test plan that has **multiple levels** of testing. It includes a **complete test strategy**.

**Phase Test Plan**

- A phase test plan is a type of test plan that **addresses any one phase** of the testing strategy. For example, a list of tools, a list of test cases, etc.

**Specific Test Plans**

- Specific test plan designed for **major types of testing** like security testing, load testing, performance testing, etc. In other words, a specific test plan designed for **non-functional testing.**

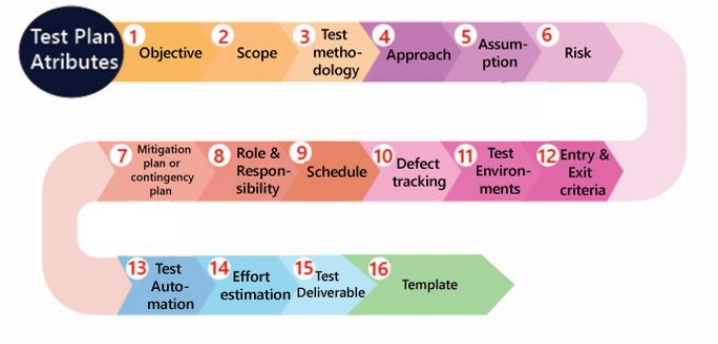
**How to write a Test Plan**

Making a test plan is the most crucial task of the test management process. **seven steps** to prepare a test plan.

- First, analyze product **structure** and **architecture**.
- Now design the test **strategy**.
- Define all the test **objectives**.
- Define the **testing area**.
- Define all the **useable resources**.
- Schedule all **activities** in an appropriate manner.
- Determine all the **Test Deliverables**.

**Test plan components or attributes**

The test plan consists of various parts, which help us to derive the entire testing activity.




**Test Plan Template**

Below find important constituents of a test plan

**1 Introduction**

1.1 Scope

1.1.1 In Scope

1.1.2 Out of Scope

1.2 Quality Objective

1.3 Roles and Responsibilities

**1) Introduction**

- **Objectives**: It consists of information about **modules, features, test data** etc., which indicate the **aim** of the application means the application **behavior, goal**, etc.
- **1.1) Scope**: It contains **information that needs to be tested** with respective of an application. The Scope can be further divided into two parts:
  - **1.1.1) In scope:** These are the modules that **need to be tested** rigorously (in detail).
  - **1.1.2) Out scope:** These are the modules, which **need not** be tested rigorously.

  For example, Suppose we have a Gmail application to test, where features to be tested such as Compose mail, Sent Items, Inbox, Drafts and the features which not be tested such as Help, and so on which means that in the **planning stage**, we will decide that which functionality has to be checked or not based on the time limit given in the product.

- **1.2) Quality Objective**
  Here make a mention of the **overall objective that you plan to achieve** with your manual testing and automation testing.
  Some **objectives of your testing project** could be
  - Ensure the Application Under Test (AUT)conforms to **functional** and **non-functional requirements**
  - Ensure the AUT meets the **quality specifications** defined by the client
  - **Bugs/issues** are identified and fixed before go live
- **1.3) Roles and Responsibilities**
  Detail description of the Roles and responsibilities of different team members like
    - QA Analyst
    - Test Manager
    - Configuration Manager
    - Developers
    - Installation Team

**2) Test Methodology**

- **2.1) Overview**
- Mention the reason of adopting a particular test methodology for the project. The test methodology selected for the project could be
    - Waterfall
    - Iterative
    - Agile
    - Extreme Programming
- **2.2) Test Levels**
  Test Levels define the **Types of Testing to be executed** on the Application Under Test (AUT). The Testing Levels primarily **depends on the scope** of the project, **time** and **budget constraints**.

- **2.3) Bug Triage**
  The goal of the triage is to
    - To define the **type of resolution** for each bug
    - To **prioritize bugs** and determine a **schedule** for all "**To Be Fixed Bugs'**.
- **2.4) Suspension Criteria and Resumption Requirements**
  Suspension criteria define the **criteria to be used to suspend** all or part of the testing procedure while Resumption criteria determine **when testing can resume** after it has been suspended.
- **2.5) Test Completeness**
- Here you define the **criteria's that will deem your testing complete**.
    - For instance, a few criteria to check Test Completeness would be **100% test coverage**
    - All Manual & Automated **Test cases executed**
    - All open **bugs are fixed** or will be fixed in next release

**3) Test Deliverables**

Here mention all the **Test Artifacts that will be delivered** during **different phases** of the **testing lifecycle**. Here are the simple deliverables

- Test Plan
- Test Cases
- Requirement Traceability Matrix
- Bug Reports
- Test Strategy
- Test Metrics
- Customer Sign Off

| Requirement no | Module name | High level requirement | Low level requirement | Test case name |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

**4) Resource & Environment Needs**

- **4.1) Testing Tools**
  Make a **list of Tools** like
    - **Requirements** Tracking Tool
    - **Bug** Tracking Tool
    - **Automation** Tools
- **4.2) Test Environment**
  It mentions the minimum hardware requirements that will be used to test the Application.
  Following software's are required in addition to client-specific software.
    - Windows 8 and above
    - Office 2013 and above
    - MS Exchange, etc.

**Sample Test Plan Document Banking Web Application Example**

**1 Introduction**

• The Test Plan is designed to prescribe the **scope, approach, resources, and schedule** of all **testing activities** of the project.

• The plan identifies the **items to be tested, the features to be tested, the types of testing** to be performed, the **personnel responsible** for testing, the **resources and schedule** required to complete testing, and the **risks** associated with the plan.

- **1.1 Scope**
    - **1.1.1 In Scope**
      All the feature of the Banking web application which were defined in software requirement and the specifications are to be tested

| Module Name | Applicable Roles | Description |
|---|---|---|

| | | |
|---|---|---|
| **Balance Enquiry** | Manager, Customer | **Customer**: A customer can view the balance of his accounts only.<br><br>**Manager**: A manager can view balances of all customers under his supervision. |
| **Fund Transfer** | Manager, Customer | **Customer**: A customer can transfer funds from his own account to any destination account.<br><br>**Manager**: A manager can transfer funds from any source bank account to a destination account. |
| **Mini Statement** | Manager, Customer | A mini statement shows the last 5 transactions of an account.<br><br>**Customer**: A customer can see the mini-statement of only his own accounts.<br><br>**Manager**: A manager can see the mini-statement of any account. |
| **Customized Statement** | Manager, Customer | A customized statement allows filtering and displaying transactions based on date and transaction value.<br><br>**Customer**: A customer can see customized statements of only his own accounts.<br><br>**Manager**: A manager can see customized statements of any account. |
| **Change Password** | Manager, Customer | **Customer**: A customer can change the password of only his account.<br><br>**Manager**: A manager can change the password of only his account, not his customers'. |
| **New Customer** | Manager | A manager can add a new customer. |
| **Edit Customer** | Manager | A manager can edit details like address, email, and telephone of a customer. |
| **New Account** | Manager, Customer | **Customer**: A customer can have multiple saving and current accounts.<br><br>**Manager**: A manager can add a new account for an existing customer. |
| **Edit Account** | Manager | A manager can edit account details for an existing account. |
| **Delete Account** | Manager | A manager can delete an account for a customer. |
| **Delete Customer** | Manager, Customer | **Customer**: A customer can be deleted only if he/she has no active accounts.<br><br>**Manager**: A manager can delete a customer. |
| **Deposit** | Manager | A manager can deposit money into any account, usually done when cash is deposited at a bank branch. |
| **Withdrawal** | Manager | A manager can withdraw money from any account, usually done when cash is withdrawn at a bank branch. |

- **1.1.2 Out of Scope**

  These features are not be tested because they are not included in the software requirement specs

  1. User Interfaces
  2. Hardware Interfaces
  3. Software Interfaces
  4. Database logical
  5. Communications Interfaces

6. Website Security and Performance

2. **1.2 Quality Objective**

   The test objectives are to **verify the Functionality** of bank website, the project should focus on **testing** the banking **operation** such as Account Management, Withdrawal, and Balance.

3. **1.3 Roles and Responsibilities**

| No. | Member | Tasks |
|---|---|---|
| 1 | **Test Manager** | - **Manage** the whole project<br>- Define project **directions**<br>- Acquire appropriate **resources** |
| 2 | **Test** | - **Identify** and describe appropriate test **techniques/tools/automation architecture**<br>- **Verify and assess** the Test Approach<br>- **Execute** the tests, log results, report defects |
| 3 | **Developer in Test** | - **Implement** the test **cases**, test **program**, test **suite**, etc. |
| 4 | **Test Administrator** | - Build up and ensure test **environment** and **assets** are managed and maintained<br>- Support testers to use the test environment **for test execution** |
| 5 | **SQA Members** | - Take charge of **quality assurance**<br>- Check to confirm whether the testing **process meets specified requirements** |

**2 Test Methodology**

4. **2.1 Overview**
5. **2.2 Test Levels**

   In the project, there're 3 types of testing should be conducted.

   - **Integration Testing** : Individual software modules are combined and tested as a group
   - **System Testing**: Conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements
   - **API testing**: Test all the APIs create for the software under tested

6. **2.3 Bug Triage**
7. **2.4 Suspension Criteria and Resumption Requirements**

   If the team members report that there are 40% of test cases failed, suspend testing until the development team fixes all the failed cases.

8. **2.5 Test Completeness**
   - Specifies the **criteria** that denote a successful completion of a test phase
   - **Run rate** is mandatory to be 100% unless a clear reason is given.
   - **Pass rate** is 80%, achieving the pass rate is mandatory
9. **2.6 Project task and estimation and schedule**

| Task | Members | Estimate effort |
|---|---|---|
| **Create the test specification** | Test Designer | 170 man-hour |
| **Perform Test Execution** | Tester, Test Administrator | 80 man-hour |
| **Test Report** | Tester | 10 man-hour |
| **Test Delivery** | | 20 man-hour |
| **Total** | | 280 man-hour |

10. **3 . Test Deliverables**

    Test deliverables are provided as below

    **Before testing phase**

    - Test **plans** document.

- Test **cases** documents
- Test **Design** specifications.

**During the testing**

- Test **Tool Simulators**.
- Test **Data**
- Test **Trace-ability Matrix**
- **Error logs** and **execution** logs.

**After the testing cycles is over**

- Test **Results**/reports
- **Defect** Report
- **Installation**/ Test procedures **guidelines**
- **Release notes**

**4 Resource & Environment Needs**

- **4.1 Testing Tools**

| No. | Resources | Descriptions |
|---|---|---|
| **1.** | Server | Need a Database server which install MySQL server<br>Web server which install Apache Server |
| **2.** | Test tool | Develop a Test tool which can auto generate the test result to the predefined form and automated test execution |
| **3.** | Network | Setup a LAN Gigabit and 1 internet line with the speed at least 5 Mb/s |
| **4.** | Computer | At least 4 computer run Windows 7, Ram 2GB, CPU 3.4GHZ |

- **4.2 Test Environment**
  It mentions the minimum **hardware and software requirements** that will be used to test the Application. Following software's are required in addition to client-specific software.
  - Windows 11 and above
  - Office 2021 and above
  - MS Exchange, etc.

**Test Design**

- Test design is a process that **describes "how" testing should be done**. It includes processes for the **identifying test cases** by enumerating steps of the defined **test conditions**.
- The test cases may be **linked** to the test conditions and project objectives **directly or indirectly** depending upon the methods used for **test monitoring, control and traceability**.
- The objectives consist of **test objectives, strategic objectives** and **stakeholder definition of success**.

**When to create test design?**

- After the test **conditions are defined** and **sufficient information is available** to create the test cases of high or low level, test design for a specified level can be created.
- For **lower-level testing**, test analysis (what is to be tested) and design (test cases) are combined activity. For **higher level testing**, test analysis is performed first, followed by test design.
- There are some activities that routinely take place when the test is implemented. These activities may also be incorporated into the design process when the tests are created in an iterative manner.

**Test Implementation**

- Test implementation is the practice of **organizing and prioritizing tests**. This is carried out by **Test Analysts** who implement the test designs as **real test cases, test processes** and **test data**.
- If you are observing **the IEEE 829 standard**, you must define these **parameters** as well :
  - Test **inputs**

- **Expected results** for each test case
- **Steps** to be followed for each test process

**Some of the checks that could be performed to confirm that the team is ready to execute tests include:**

- Ensuring that the **test environment** is in place
- Ensuring every **test case** is well **documented and reviewed**
- Putting test environment in a **state of readiness**
- Checking against **explicit and implicit entry** criteria for the **specified test level**
- Describing test **environment** as well as test **data** in **great detail**
- Performing **code acceptance check** by running it on test environment
- For example, if the tests are to be documented for using again in future for regression testing, the test documents will record step by step description of executing the test.

**Disadvantages of early test implementation**

Implementing the tests early may have some disadvantages too.

- For example, if Agile lifecycle has been adopted for product development, the **code itself may undergo drastic changes** between **consecutive iterations**. This will render the whole test implementation useless.
- In fact, any **iterative development lifecycle** will affect the code between iterations, even if it is not as drastic as that in the Agile lifecycle.
- This will make **pre-defined tests obsolete** or require continuous and **resource intensive maintenance**.
- Similarly, even in case of **sequential lifecycles**, if the project is badly managed and **requirements keep changin**g even when project is in an advanced state, early test implementation can be rendered obsolete.
- Therefore, before starting the test implementation process, **Test Manager** should consider these important points:
  - Software development **life cycle** being used
  - **Features** that need to be tested
  - Probability of **change in requirement** late into project lifecycle
  - Possibility of **changes in code** between two iterations

**Test Execution**

- The term Test Execution tells that the testing for the product or application needs to be executed in order to obtain the expected result.
- After the development phase, the testing phase will take place where the various levels of testing techniques will be carried out and the **creation and execution of test cases** will be taken place.
- Test Execution is the process of executing the **tests written by the tester** to **check** whether the developed code or functions or modules are **providing the expected result** as per the client requirement or business requirement.

**Importance of Test Execution:**

- *The project runs efficiently*: Test execution ensures that the project runs smoothly and efficiently.
- *Application competency*: It also helps to make sure the application's competency in the global market.
- *Requirements are correctly collected*: Test executions make sure that the requirements are collected correctly and incorporated correctly in design and architecture.
- *Application built in accordance with requirements*: It also checks whether the software application is built in accordance with the requirements or not.

**Activities for Test Execution**

The following are the **5 main activities** that should be carried out during the test execution.

1. **Defect Finding and Reporting:** Defect finding is the process of **identifying the bugs or errors** raised while executing the test cases on the developed code or modules. If any error appears or any of the test cases failed then it will be **recorded** and the same will be **reported** to the respective development team.

2. **Defect Mapping**: After the error has been detected and reported to the development team, the **development team will work on those errors and fix them** as per the requirement. Once the development team has done its job, the **tester team will again map the test cases** or test scripts to that developed module or code to run the entire tests to ensure the correct output.
3. **Re-Testing**: Re-Testing is the process of **testing the modules or entire product again** to ensure the smooth release of the module or product. In some cases, the new module or functionality will be developed after the product release. In this case, **all the modules will be re-tested** for a **smooth release**. So that it cannot cause any other defects after the release of the product or application.
4. **Regression Testing**: Regression Testing is software testing that ensures that the **newly made changes** to the code or newly developed modules or functions **should not affect the normal processing** of the application or product.
5. **System Integration Testing**: System Integration Testing is a type of testing technique that will be used to **check the entire component or modules** of the system in a single run. It ensures that the whole system will be checked in a **single test environment** instead of checking each module or function separately.

**Test Execution Process**

The test Execution technique consists of **three different phases**. The three main phases of test execution are the **creation of test cases, test case execution,** and **validation of test results.**

1. **Creation of Test Cases**: The first phase is to create suitable test cases ***for each module or function***. Here, the ***tester with good domain knowledge*** must be required to create suitable test cases. The creation of test cases ***should not be delayed*** else it will cause excess time to release the product. The created test cases should ***not be repeated*** again. It should ***cover all the possible scenarios*** raised in the application.
2. **Test Cases Execution**: After test cases have been created, execution of test cases will take place. The ***Quality Analyst team*** will either do ***automated or manual testing*** depending upon the test case scenario. The ***selection of testing tools*** is also important to execute the test cases.
3. **Validating Test Results:** After executing the test cases, ***note down the results*** of each test case in a ***separate file or report***. Check whether the executed test cases ***achieved the expected*** result and ***record the time*** required to complete each test case i.e., ***measure the performance*** of each test case. If any of the test cases is failed or not satisfied the condition then ***report it to the development team*** for validating the code.

**Ways to Perform Test Execution**

1. **Run test cases**: It is a simple and easiest approach to run test cases on the local machine and it can be coupled with other artifacts like test plans, test suites, test environments, etc.
2. **Run test suites**: A test suite is a ***collection of manual and automated test cases*** and the test cases can be executed ***sequentially or in parallel***. Sequential execution is useful in cases where the result of the last test case depends on the success of the current test case.
3. **Run test case execution and test suite execution records**: Recording test case execution and test suite execution is a key activity in the test process and ***helps to reduce errors***, making the testing process more ***efficient***.
4. **Generate test results without execution**: Generating test results from non-executed test cases can be helpful in achieving ***comprehensive test coverage***.
5. **Modify execution variables**: Execution variables can be modified in the test scripts for ***particular test runs***.
6. **Run automated and manual tests**: Test execution can be done manually or can be automated.
7. **Schedule test artifacts:** Test artifacts include ***video, screenshots, data reports***, etc. These are very helpful as they ***document the results*** of the past test execution and provide information about what needs to be done in ***future test execution***.
8. **Defect tracking**: Without defect tracking test execution is not possible, as during testing one should be able to track the defects and ***identify what when wrong and where***.

**Test Execution Priorities** are nothing but prioritizing the test cases depending upon several factors.

- **Complexity**: The complexity of the test cases can be determined by including several factors such as ***boundary values*** of test cases, ***features*** or ***components*** of test cases, ***data entry*** of test cases, and how much the test cases ***cover*** the given business problem.
- **Risk Covered**: How much risk that a certain test case may undergo to achieve the result. Risk in the form of ***time required to complete*** the test case process, ***space complexity*** whether it is executed in the given memory space, etc.,
- **Platforms Covered**: It simply tells that in which ***platform or operating system*** the test cases have been executed i.e., test cases executed in the Windows OS, Mac OS, Mobile OS, etc.,
- **Depth**: It covers how depth the given test cases cover each functionality or module in the application i.e., how much a given test procedure ***covers all the possible conditions*** in a ***single functionality or module***.
- **Breadth**: It covers how the breadth of the given test cases covers the entire functionality or modules in the application i.e., how much a given test procedure ***covers all the possible conditions*** in the ***entire functionality*** or modules in the product or application.

**Test Execution States**

The **tester** or the **Quality Analyst team** reports or notices the result of each test case and records it in their documentation or file. There are various **results** raised when executing the test cases. They are

- **Pass**: It tells that the test cases executed for the module or function are successful.
- **Fail**: It tells that the test cases executed for the module or function are not successful and resulted in different outputs.
- **Not Run**: It tells that the test cases are yet to be executed.
- **Partially Executed**: It tells that only a certain number of test cases are passed and others aren't met the given requirement.
- **Inconclusive**: It tells that the test cases are executed but it ***requires further analysis*** before the final submission.
- **In Progress**: It tells that the test cases are currently executed.
- **Unexpected Result:** It tells that all the test cases are executed successfully but provide different unexpected results.

**Test Execution Report**

The Test Execution Report is nothing but a ***document that contains all the information*** about the ***test execution process***. The documentation or the report contains various information. They are:

- Who all are going to execute the test cases?
- Who is doing the unit testing, integration testing, system testing, etc.,
- Who is going to write test cases?
- The number of test cases executed successfully.
- The number of test cases failed during the testing.
- The number of test cases executed today.
- The number of test cases yet to be executed.
- What are the automation test tools used for today's test execution?
- What are the modules/functions testing today?
- Recording the issues while executing the test cases.
- What is today's testing plan?
- What is tomorrow's testing plan?
- Recording the pending plans.
- Overall success rate.
- Overall failure rate.
- Test Summary Report Identifier.
- Summary.
- Variances.
- Comprehensive Assessment.

- Summary of Results.
- Evaluation.
- Summary of Activities.
- Approval.

**Guidelines for Test Execution**

- **Write** the suitable test cases for each module of the function.
- **Assign** suitable test cases to respective modules or functions.
- Execute both **manual testing** as well as **automated testing** for successful results.
- Choose a suitable **automated tool** for testing the application.
- Choose the correct **test environment** setup.
- **Note down** the **execution status** of each test case and note down the **time taken** by the system to complete the test cases.
- **Report** all the success **status** and the failure status to the **development team**
- **Track the test status** again for the **already failed** test cases and **report** it to the team.
- **Highly Skilled Testers are required** to perform the testing with **less or zero failures/defects**.
- **Continuous testing** is required until success test report is achieved. Reviews, Inspection and Auditing

**HYBRID INTEGRATION TESTING METHOD**

combines the features of **Top-Down Integration Testing** and **Bottom-Up Integration Testing**. This method is used in **integration testing**, a level of software testing where individual modules are combined and tested as a group to detect issues in their interaction.

**Advantages:**

1. **Features of Both Approaches:**
    - It leverages the strengths of both Top-Down and Bottom-Up testing.
    - Top modules and bottom modules are tested ***simultaneously***.
2. **Time-Efficient:**
    - ***Parallel testing*** in both directions reduces the overall testing time.
3. **Comprehensive Testing:**
    - Ensures ***all modules are covered*** and their interactions are tested thoroughly.

**Disadvantages:**

1. **High Complexity:**
    - Managing simultaneous testing in both directions can be challenging.
2. **Requires High Focus:**
    - Testers need to ***manage dependencies*** carefully, as both top-level and bottom-level modules interact during the process.

**REGRESSION TESTING**

- Regression testing is a ***black box testing technique***. It is used to authenticate a ***code change*** in the software ***does not impact the existing functionality*** of the product. Regression testing is making sure that the product works fine with new functionality, bug fixes, or any change in the existing feature.
- Regression testing is a type of software testing. ***Test cases are re-executed*** to check the previous functionality of the application is working fine, and the new changes have not produced any bugs.

- Regression testing can be performed on a new build when there is a significant change in the original functionality. It ensures that the code still works even when the changes are occurring. Regression means ***Re-test those parts of the application, which are unchanged.***
- Regression tests are also known as the ***Verification Method***. Test cases are often ***automated***. Test cases are required to execute many times and running the same test case again and again manually, is time-consuming and tedious too.

**When to Perform Regression Testing**

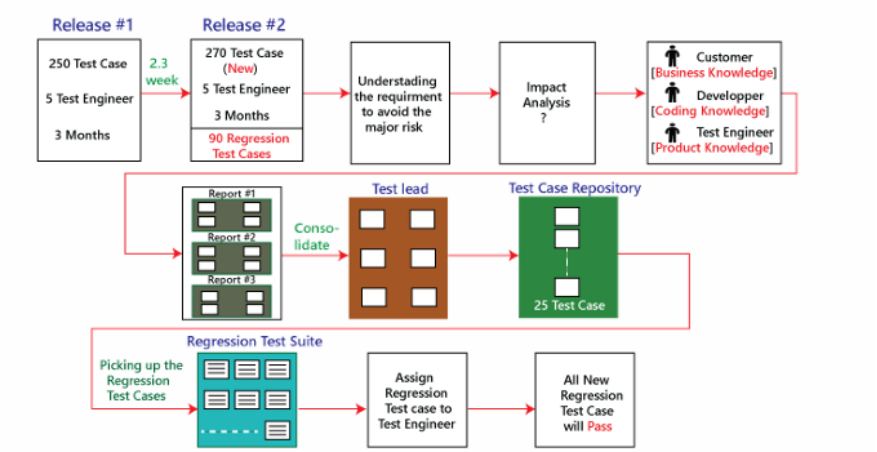
- **Production Code Modification**: Anytime the production code is changed.
- **Scenarios for Regression Testing**:
    i. **New Functionality Added**: Example: Adding Facebook login to a website.
    ii. **Change Requirement**: Example: Removing the "Remember Password" feature.
    iii. **Defect Fixes**: Example: Testing a fixed login button after a bug report.
    iv. **Performance Issue Fix**: Example: Reducing a homepage's load time from 5 seconds to 2 seconds.
    v. **Environment Change**: Example: Upgrading the database from MySQL to Oracle.

**How to Perform Regression Testing**

- **Need for Regression Testing**: Necessary during software maintenance involving enhancements, error corrections, and deletions of features that may affect functionality.
- **Techniques for Regression Testing**:
    i. **Re-test All**: Confirmation testing, Re-execution of all test cases; expensive in terms of time and resources.
    ii. **Regression Test Selection**: Executing a selected suite of test cases, divided into:
        - **Reusable Test Cases**: Can be used in future cycles.
        - **Obsolete Test Cases**: Cannot be reused.
    iii. **Prioritization of Test Cases**: Selecting test cases based on business impact and critical functionalities.

**Regression Testing Process**

- **Across Builds**: Retesting bugs and performing regression testing on ***dependent modules***.
- **Across Releases**: Begins with a ***new release*** due to potential impacts on existing features.
- **Steps in the Process**:
    i. No regression testing in Release #1 (new release).
    ii. Regression testing starts in Release #2 with new requirements.
    iii. Understanding ***requirements*** before impact analysis.
    iv. ***Impact analysis*** conducted by customers, developers, and test engineers.
    v. Preparing ***impact area documents*** for maximum coverage.
    vi. Consolidating ***reports*** by the Test Lead.
    vii. Selecting necessary regression ***test cases*** for the suite.
    viii. Assigning regression test cases to ***test engineers***.
    ix. Ensuring ***stability*** of regression and ***new feature tests*** before handing over to the customer.




**Types of Regression Testing**

1. **Unit Regression Testing (URT)**: Testing only the changed unit without impacting other areas.
    - Example: Testing a modified search button.
2. **Regional Regression Testing (RRT)**: Testing the modified unit along with its impact areas.
    - Example: Testing a fixed bug in one module and checking its impact on related modules.
3. **Full Regression Testing (FRT)**: Testing all features when significant changes occur.
    - Conditions for FRT:
        - Modifications in source files affecting the entire product.
        - Multiple changes necessitating comprehensive testing.

`

**MUTATION TESTING**

- **Definition**: Mutation testing is a ***white-box testing technique*** where ***intentional errors*** (mutations) are introduced into the program to evaluate the effectiveness of existing test cases in detecting these errors.
- **Objective**: The primary aim is to ensure that the ***test cases can identify differences in output*** between the original program and its mutated versions (mutants).
- **Process**: Slight modifications are made to the original program to create mutants. If a test case fails to detect a mutation, it indicates either a flaw in the program or inefficiency in the test case itself.

**Importance of Mutation Testing**

- **Quality Assessment**: It assesses the quality of test cases by determining their ability to identify faults in the code.
- **Fault-Based Testing**: This method is also known as fault-based testing, as it deliberately introduces errors to test the robustness of the test cases.

**Types of Mutation Testing**

Mutation testing can be classified into three main types:

1. **Decision Mutations**:
    - **Purpose**: Focuses on identifying design errors by modifying decision-making constructs in the code.
    - **Modifications**:

        - **Arithmetic Operators**:
            - Change + to -
            - Change * to **
            - Change + to i++
        - **Logical Operators**:
            - Change > to <
            - Change OR to AND
            - Change >= to <
    - **Example**: If a condition checks if x > 10, changing it to x < 10 tests if the logic holds under altered conditions.
2. **Value Mutations**:
    - **Purpose**: Involves modifying specific values in the code to identify errors.
    - **Modifications**:
        - Change small values to larger values.
        - Change larger values to smaller values.
    - **Example**: If a variable is supposed to hold a maximum value of 100, changing it to 200 tests the program's response to unexpected input ranges.
3. **Statement Mutations**:
    - **Purpose**: Involves altering or removing statements in the code to evaluate the impact on program execution.
    - **Modifications**:
        - Replace a statement with another.
        - Remove a statement entirely.
    - **Example**: If a loop iterates through an array, removing the loop statement tests how the program behaves without that control flow.

**How to Perform Mutation Testing**

The process of conducting mutation testing involves several steps:

1. **Create Mutants**:
    - Introduce intentional errors into the source code to create multiple versions of the program, known as mutants. Each mutant should contain one specific error.
2. **Execute Test Cases**:
    - Use the existing test cases to run against both the original program and the mutants. The goal is to identify whether the test cases can detect the introduced errors.
3. **Compare Outputs**:
    - Analyze the outputs of the original program and the mutants. If the outputs differ, the test case has successfully identified the mutation, indicating its effectiveness.
4. **Evaluate Test Case Efficiency**:

- If a mutant produces the same output as the original program, it suggests that the test case is inadequate for detecting that specific fault. This indicates the need for improving the test case.

5. **Refinement**:
    - Based on the results, refine the test cases to increase their effectiveness in detecting faults. This might involve adding new test cases or modifying existing ones.

**Conclusion**

Mutation testing is a powerful technique for evaluating the robustness of test cases. By systematically introducing errors and analyzing the ability of test cases to detect them, developers can enhance the quality of their software testing processes and ensure that their test suites are comprehensive and effective.

**WEB-BASED TESTING**

- **Definition**: Web testing is a software testing technique focused on ***evaluating web applications or websites*** to identify ***errors, bugs, and quality-related risks*** before they are released to end-users.
- **Importance**: Proper testing is essential to ensure that a web application functions correctly and ***meets user expectations***. It goes beyond finding common bugs to assessing the ***overall quality and performance*** of the application.
- **Testing Approach**: Understanding the ***architecture and key areas of a web application*** is crucial for effective planning and execution of testing strategies.

**Types of Web Testing**

Web testing can be categorized into four main types:

1. **Static Website Testing**:
    - **Definition**: A static website displays content that is exactly as it is stored on the server, without dynamic features.
    - **Focus Areas**:
        - **User Interface (UI)**: Emphasizes the ***visual aspects*** such as font size, color, spacing, and layout.
        - **Functionality Checks**: Includes ***verifying links,*** checking the "Contact Us" form, and ensuring that all ***URLs are operational***.
    - **Testing Techniques**: ***Manual review*** and ***automated tools*** can be used to test UI elements and links.
2. **Dynamic Website Testing**:
    - **Definition**: A dynamic website includes ***both front-end (UI) and back-end (database)*** components, allowing for regular updates based on user interactions.
    - **Focus Areas**:
        - **Functionality Testing**: Evaluates how buttons and forms behave, checks error messages, and assesses ***user interactions***.
        - **Backend Validation**: Ensures that data entered in the front-end is ***correctly updated in the database.***
    - **Testing Techniques**: Involves both ***manual testing of user interactions*** and ***automated testing for backend*** processes.

3. **E-Commerce Website Testing**:
    - **Definition**: E-commerce websites are ***complex platforms*** that require thorough testing due to multiple functionalities and user interactions.
    - **Focus Areas**:
        - **Shopping Cart Functionality**: Verifying that items can be added, removed, and that the cart behaves as expected.
        - **User Registration and Login**: Ensuring that users can create accounts and log in without issues.
        - **Payment Processing**: Testing the payment gateway to confirm that transactions are secure and successful.
        - **Security Checks**: Assessing the website's security measures to protect user data and transactions.
    - **Testing Techniques**: Combination of ***manual and automated testing*** to cover all functional aspects and security requirements.
4. **Mobile-Based Web Testing**:
    - **Definition**: This testing focuses on the ***compatibility of websites across various devices***, particularly mobile devices, due to the increasing number of users accessing sites via smartphones and tablets.
    - **Focus Areas**:
        - **Responsive Design**: Ensuring that the website adapts well to different ***screen sizes*** and ***orientations***.
        - **Performance Testing**: Checking ***load times and usability*** on mobile devices.
        - **Cross-Browser Compatibility**: Testing how the site performs on different mobile browsers (e.g., Chrome, Safari).
    - **Testing Techniques**: Use of ***emulators*** and ***real devices*** to test responsiveness and functionality.

**Points to Consider While Testing a Website:**

When conducting web testing, several key factors should be considered:

- **Link Validity**: Ensure that all internal and external links are valid and lead to the correct pages without any broken links.
- **System Compatibility**: Verify that the website functions correctly across different operating systems, browsers, and devices.
- **User Interface Optimization**: Check that display sizes, font choices, and layout are optimal for user experience.
- **Security Requirements**: Identify necessary security measures, especially for sensitive information and transactions.
- **Analytics and Control**: Determine requirements for website analytics, including tracking user behavior and managing graphics and URLs.
- **Customer Assistance Features**: Assess the need for contact forms or customer support features on the website to enhance user interaction.

**Conclusion**

Web-based testing is essential for ensuring the functionality, security, and overall user experience of web applications. By understanding the types of web testing and considering critical factors during the testing process, developers and testers can create robust applications that meet user needs and expectations.

**Best Mobile App Testing Tools for Automation Testing**

**1. Kobiton**

- **Overview**: Kobiton provides an easy-to-use platform for ***accessing real devices*** for both manual and automated testing.
- **Key Features**:
    - **Access to Real Devices**: Supports complex gestures, ADB shell commands, geo-location, and device connection management.
    - **Real-Time Logs**: Offers ***real-time insights*** into logs for issue identification and resolution.
    - **Device Lab Management**: Facilitates ***centralized testing history and data*** logs for better collaboration.
    - **Programming Support**: Compatible with languages like ***C#, Java, Ruby, NodeJS, PHP, and Python***.
    - **Framework Support**: Works with ***React Native, Ionic, Electron, NativeScript, Xamarin, and Flutter***.
    - **Testing Types**: Supports ***performance, automation, manual, and functional*** testing.
    - **Integrations**: Seamlessly integrates with CI/CD tools like ***Travis CI, Jenkins, Azure DevOps, and Jira***.
    - **Customer Support**: Available via ***chat, contact form, and email***.
    - **Platforms**: Supports both ***iOS and Android***.
    - **Pricing**: Plans start at ***$75/month*** with a 14-day free trial (no credit card required).

**2. testRigor**

- **Overview**: testRigor allows users to ***express tests in plain English***, enabling users of all technical abilities to build end-to-end tests.
- **Key Features**:
    - **English Test Cases**: Tests are written in plain English, making them easy to understand.
    - **Unlimited Users & Tests**: No restrictions on the number of users or tests.
    - **Multi-Platform Support**: Covers ***mobile, web, and API testing*** in ***one test case***.
    - **Programming Support**: Compatible with ***Python, Java, Ruby, JavaScript, PHP, and C#.***
    - **API Testing**: Supports ***audio*** testing, ***functional*** testing, and ***security*** testing.
    - **Integrations**: Integrates with tools like ***TestRail, Jira, and Azure DevOps.***
    - **Customer Support**: Available via ***contact form***.
    - **Platforms**: Supports ***iOS and Android***.
    - **Pricing**: Plans start at ***$900/month*** with a lifetime free public open-source plan.

**3. ACCELQ**

- **Overview**: ACCELQ provides ***AI-powered codeless test automation and management*** on a ***cloud-native*** platform.
- **Key Features**:

- **Codeless Automation**: Allows testing teams to automate without deep programming skills.
- **Unified Testing**: Supports ***mobile, web, API, database***, and packaged apps in ***one platform***.
- **Natural Language Processing**: Features a ***no-code editor*** for easy test creation.
- **AI-Powered Features**: Includes ***self-healing capabilities*** to reduce test flakiness.
- **Seamless CI/CD Integration**: Supports ***in-sprint automation*** for Agile development.
- **Customer Support**: Provides various support options.
- **Platforms**: Supports web on ***mobile, native iOS, native Android, and hybrid apps***.
- **Pricing**: Plans start at ***$150/month*** with a 14-day free trial (no credit card required).

**4. Katalon Platform**

- **Overview**: Katalon Platform simplifies mobile testing by providing a ***codeless experience built on Appium and Selenium.***
- **Key Features**:
  - **Easy Setup**: Simple test creation using ***record & playback, keywords, and images***.
  - **Cross-Platform Testing**: Execute tests on ***real devices, simulators, or cloud-based devices***.
  - **Test Reusability**: Supports reusability across ***mobile platforms, API, and web***.
  - **Integrations**: Built-in integration with ***project management tools*** like Jira and Git.
  - **Customer Support**: Available via ***chat, contact form, and email***.
  - **Platforms**: Supports ***iOS and Android***.
  - **Pricing**: Plans start at ***$29/month*** with a 16% discount on yearly payment.

**5. Perfecto**

- **Overview**: Perfecto is a leading ***testing cloud*** for mobile app testing, focusing on delivering ***exceptional digital experiences.***
- **Key Features**:
  - **Comprehensive Coverage**: Offers extensive platform and testing scenario coverage.
  - **Smart Analytics**: Provides analytics for faster feedback and fixes.
  - **Unified Cloud Platform**: Supports both web and mobile app testing.
  - **Enterprise-Grade Security**: Ensures security and scalability for large organizations.
  - **Customer Support**: Available via chat and contact form.
  - **Platforms**: Supports iOS and Android.
  - **Pricing**: Plans start at $99/month with a 16% discount on yearly payment.

**6. TestGrid**

- **Overview**: TestGrid allows users to perform manual and automated testing on real devices, hosted either on the cloud or on-premise.
- **Key Features**:
  - **User-Friendly**: Engages testing and business teams without requiring programming knowledge.

- **Test Reusability**: Allows reuse of tests across different app versions and apps.
- **Pricing**: Starts with a free plan and upgrades as low as $29/month.

**7. Appium**

- **Overview**: Appium is an open-source, cross-platform mobile testing tool for hybrid and native iOS and Android applications.
- **Key Features**:
  - **Cross-Platform Support**: Works with multiple programming languages like Java, Ruby, and C#.
  - **UI Automation**: Automates Android using the UIAutomator library and controls mobile browsers like Safari and Chrome.

**8. Selendroid**

- **Overview**: Selendroid is a test automation framework that drives the UI of Android native and hybrid applications.
- **Key Features**:
  - **Selenium Integration**: Tests are written using the Selenium 2 client API.

**9. Calabash**

- **Overview**: Calabash provides libraries that allow test code to interact programmatically with native and hybrid apps.
- **Key Features**:
  - **Cross-Platform Testing**: Supports both Android and iOS.

**10. KIF**

- **Overview**: KIF is an Objective-C based framework specifically for automated testing of iOS applications.
- **Key Features**:
  - **Integration with XCTest**: Integrates directly with XCTest for seamless testing.

**Conclusion**

These mobile app testing tools offer a range of features to facilitate both manual and automated testing across various platforms. Selecting the right tool depends on specific project needs, team expertise, and budget considerations. Each tool provides unique capabilities that cater to different aspects of mobile app testing, ensuring comprehensive coverage and effective quality assurance

**DEVOPS**

**Definition**

- **DevOps** is a cultural shift aimed at improving collaboration between development (Dev) and operations (Ops) teams.
- The primary goal is to enhance software delivery speed, quality, and responsiveness to business needs.

**Key Principles**

- **Not Just Tools**: DevOps is not a methodology or a suite of tools; it's a cultural transformation that promotes collaboration and continuous improvement.
- **Agility**: DevOps allows organizations to respond more quickly to changing business requirements by optimizing workflows, architecture, and infrastructure.

**Transition from Agile to DevOps**

- **Agile Manifesto Principles**:
    i. Individuals and interactions over processes and tools.
    ii. Working software over comprehensive documentation.
    iii. Customer collaboration over contract negotiation.
    iv. Responding to change over following a plan.
- **Agile vs. DevOps**:
    - **Agile** focuses on iterative development and frequent interaction between IT and business users.
    - **DevOps** extends Agile by integrating operations support into the development process, emphasizing the overall service delivered to customers.

**DevOps Pipeline Stages**

1. **Continuous Development**
    - Tools like **Jira** facilitate project management and track progress.
2. **Continuous Testing**
    - Tools like **Zephyr for Jira** manage test cases and integrate seamlessly with development workflows.
3. **Continuous Integration**
    - **Jenkins** automates testing and builds, allowing for early bug detection.
4. **Continuous Delivery/Deployment**
    - Tools like **Puppet** and **Chef** automate server configurations, ensuring consistency and avoiding "snowflake servers."
5. **Continuous Monitoring**
    - Tools like **Splunk** and **ELK Stack** monitor application performance and security post-deployment.

**DevOps Test Automation**

- Choosing the right automation tools is crucial for fostering collaboration and enhancing efficiency.
- **Key Tools**:
    - **Bamboo**: A CI/CD server integrated with Atlassian products.
    - **Selenium**: A suite for automated web application testing across browsers.
    - **TestComplete**: Comprehensive testing for web, mobile, and desktop applications.
    - **SoapUI**: Focused on API testing with user-friendly features.

**BIG DATA TESTING**

**Definition**

- **Big Data Testing** ensures that big data applications function correctly, maintaining performance and security while processing large datasets.

**Big Data Characteristics**

- **Volume**: Large amounts of data.

- **Variety**: Different types of data (structured, unstructured).
- **Velocity**: Speed of data processing.

**Big Data Testing Strategy**

- Focuses on verifying data processing rather than individual software features.
- **Key Components**:
    - **Performance Testing**: Ensures systems can handle data processing efficiently.
    - **Functional Testing**: Verifies that all functionalities work as expected.

**Phases of Big Data Testing**

1. **Data Staging Validation**:
    - Validate data from various sources (e.g., RDBMS, social media).
    - Ensure correct data extraction and loading into Hadoop.
2. **MapReduce Validation**:
    - Verify business logic across nodes.
    - Ensure correct data aggregation and segregation.
3. **Output Validation**:
    - Check that transformation rules are applied correctly.
    - Validate data integrity and successful loading into target systems.

**Architecture Testing**

- Essential for ensuring the system can handle large volumes of data.
- **Performance Testing**: Assesses job completion time, memory utilization, and data throughput.
- **Failover Testing**: Ensures data processing continues seamlessly during node failures.

**Performance Testing Approach**

1. **Cluster Setup**: Establish the Big Data cluster for testing.
2. **Workload Design**: Identify and design workloads for testing.
3. **Client Preparation**: Create custom scripts for testing.
4. **Execution and Analysis**: Run tests, analyze results, and tune components as necessary.
5. **Optimum Configuration**: Ensure the system is configured for optimal performance.

**Conclusion**

Both **DevOps** and **Big Data Testing** play crucial roles in modern software development and data management. DevOps fosters a culture of collaboration and rapid delivery, while Big Data Testing ensures that large datasets are processed accurately and efficiently. Understanding these concepts and their associated tools and strategies is essential for organizations looking to enhance their software delivery and data processing capabilities.

# MODULE 7

**1. Software Engineering Module 7: Product & Process Metrics, Quality Standards & Models**

**Overview**

This module focuses on the importance of metrics in software engineering, quality standards, and models that ensure high-quality software products. It emphasizes how these metrics and standards contribute to better processes and products.

**2. Product and Process Metrics**

**Metrics**

- **Definition**: Metrics are quantitative measures that help assess various attributes of software products and processes.
- **Importance**: They provide insights into performance, quality, and efficiency, enabling teams to make informed decisions.

**Framework for Product Metrics**

1. **Metrics**: The overall quantitative measures.
2. **Measures**: Specific quantitative indications of attributes, such as lines of code or function points.
3. **Measurement**: The process of quantifying these measures.
4. **Indicators**: Specific metrics that provide insights into the status of a software product or process.

**Measures**

- **Characteristics**: Measures give a quantitative indication of attributes like size, complexity, and performance.
- **Example Metrics**:
    - Lines of Code (LOC)
    - Cyclomatic Complexity
    - Function Points

**Product Example**

- **Hyundai Creta**:
    - **Price**: ₹14.92 lakhs
    - **Seating Capacity**: 5
    - **Fuel Economy**: 20 km/l
    - **Dimensions**: 4.3m (L) x 1.79m (W) x 1.63m (H)
    - **Fuel Tank Capacity**: 50 liters
- **Maruti Suzuki Baleno**:
    - **Price**: ₹9.03 lakhs
    - **Fuel Economy**: 24 km/l
    - **Dimensions**: 3.99m (L) x 1.74m (W) x 1.51m (H)

**Importance of Product & Process Metrics**

- **Assessment**: Metrics help software engineers assess the quality and performance of software products.
- **Design Insights**: They provide insights into design and construction aspects, allowing for better decision-making.

- **Quality Evaluation**: Metrics facilitate systematic quality assessments based on defined rules.

**3. Quality Standards & Models**

**Importance of Standards**

- **Best Practices**: Standards encapsulate best practices that enhance the quality assurance process.
- **Framework for Quality Assurance**: They provide a structured approach to implement quality assurance across various stages of the software lifecycle.
- **Continuity**: Standards help maintain consistency when different teams work on the same project.

**ISO (International Organization for Standardization)**

- **Definition**: ISO is a non-governmental organization that develops and publishes international standards.
- **ISO 9000 Family**: This family focuses on quality management systems, ensuring organizations meet customer and regulatory requirements.

**Eight Quality Management Principles**

1. **Customer Focus**: Prioritize customer needs and satisfaction.
2. **Process Approach**: Manage activities as processes to achieve desired results.
3. **Leadership**: Establish a clear vision and direction.
4. **System Approach to Management**: Integrate processes to achieve organizational objectives.
5. **Factual Approach to Decision-Making**: Use data and information for decision-making.
6. **Involvement of People**: Engage all employees in the quality process.
7. **Continual Improvement**: Focus on ongoing improvement in overall performance.
8. **Mutually Beneficial Supplier Relationships**: Foster relationships that enhance both parties.

**ISO 9000 Series**

- **ISO 9000**: Explains fundamental quality concepts and provides guidelines.
- **ISO 9001**: Model for quality assurance in design, development, production, installation, and servicing.
- **ISO 9002**: Quality assurance in production and installation.
- **ISO 9003**: Quality assurance in final inspection and testing.

**Advantages of ISO Certification**

- **Quality Maintenance**: Ensures consistent quality in products and services.
- **Market Credibility**: Enhances the organization's reputation and credibility.
- **Customer Confidence**: Builds trust and satisfaction among customers.

**4. Total Quality Management (TQM)**

**Definition**

- **TQM**: A comprehensive approach to improving the quality of products and services through continuous improvement, emphasizing customer satisfaction.

**Seven Quality Management Principles**

1. **Customer Focus**: Understanding and meeting customer needs.

2. **Leadership**: Creating a unity of purpose.
3. **Engagement of People**: Involving employees at all levels.
4. **Process Approach**: Managing activities as interconnected processes.
5. **Improvement**: Continuous enhancement of performance.
6. **Evidence-Based Decision Making**: Making decisions based on data analysis.
7. **Relationship Management**: Building mutually beneficial relationships with stakeholders.

**Stages of TQM**

- **Quality Control**: Monitoring processes to ensure quality.
- **Quality Assurance**: Systematic evaluation of quality systems.
- **Quality Improvement**: Continuous efforts to enhance quality.
- **Quality Management**: Overarching approach to managing quality.

**PDCA Cycle (Plan-Do-Check-Act)**

- **Plan**: Identify opportunities for improvement.
- **Do**: Implement changes.
- **Check**: Evaluate results.
- **Act**: Standardize successful changes.

**Implementing TQM**

1. **Plan**: Define the project, set targets, and analyze current processes.
2. **Do**: Execute corrective actions and provide training.
3. **Check**: Assess the effectiveness of changes.
4. **Act**: Document improvements and standardize processes.

**5. Six Sigma**

**Definition**

- **Six Sigma**: A disciplined, data-driven approach to eliminating defects and improving processes, focusing on reducing variability.

**History**

- **Origin**: Coined by Bill Smith at Motorola in the late 1970s.
- **Impact**: Motorola saved over $15 billion in the first decade of Six Sigma implementation.

**DMAIC Approach**

1. **Define**: Identify project goals and customer requirements.
2. **Measure**: Collect data on current processes.
3. **Analyze**: Investigate root causes of defects.
4. **Improve**: Implement solutions based on data analysis.
5. **Control**: Establish controls to sustain improvements.

**DMADV Approach**

1. **Define**: Set design goals based on customer needs.
2. **Measure**: Identify critical characteristics for quality.
3. **Analyze**: Develop design alternatives.
4. **Design**: Create improved designs.
5. **Verify**: Validate designs through pilot testing.

**Differences Between DMAIC and DMADV**

| DMAIC | DMADV |
|---|---|
| **Focuses on existing processes** | Focuses on new product designs |
| **Identifies root causes of defects** | Analyzes options to meet customer needs |
| **Implements improvements to reduce defects** | Designs models to fulfill customer requirements |

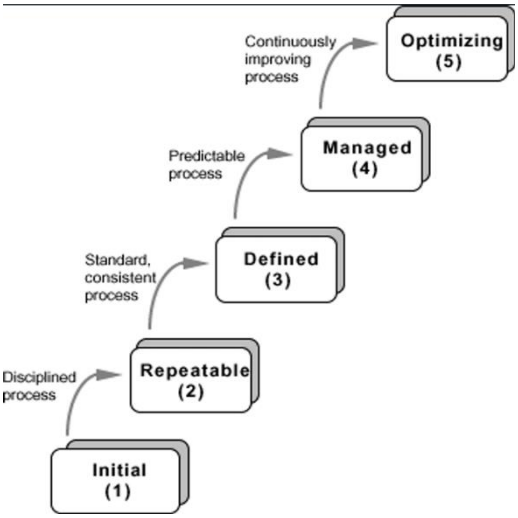
| Feature | DMAIC | DMADV |
|---|---|---|
| **Definition** | Improve existing processes | Design new processes/products |
| **Purpose** | Eliminate defects | Meet customer requirements from the start |
| **Steps** | Define, Measure, Analyze, Improve, Control | Define, Measure, Analyze, Design, Verify |
| **Focus** | Process improvement | Process/product design |
| **Application** | Existing processes | New processes/products |
| **Tools** | Statistical analysis, control charts | QFD, DOE, prototyping |

**6. Capability Maturity Model (CMM)**

**Definition**

- **CMM**: A framework for assessing and improving software processes, developed by the Software Engineering Institute.

**Maturity Levels**




1. **Initial**: Processes are unpredictable and reactive.
2. **Repeatable**: Basic project management processes are established.
3. **Defined**: Standardized processes are documented and followed.
4. **Managed**: Processes are measured and controlled.
5. **Optimizing**: Focus on continuous process improvement.

**CMM vs. CMMI**

- **CMM**: Focuses specifically on software engineering processes.
- **CMMI**: Broader framework that includes systems engineering and service delivery.

| Feature | CMM | CMMI |
|---|---|---|
| **Definition** | Focused on software processes | Broader scope including multiple disciplines |
| **Scope** | Software engineering | Systems, hardware, services, and software |
| **Structure** | Five maturity levels | Staged and continuous representations |
| **Flexibility** | Rigid structure | More flexible and tailored |
| **Focus Areas** | Software development | Product development, service delivery, project management |
| **Implementation** | Primarily in software organizations | Applicable across various industries |
| **Benefits** | Baseline for software improvement | Comprehensive process improvement framework |

**7. Summary**

- **Product Metrics**: Describe product characteristics such as size, complexity, and quality.
- **Process Metrics**: Focus on improving software development and maintenance.
- **ISO Standards**: Provide a framework for quality management across organizations.
- **TQM**: A continuous process aimed at improving quality and customer satisfaction.
- **Six Sigma**: A method to reduce defects and enhance process performance through data analysis.

**8. Hot Questions**

1. **ISO Certification**: What are the various ISO quality metrics available for a manufacturing company?
    - **Response**: ISO metrics include customer satisfaction, process efficiency, defect rates, and compliance with regulatory requirements. Organizations can use ISO 9001 to assess their quality management systems and identify areas for improvement.
2. **Six Sigma Application**: Discuss the application of Six Sigma in Samsung Electronics.
    - **Response**: Samsung uses Six Sigma to streamline its manufacturing processes, reduce defects in production, and enhance product quality. By applying DMAIC, Samsung focuses on understanding customer needs, measuring process performance, and implementing improvements to maintain high standards of quality across its product lines.

# MODULE 6

**Module 6: Software Evolution**

**Software Maintenance**

**Definition**

Software maintenance is the process of modifying and updating software after it has been deployed. This is crucial for ensuring that the software continues to meet user needs and remains functional in changing environments.

**Importance**

- **Continuous Improvement**: Software must evolve with technological advancements and changing user requirements to remain competitive.
- **Customer Satisfaction**: Regular updates and maintenance ensure that users have a positive experience and that their needs are met.

**Types of Software Maintenance**

1. **Corrective Maintenance**:
    - **Purpose**: Fixes defects and errors found in the software.
    - **Examples**: Bug fixes, addressing system crashes, and resolving performance issues.
    - **Process**: Often initiated by user-reported issues or internal testing.
2. **Preventative Maintenance**:
    - **Purpose**: Proactively addresses potential issues that could arise in the future.
    - **Examples**: Updating libraries, refactoring code, and optimizing performance.
    - **Process**: Involves regular reviews and updates to anticipate and mitigate future problems.
3. **Perfective Maintenance**:
    - **Purpose**: Enhances the software by adding new features or improving existing functionality based on user feedback.
    - **Examples**: Adding new user-requested features or removing outdated functionalities.
    - **Process**: Driven by user needs and market trends, often gathered through surveys or direct feedback.
4. **Adaptive Maintenance**:
    - **Purpose**: Modifies the software to keep it compatible with changing environments, such as new operating systems or hardware.
    - **Examples**: Updating software to run on a new version of an operating system or integrating with new hardware.
    - **Process**: Requires monitoring external changes and implementing necessary adjustments.

**Software Maintenance Process**

1. **Identification & Tracing**:
    - Determine which parts of the software require changes. This can come from user feedback, bug reports, or developer insights.
2. **Analysis**:
    - Assess the impact of proposed changes, including cost analysis and potential risks.
3. **Design**:
    - Create a design for the modifications, ensuring it aligns with existing architecture and user requirements.
4. **Implementation**:
    - Developers write and integrate the new code into the existing software.
5. **System Testing**:
    - Conduct tests to ensure that the changes work as intended and do not introduce new issues.

6. **Acceptance Testing**:
    - Users test the updated software to confirm that it meets their needs and expectations.
7. **Delivery**:
    - Release the updated software to users, often accompanied by documentation and support.

**Costs of Software Maintenance**

- **High Costs**: Maintenance can consume up to 50-70% of the total software lifecycle costs. Factors influencing costs include:
    - **Software Age**: Older software may require more extensive updates.
    - **Complexity**: More complex systems typically incur higher maintenance costs.
    - **Technology Changes**: Rapid advancements can necessitate frequent updates.

**Strategies for Effective Maintenance**

- **Documentation**: Maintain clear and up-to-date documentation to facilitate easier updates and modifications.
- **Quality Assurance (QA)**: Integrate QA processes early in the development cycle to catch potential issues before they escalate.

---

**Software Configuration Management (SCM)**

**Definition**

Software Configuration Management (SCM) is the discipline of managing changes to software products throughout their lifecycle. It ensures that changes are made systematically and that the integrity of the software is maintained.

**Importance**

- **Change Control**: Helps manage multiple versions and changes, preventing conflicts in collaborative environments.
- **Version Control**: Keeps track of different versions of software components, allowing teams to revert to previous versions if necessary.

**SCM Process**

1. **Identification of Objects**:
    - Define and catalog all software components and their relationships.
2. **Version Control**:
    - Manage different versions of configuration objects generated during the software process. Tools like Git are commonly used for this purpose.
3. **Change Control**:
    - Evaluate change requests to assess their impact and feasibility. This involves a formal process for submitting and reviewing change requests.
4. **Configuration Audit**:
    - Regular audits ensure that the software product meets baseline requirements and that all changes are documented.
5. **Status Reporting**:

- Provide accurate status updates on the current configuration and changes to stakeholders.

**Popular SCM Tools**

- **CFEngine**: Automates configuration for large systems.
- **Puppet**: Uses a declarative language for system configuration.
- **Chef**: Manages infrastructure as code across various environments.
- **Ansible**: An agentless tool for automation and orchestration.
- **Docker**: Containerization technology for deploying applications consistently across environments.

**Re-engineering**

**Definition**

Re-engineering is the process of redesigning existing software to improve its performance, maintainability, and adaptability without completely rewriting it.

**Importance**

- **Adaptation**: Helps software adapt to changing business needs and technologies.
- **Quality Improvement**: Identifies and rectifies defects, making the software more reliable.

**Steps in Re-engineering**

1. **Assessment**:
    - Analyze the existing software system to identify strengths, weaknesses, and areas for improvement.
2. **Planning**:
    - Develop a strategic plan to guide the re-engineering effort, including goals and performance metrics.
3. **Requirements Analysis**:
    - Define user needs and business requirements for the new or modified system.
4. **Architecture and Design**:
    - Create a new architecture that addresses the identified requirements while considering the limitations of the existing system.
5. **Implementation**:
    - Modify the existing system or develop a new system based on the new design.
6. **Testing**:
    - Conduct thorough testing to ensure that the re-engineered system meets the new requirements and is free from defects.
7. **Maintenance**:
    - Establish a maintenance plan to keep the re-engineered system reliable and up-to-date.

**Advantages of Re-engineering**

- **Reduced Risk**: Existing software has known issues, making it less risky than starting from scratch.
- **Cost-Effectiveness**: Generally cheaper than developing new software.
- **Improved Efficiency**: Streamlined processes can lead to significant productivity gains.

**Disadvantages of Re-engineering**

- **High Costs**: Can still be expensive, especially if extensive changes are required.
- **Resistance to Change**: Employees may be resistant to new processes or technologies.

**Reverse Engineering**

**Definition**

Reverse engineering is the process of deconstructing a software product to understand its design, architecture, and functionality. This can involve analyzing code to recover design specifications and requirements.

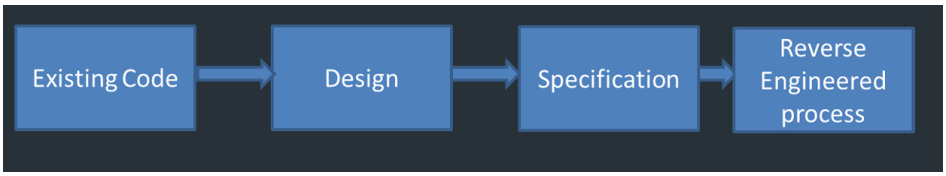
**Objectives**

- **Cost Reduction**: Identify cost-effective alternatives for components or systems.
- **Security Analysis**: Examine software for vulnerabilities to improve security measures.
- **Integration and Customization**: Modify existing systems to improve performance or tailor them to specific needs.

**Goals of Reverse Engineering**

1. **Cope with Complexity**: Helps engineers understand and manage complex systems.
2. **Recover Lost Information**: Aims to retrieve design details and specifications when documentation is unavailable.
3. **Detect Side Effects**: Analyzes unintended consequences and interactions within the system.

**Steps in Reverse Engineering**




1. **Collection of Information**: Gather all relevant documentation and source code.
2. **Examining the Information**: Study the collected data to understand the system.
3. **Extracting Structure**: Identify and document the program structure.
4. **Recording Functionality**: Document how each module operates.
5. **Recording Data Flow**: Create diagrams to show how data moves through the system.
6. **Recording Control Flow**: Document the high-level control structures.
7. **Review Extracted Design**: Ensure consistency and correctness of the extracted design.
8. **Generate Documentation**: Create comprehensive documentation for future reference.

**Tools for Reverse Engineering**

- **CIAO and CIA**: Tools for navigating software repositories.
- **Rigi**: A visual tool for understanding software.
- **Bunch**: A tool for software clustering and modularization.

**Software Reuse**

**Definition**

Software reuse involves leveraging existing software components to build new applications, reducing redundancy and development time.

**Advantages of Software Reuse**

- **Less Effort**: Utilizing existing components saves time and resources.
- **Time-Saving**: Ready-made components allow teams to focus on new functionality.
- **Cost Reduction**: Overall project costs decrease due to reduced development time.

**Types of Reusable Components**

- **Source Code**: Code libraries or modules that can be reused.
- **Design and Interfaces**: Existing design patterns and user interfaces.
- **Documentation**: User manuals and technical documentation.

**Commercial-off-the-shelf (COTS)**

COTS refers to ready-made software solutions that can be integrated into new systems, offering a quick way to implement functionality without starting from scratch.

**Reuse Software Engineering**

This involves guidelines and principles for effectively reusing software components, ensuring that reused elements are compatible and maintainable.

**SOFTWARE RE-ENGINEERING & REVERSE ENGINEERING**

**Software Re-Engineering**

**Definition**

- **Software Re-Engineering** is the process of examining and altering an existing software system to reconstitute it in a new form. This involves understanding the current system and making improvements or adaptations to enhance its performance, maintainability, and adaptability.

**Principles**

- The principles of re-engineering positively impact software cost, quality, customer service, and speed of delivery.

**Steps in Software Re-Engineering**

1. **Inventory Analysis**:
    - **Purpose**: Maintain a database of all developed applications.
    - **Contents**: The inventory includes details like application name, description, size, age, and other relevant metrics.
    - **Process**: Sort the inventory to identify candidates for re-engineering based on specific needs. Allocate resources accordingly for the selected candidates.
2. **Document Restructuring**:
    - **Importance**: Documentation is essential for explaining how the application operates and is used.
    - **Challenges**: Updating documentation can be time-consuming and hindered by inadequate resources.
    - **Requirement**: The re-developed system must be thoroughly documented to ensure clarity and usability.

3. **Code Restructuring**:
    - **Objective**: Improve the performance and reduce costs of legacy software.
    - **Process**:
        - Analyze source code using specialized tools to identify violations in programming constructs.
        - Rewrite the problematic code using modern programming languages.
        - Review and test the modified code to ensure it is free from errors.
4. **Data Restructuring**:
    - **Initiation**: Begins with reverse engineering activities.
    - **Analysis**: Examine the existing data architecture to identify weak data structures and re-engineer them.
    - **Impact**: Changes in data architecture can lead to significant architectural or code-level modifications.
5. **Forward Engineering**:
    - **Definition**: Also known as renovation or reclamation; it alters existing functionalities by adding new features.
    - **Time Consumption**: Takes more time than reverse engineering.
    - **Outcome**: The final product must be complete and precise.

**Reverse Engineering**

**Definition**

- **Reverse Engineering** is the process of analyzing a software system to extract design and implementation information, effectively working backward from the final product to the original design specifications.

**Objectives**

- To reconstruct existing objects and understand their functionality.
- To analyze the program to create higher levels of abstraction.

**Levels of Reverse Engineering**

1. **Abstraction Level**:
    - **Low Level**: Design representation.
    - **Higher Level**: Program and data structures.
    - **Relatively High Level**: Object models, data/control flow.
    - **High Level**: Entity-Relationship (ER) models.
2. **Completeness Level**:
    - Completeness decreases as the abstraction level increases. This means that higher levels of abstraction may not provide all the details found at lower levels.
3. **Directionality Level**:
    - Extracted information is directed towards developers to assist in understanding and modifying the software.

**Steps in Reverse Engineering**

1. **Understanding Data Structures**:
    - Analyze global and internal data structures to create a refined database model.
    - Identify relationships among data objects to inform the new schema.
2. **Defining New Database Model**:
    - Build an initial object model.
    - Identify candidate keys from attributes.
    - Refine tentative classes and identify generalizations.
    - Apply techniques to identify associations, leading to transformations that map the old database to the new one.
3. **Understanding Process**:
    - Extract procedural abstractions from various levels (system, component, pattern, statement).
    - Use block diagrams to represent interactions between functional abstractions.
    - Document the processing functionality of individual components.
4. **Reverse Engineering User Interfaces**:
    - Study the current user interface to inform the design of a new one.
    - Address key questions regarding actions (keystrokes, mouse clicks), system responses, and how to effectively replace the existing interface.

**Summary**

- **Software Re-Engineering** involves altering an existing system to enhance its form and functionality.
- The process includes inventory analysis, document restructuring, code restructuring, data restructuring, and forward engineering.
- **Reverse Engineering** starts from the final product and works backward to derive design specifications, focusing on understanding data structures, processes, and user interfaces.

**Software Maintenance**

**Definition**

**Software Maintenance** is the ongoing process of modifying a software system after it has been delivered to the customer. This process is essential for ensuring that the software continues to meet user needs and remains functional over time.

**Objectives of Software Maintenance**

- **Fixing Errors**: Addressing coding and design errors that were not identified during the initial development.
- **Enhancing Functionality**: Adding new features or improving existing ones to meet changing user requirements.
- **Adapting to Changes**: Modifying the software to accommodate changes in the external environment, such as new hardware or operating systems.

**Types of Changes**

1. **Coding Errors**: Mistakes in the program code that lead to incorrect behavior or system crashes.
2. **Design Errors**: Flaws in the software architecture or design that hinder performance or usability.

3. **Specification Errors**: Changes needed to align the software with updated user requirements.
4. **New Requirements**: Integrating additional functionalities based on evolving business needs.

**Maintenance Activities**

- Modifications are typically made by altering existing modules or introducing new ones.
- Maintenance tasks are usually handled by a specialized team known as the **Operations and Support** team, which focuses on ensuring the software's ongoing functionality and user satisfaction.

---

**Types of Software Maintenance**

**1. Corrective Maintenance**

- **Purpose**: To repair faults in the software.
- **Characteristics**:
    - Involves fixing bugs that affect the software's functionality.
    - Includes addressing coding errors, design flaws, and discrepancies in requirements.
- **Example**: A bug that causes an application to crash when processing certain inputs would be addressed through corrective maintenance.

**2. Adaptive Maintenance**

- **Purpose**: To modify the software to function in a new or changing environment.
- **Characteristics**:
    - Occurs when there are changes in the system environment, such as hardware upgrades or changes in operating systems.
    - Ensures that the software remains compatible with new technologies.
- **Example**: Updating an application to work with a new version of an operating system or integrating it with new hardware.

**3. Perfective Maintenance**

- **Purpose**: To enhance the software by adding new functionalities or improving existing features.
- **Characteristics**:
    - Necessary when organizational or business changes require adjustments to the software's capabilities.
    - Focuses on improving performance, usability, or efficiency.
- **Example**: Adding a reporting feature to a business application to meet new regulatory requirements.

---

**Cost Involved in Software Maintenance**

- Software maintenance often consumes a significant portion of the overall budget, sometimes exceeding the costs of initial development.

**Reasons for High Costs**

1. **Learning Curve**: Maintenance teams must invest time to understand the existing system architecture and codebase.

2. **Impact Analysis**: Assessing how proposed changes will affect the entire system requires thorough analysis and planning.
3. **Documentation**: Proper documentation is essential for maintenance activities, and creating or updating this documentation can be time-consuming.

**Best Practices to Reduce Maintenance Costs**

- **Precise Specification**: Clearly defined requirements help minimize misunderstandings and errors during development.
- **Object-Oriented Development**: Promotes reusability of components, making future maintenance easier and less costly.
- **Configuration Management**: Helps track changes systematically, reducing the likelihood of introducing new errors during maintenance.

---

**Maintenance Prediction**

**Importance**

Predicting future maintenance needs is crucial for effective software management. It helps organizations prepare for upcoming changes and allocate resources efficiently.

**Factors Influencing Maintenance Prediction**

1. **System Complexity**: The number and complexity of system interfaces can affect how easily changes can be made.
2. **Volatility of Requirements**: Systems with frequently changing requirements may require more maintenance.
3. **Business Processes**: Understanding how the software interacts with business processes helps identify potential areas for change.

---

**Software Configuration Management**

**Definition**

**Configuration Management (CM)** is the discipline within software engineering that focuses on managing changes to software systems. It ensures that all changes are made systematically and that the system remains consistent and reliable.

**Objectives of Configuration Management**

- Maintain integrity and traceability of the system throughout its lifecycle.
- Ensure that all team members have access to the most current version of the software.
- Facilitate collaboration among multiple developers working on the same project.

---

**Configuration Management Activities**

**1. Change Management**

- **Definition**: The process of managing changes to the software system in a controlled manner.
- **Activities**:
    - **Change Request Analysis**: Evaluating the costs and benefits of proposed changes.

- **Approval Process**: Ensuring that changes are reviewed and approved before implementation.
- **Tracking Changes**: Keeping records of which components have been modified and the nature of those changes.

**2. Version Management**

- **Definition**: The process of tracking different versions of software components to ensure consistency.
- **Key Concepts**:
  - **Codeline**: A sequence of versions of source code where later versions are derived from earlier ones.
  - **Baseline**: A reference point for a specific version of a software component, representing a stable state of the system.

**3. System Building**

- **Definition**: The process of creating a complete, executable system from its components.
- **Involves**:
  - **Development System**: The environment where development takes place.
  - **Build Server**: A dedicated server that compiles and links the necessary components.
  - **Target Environment**: The environment where the final system will be deployed.

**4. Release Management**

- **Definition**: The process of delivering a version of software to customers.
- **Types of Releases**:
  - **Major Release**: Introduces significant new functionalities and enhancements.
  - **Minor Release**: Primarily focused on bug fixes and minor improvements.

**Components of a Software Release**

- Configuration files.
- Data files.
- Installation procedures.
- Release notes detailing changes and updates.

**Responsibilities in Release Management**

- **Release Planning**: Deciding on release dates and preparing documentation.
- **Customer Communication**: Keeping customers informed about new releases and changes.
- **Issue Resolution**: Quickly addressing problems that arise during or after a release.

**Summary**

- **Software Maintenance** is a critical aspect of software engineering, often consuming more resources than initial development.
- The three main types of maintenance—Corrective, Adaptive, and Perfective—address different needs and challenges.
- **Configuration Management** ensures that all changes are managed effectively, maintaining the integrity of the software system and facilitating collaboration among developers.

# MODULE 3

**Module 3: Modelling Requirements**

**Software Requirements**

**Definition**

Software requirements are ***detailed descriptions of the services*** a software system should provide, along with its ***operational constraints***. They serve as a ***contract between stakeholders and developers***, ensuring that everyone has a shared understanding of ***what the software will deliver***.

**Importance**

- **Foundation of Development**: They ***guide*** the entire software development lifecycle (***SDLC***), influencing design, implementation, testing, and maintenance.
- **Stakeholder Alignment**: Clear requirements help ***align the expectations of various stakeholders***, ensuring that the final product meets their needs.
- **Risk Management**: Well-defined requirements can ***reduce the risk of project failure*** by identifying potential issues early in the development process.

**Role**

- **Guidance**: Requirements inform ***design and implementation decisions***.
- **Validation**: They provide a basis for testing and validation, ensuring the ***final product meets specified criteria***.
- **Documentation**: Serve as a ***reference point*** throughout the project lifecycle, aiding in communication and understanding.

**Need for Gathering Requirements**

**Key Reasons**

1. **Initial Communication**: Establishes the first line of communication between customers/stakeholders and project developers/testers, ***ensuring that everyone is on the same page***.
2. **Thorough Documentation**: ***Captures customer needs*** comprehensively to avoid missing critical aspects.
3. **Effectiveness**: Effective requirements gathering ***minimizes misunderstandings*** and miscommunications.
4. **Process**: Involves a ***systematic approach*** that includes:
    - **Fixing Appointments**: Scheduling meetings with stakeholders.
    - **Interviews**: Conducting detailed discussions to extract information.
    - **Elicitation**: Gathering requirements through various techniques.
    - **Documentation**: Recording the gathered requirements for future reference.

**Problems Faced in Gathering Requirements**

**Common Issues**

- **Incomplete Requirements**: Missing essential details can lead to significant issues later in development.
- **Ambiguous Requirements**: Vague language can result in ***different interpretations*** among stakeholders.
- **Changing Requirements**: Evolving needs can disrupt the development process and lead to ***scope creep***.

- **Conflicting Requirements**: Different stakeholders may have ***opposing needs,*** leading to conflicts.
- **Lack of Stakeholder Involvement**: Insufficient engagement can result in ***missed requirements***.
- **Scope Creep**: Uncontrolled changes or ***continuous growth in project scope*** can lead to ***project delays*** and ***increased costs***.
- **Technological Constraints**: Limitations in technology can ***affect what is feasible***.
- **Unclear Prioritization**: Without clear priorities, ***critical requirements may be overlooked***.
- **Communication Challenges**: Miscommunication can lead to ***misunderstandings and errors***.
- **Amalgamation Problem**: Combining ***multiple requirements into one*** can obscure ***individual needs***.

**Solutions to Problems in Gathering Requirements**

**Recommended Approaches**

1. **Structured Language Specification**: Use ***clear, unambiguous language*** to describe requirements.
2. **Standardized Writing**: Write requirements in a ***consistent format*** to enhance clarity and understanding.
3. **Uniformity**: Maintain uniformity in ***terminology and structure*** across all requirements.
4. **Common Templates**: Utilize templates to ***streamline the documentation process*** and ensure ***completeness***.

**Types of Requirements (Abstract Level)**

**User Requirements**

- **Definition**: High-level descriptions of ***what users need*** from the system.
- **Characteristics**:
    - ***First attempt*** to describe requirements.
    - Focus on ***services and constraints***.
    - Written in ***natural language*** or represented through ***diagrams***.
    - Must be ***understandable*** by all stakeholders.
    - Align with ***business objectives***.

**System Requirements**

- **Definition**: Detailed descriptions of ***system functionalities and constraints***.
- **Characteristics**:
    - Provide a ***comprehensive view*** of what the system should do.
    - Useful for ***design and development phases***.
    - Must be ***precise*** and ***cover all possible cases***.
    - Presented in a ***structured format*** for clarity.

**System Requirements Gathering**

**Key Elements**

- **Function**: What the system should do.
- **Description**: Detailed explanation of functionalities.

- **Inputs**: Data that the system receives.
- **Source**: Origin of the inputs.
- **Outputs**: Data produced by the system.
- **Destination**: Where the outputs go.
- **Action**: Specific ***operations*** the system performs.
- **Precondition & Post-condition**: Conditions that must be met before and after a function is executed.
- **Side Effects**: Any additional effects resulting from the function.

---

**Requirements Attributes**

| Attribute | Description |
|---|---|
| **Requirement ID** | Unique identifier for each requirement. |
| **Requirement Name** | Short, descriptive name for the requirement. |
| **Priority** | Importance level (e.g., high, medium, low). |
| **Status** | Current state (e.g., draft, approved, implemented). |
| **Source** | Origin of the requirement (e.g., stakeholder, regulatory). |
| **Dependency** | Other requirements that this requirement depends on. |
| **Created On** | Date the requirement was created. |
| **Created By** | Individual or team who created the requirement. |

---

**Types of Requirements**

**Functional Requirements**

- **Definition**: Statements that specify what the ***system should do***.
- **Characteristics**:
    - Describe specific ***behaviors*** in response to particular inputs.
    - Outline the system's ***functionalities and services***.
    - Examples include user authentication, data processing, and reporting features.

**Non-Functional Requirements**

- **Definition**: ***Constraints*** on the services or functions offered by the system.
- **Characteristics**:
    - Include ***performance metrics, security standards, usability,*** and ***compliance***.
    - ***Apply to the system as a whole*** rather than to specific functionalities.
    - Examples include response time, reliability, and scalability.

**Non-Functional Classifications**

**Categories**

- **Product**: Attributes related to the product itself (e.g., ***usability, reliability***).
- **Organizational**: Constraints imposed by the organization (e.g., ***policies, procedures***).
- **External**: Factors outside the organization (e.g., ***legal, regulatory*** requirements).

**Specific Non-Functional Requirements**

- **Usability**: ***Ease of use*** and user satisfaction.
- **Efficiency**: Performance metrics such as ***response time*** and ***resource utilization***.
- **Reliability**: System ***availability*** and ***error rates***.
- **Portability**: Ability to transfer the system to ***different environments***.

**Metrics for Specifying Non-Functional Requirements**

| Property | Measure |
|---|---|
| **Speed** | Processed transactions/second |
| | User/event response time |
| | Screen refresh time |
| **Size** | Mbytes |
| | Number of ROM chips |
| **Ease of Use** | Training time |
| | Number of help frames |
| **Reliability** | Mean time to failure |
| | Probability of unavailability |
| | Rate of failure occurrence |
| **Availability** | Robustness |
| | Time to restart after failure |
| | Probability of data corruption on failure |
| **Portability** | Percentage of target dependent statements |
| | Number of target systems |

**Domain Requirements**

**Definition**

Domain requirements are specific to the ***application area of the system*** and reflect the unique ***characteristics*** and ***constraints*** of that domain.

**Domain Requirements Problems**

- **Understandability**: ***Domain-specific language*** may not be easily understood by software engineers.
- **Implicitness**: ***Domain experts*** may assume knowledge and ***fail to document*** important requirements explicitly.

**Key Points**

- Requirements define what a software system should do and set constraints on its operation and implementation.
- Functional requirements specify the services the system must provide, while non-functional requirements impose constraints that often relate to the system's overall quality and performance.
- Understanding and managing requirements is critical for project success.

**Requirements Analysis and Documentation**

**Techniques for Analysis**

- **Refinement**: Techniques to analyze and refine requirements to ensure ***clarity and completeness***.
- **Documentation**: Using appropriate ***templates*** to ensure that requirements are ***well-organized and accessible***.
- **Change Management**: ***Establishing processes*** for managing ***changes to requirements*** as the project evolves.
- **Tools**: Utilizing software tools for requirements management to ***streamline the process***.

---

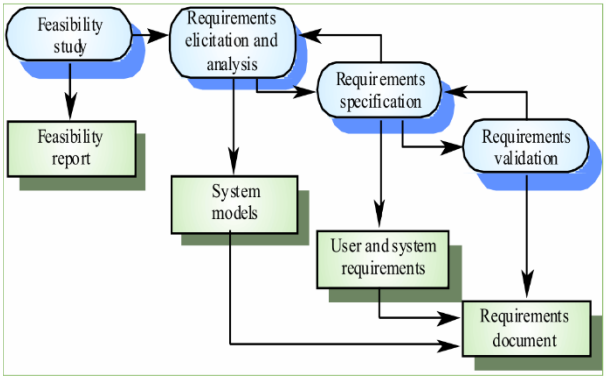
**Requirements Engineering Process**

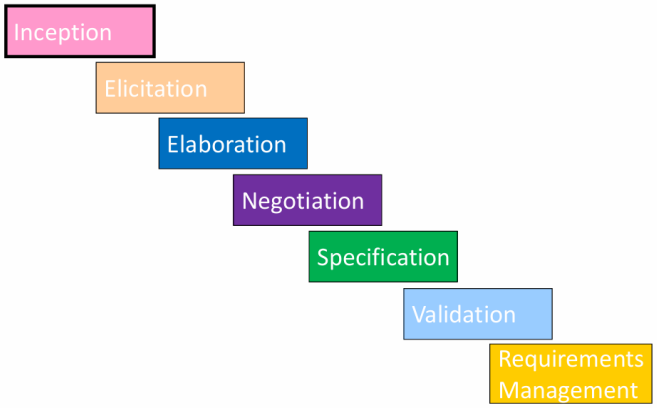
**Overview**

The requirements engineering process encompasses the activities involved in ***discovering, analyzing, and validating system requirements.***

**Generic Activities**

1. **Requirements Elicitation**: ***Gathering requirements*** from stakeholders through various techniques.
2. **Requirements Analysis**: Analyzing the gathered requirements for ***clarity, completeness, and feasibility***.
3. **Requirements Validation**: Ensuring that ***requirements meet stakeholder needs*** and are ***feasible***.
4. **Requirements Management**: ***Keeping track*** of requirements ***changes*** and maintaining ***documentation***.




**Requirements Engineering Tasks**

**Stages of the Process**

1. **Inception**: Establishing a ***basic understanding of the problem*** and ***who needs the solution***.
2. **Elicitation**: ***Gathering detailed requirements*** from stakeholders.
3. **Elaboration**: ***Refining and modeling*** requirements to create a ***clear specification***.

4. **Negotiation**: ***Resolving conflicts*** and reaching agreements on requirements.
5. **Specification**: Documenting requirements in a Software Requirements Specification ***(SRS) document***.
6. **Validation**: Checking the ***correctness and completeness*** of documented requirements.
7. **Requirements Management**

**Requirements Elicitation Techniques**

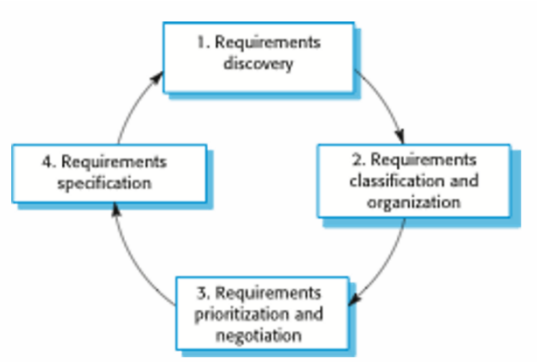
**Common Techniques**

- **Interviews**: One-on-one discussions to gather detailed insights.
- **Surveys/Questionnaires**: Structured forms to collect information from a larger audience.
- **Observations**: Watching users interact with current systems to identify needs.
- **Workshops and Focus Groups**: Collaborative sessions to gather diverse input.
- **Prototyping**: Creating mock-ups to visualize requirements and gather feedback.
- **Brainstorming**: Generating ideas in a group setting.
- **FAST (Facilitated Application Specification Technique)**: A structured approach to gathering requirements.
- **QFD (Quality Functional Deployment)**: A methodology to transform customer needs into engineering characteristics.
- **Document Analysis**: Reviewing existing documentation for relevant information.
- **Stakeholder Analysis and Management**: Identifying and managing stakeholder expectations and needs.

**Requirements Elicitation and Analysis**

**Process Activities**

1. **Requirements Discovery**: Interacting with stakeholders to uncover their needs.
2. **Requirements Classification and Organization**: Grouping related requirements into coherent clusters.
3. **Prioritization and Negotiation**: Determining the importance of requirements and resolving conflicts.
4. **Requirements Specification**: Documenting requirements for further development and validation.




**Challenges in Requirements Analysis**

- ***Stakeholders may not know what they want*** or may express requirements in their ***own terms***.
- ***Conflicting requirements*** may arise from different stakeholders.
- ***Organizational and political factors*** can influence requirements.
- Requirements may ***change during the analysis*** process due to ***emerging needs*** or environmental changes.

# MODULE 4

**Module 4: Software Design**

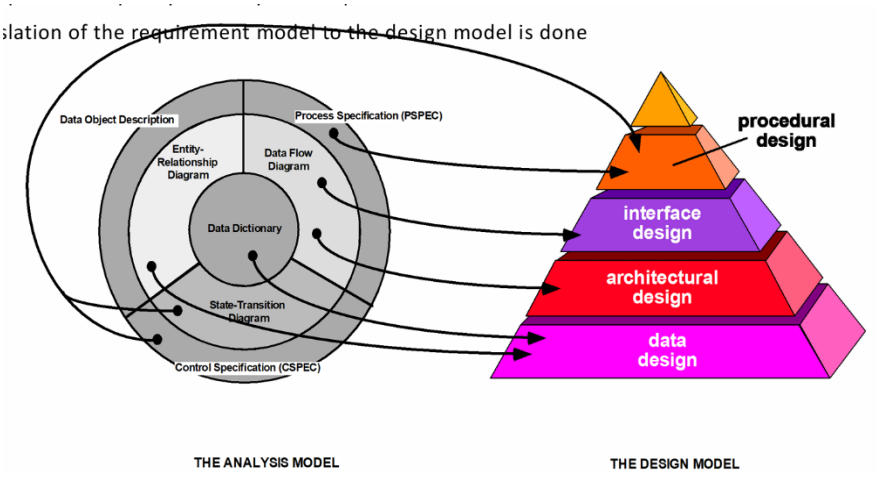
**Topics Covered**

- Design Concepts and Principles
- Architectural Design
- Detailed Design
- Object-Oriented Design
- User Interface Design

**Design Concepts & Principles**

**Objectives of Software Design**

- **Quality Improvement**: Enhance the quality of the software system or product by applying best practices and principles.
- **Translation of Requirements**: Convert the requirement model into a design model, ensuring that all requirements are effectively addressed.




**Characteristics of a Good Software Design Document (SDD)**

- **Completeness**: The SDD must include ***all requirements specified*** in the Software Requirements Specification ***(SRS).***
- **Understandability**: The document should be ***easy to read and maintain***, allowing both technical and non-technical ***stakeholders*** to comprehend it.
- **Comprehensive Overview**: It should provide a complete picture of the ***data, functions, and behavioral domains*** of the system.

**Design Principles**

1. **No Tunnel Vision**: Avoid focusing too narrowly on a single aspect; ***consider the broader context*** and how different components interact.
2. **Traceability**: Ensure that designs can be traced back to the analysis model, facilitating ***verification and validation.***
3. **Reuse Existing Solutions**: Leverage existing solutions and frameworks instead of reinventing the wheel, which ***saves time and effort***.
4. **Minimize Intellectual Distance**: Keep designs ***understandable*** and ***relatable*** to stakeholders, reducing the gap between technical and non-technical perspectives.
5. **Uniformity and Integrity**: Maintain ***consistent design practices*** across the project and ensure that all parts of the system work together ***cohesively***.
6. **Quality Assessment**: Designs should undergo assessments and reviews to identify and ***minimize conceptual errors, ensuring high quality***.

**Fundamental Concepts of Software Design**

**1. Abstraction**

- **Definition**: ***Simplifying complex systems*** by focusing on high-level functionalities while ***hiding lower-level*** details. For instance, a car's steering wheel abstracts the complexities of the steering mechanism.
- **Purpose**: Helps in managing complexity and ***allows designers to focus on essential aspects***.

**2. Refinement**

- **Definition**: ***Breaking down high-level abstractions*** into more detailed components. This process involves iteratively detailing the system until it is ready for implementation.
- **Example**: Starting with a general requirement like "manage user accounts" and refining it into specific functionalities like "create account," "delete account," and "update account details."

**3. Modularity**

- **Definition**: Dividing a system into ***smaller, manageable modules*** that can be developed and tested ***independently.***
- **Benefits**:
    - **Ease of Development**: Teams can work on different modules ***simultaneously***.
    - **Flexibility**: Modules can be ***modified*** with ***minimal impact on others***.
    - **Reusability**: Modules can often be ***reused in different projects***.

**4. Architecture**

- **Definition**: The ***overall structure*** of the system, including ***components***, their ***interactions***, and the ***principles*** governing their design.
- **Importance**: A well-defined architecture ***facilitates communication*** among stakeholders and serves as a ***blueprint*** for development.

**5. Information Hiding**

- **Definition**: Restricting access to certain details of modules to ***reduce complexity*** and ***increase security***.
- **Controlled Interfaces**: Modules should expose only what is necessary, ***hiding implementation details*** from clients.

**6. Refactoring**

- **Definition**: Improving the ***internal structure of existing code*** without changing its external behavior.
- **Purpose**: To enhance code ***readability and maintainability***, reducing ***technical debt***.

**Modularity**

**Benefits of Modular Design**

- **Ease of Development**: Modules can be developed ***independently***, making it easier to build and maintain the system.
- **Flexibility**: Changes in one module can be made with minimal impact on others.
- **Reusability**: Modules can be reused in different projects, saving time and resources.
- **Testing**: Individual modules can be tested independently, simplifying the testing process.

**Key Metrics**

- **Cost of Software Development**: The ***optimal number*** of modules can ***minimize development costs*** while ***maximizing efficiency***.

- **Number of Modules**: Determining the right number of modules is crucial; too few can lead to ***monolithic designs,*** while too many can increase complexity.

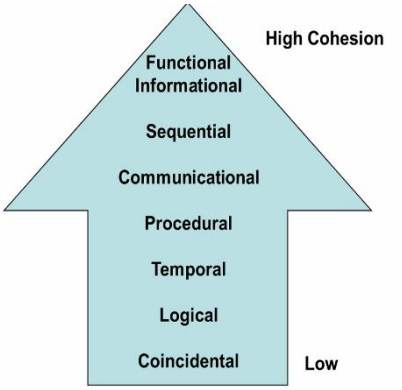
**Information Hiding**

- **Controlled Interfaces**: Modules should expose only what is necessary, hiding implementation details from clients.
- **Secret Details**: Clients should not need to know how a module works internally, only how to interact with it.

**Cohesion and Coupling**

**Cohesion**

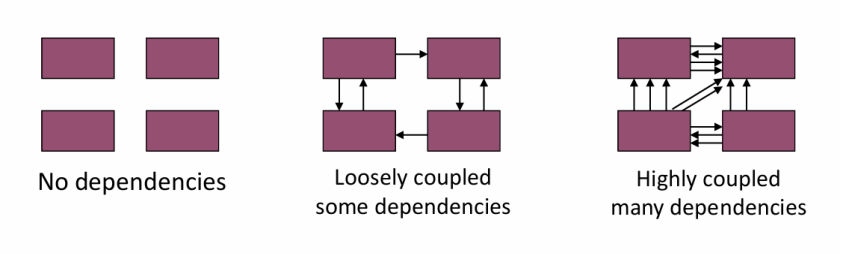
- **Definition**: The degree to which elements ***within a module*** belong together. ***High cohesion is desirable*** as it indicates that a module has a single, well-defined purpose.
- **Types of Cohesion**:




    1. **Coincidental Cohesion**: Parts are ***unrelated***; the worst form of cohesion.
    2. **Logical Cohesion**: Elements are ***related logically*** rather than functionally (e.g., a module containing functions that handle different types of input).
    3. **Temporal Cohesion**: Elements are grouped by ***when they are processed*** (e.g., initialization routines).
    4. **Procedural Cohesion**: Elements are related by ***the sequence of execution*** (e.g., a module that handles user authentication and session management).
    5. **Communicational Cohesion**: Elements ***operate on the same data*** (e.g., processing customer information).
    6. **Sequential Cohesion**: The output from one part ***serves as input to another*** (e.g., data processing stages).
    7. **Functional Cohesion**: All elements ***contribute to a single, well-defined task***; the ideal situation (e.g., a module that calculates total sales).

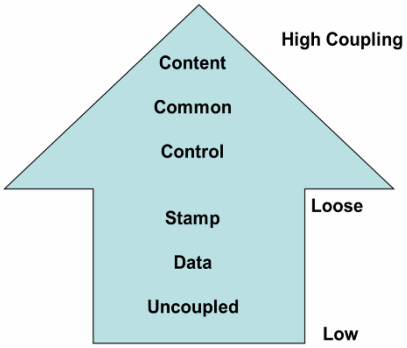
**Coupling**

- **Definition**: The degree of dependence ***between modules***. ***Lower coupling is generally preferred,*** as it indicates that modules are more independent and changes in one will have less impact on others.




- **Types of Coupling**:
    1. **Data Coupling**: Modules communicate through ***parameters***; the most desirable form.

2. **Stamp Coupling**: Modules ***share a data structure*** without needing full access to it.
3. **Control Coupling**: Modules pass ***control parameters***; can be good or bad depending on context.
4. **Common Coupling**: Multiple modules ***share global data***; usually poor design due to unclear responsibilities.
5. **Content Coupling**: One module ***modifies another***; the worst form of coupling.




**Architectural Design**

**Purpose**

- **Component Description**: Defines the ***components*** of the system and their ***interconnections***.
- **Effectiveness Analysis**: Evaluates the design's effectiveness and ***explores alternatives***.
- **Risk Reduction**: ***Identifies*** and ***mitigates potential risks*** in the design.

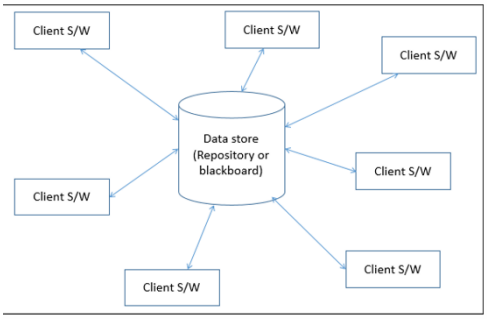
**Architectural Styles in Software Design**

Architectural styles define the fundamental organization of a software system. Each style offers a unique way to structure components and their interactions, influencing how the system is designed, implemented, and maintained. Here's a detailed explanation of the five key architectural styles:

**1. Data-Centric Architecture**

**Overview**

- **Focus**: This architecture emphasizes the ***management of data***, with ***data storage*** and access patterns being ***central*** to the system's design.
- **Components**: Typically involves databases as the core components, with various applications and services interacting with the data.




**Characteristics**

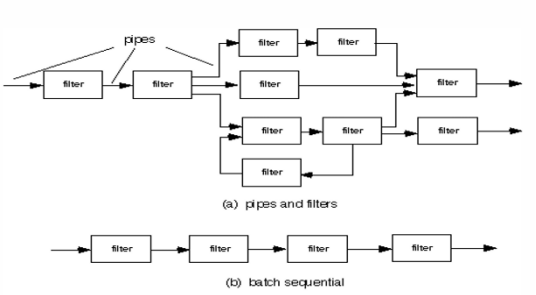
- **Data as a Central Element**: The architecture revolves around data, which is often stored in ***relational or non-relational databases.***
- **Data Access Layer**: A dedicated layer manages data access, ensuring that various components can retrieve and manipulate data without direct coupling to the underlying data storage.
- **Separation of Concerns**: Business logic is separated from data management, promoting modularity and ease of maintenance.

**Use Cases**

- **Enterprise Applications**: Systems that require robust data management, such as Customer Relationship Management (CRM) systems.
- **Reporting Systems**: Applications that focus on data analysis and reporting, where data integrity and access speed are crucial.

**2. Data Flow Architecture**

**Overview**




- **Focus**: This architecture emphasizes the ***movement and transformation of data*** through the system, with components processing data streams in a ***sequential or parallel*** manner.
- **Components**: Comprises data sources, processing nodes, and data sinks.

**Characteristics**

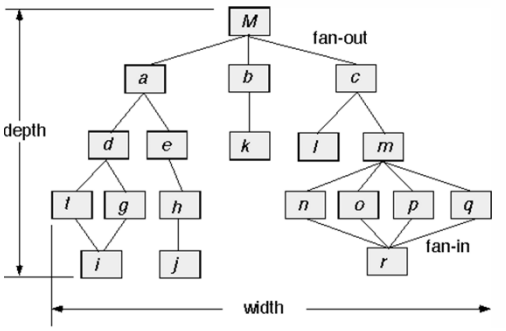
- **Stream Processing**: Data flows through a series of processing elements (components), each transforming the data as it passes through.
- **Loose Coupling**: Components are loosely coupled, allowing them to operate independently, which enhances scalability and flexibility.
- **Asynchronous Communication**: Often employs asynchronous communication to handle data streams efficiently, allowing for real-time processing.

**Use Cases**

- **Real-Time Data Processing**: Systems like financial trading platforms or live analytics dashboards that require immediate processing of incoming data.
- **Event-Driven Architectures**: Applications that react to events or data changes, often used in IoT systems.

**3. Call and Return Architecture**

**Overview**




- **Focus**: Based on function calls and returns, this architecture is common in procedural programming paradigms.
- **Components**: Consists of functions or procedures that are called by other components or modules.

**Characteristics**

- **Hierarchical Structure**: Functions are organized hierarchically, with high-level functions calling lower-level ones, creating a clear flow of control.
- **Synchronous Communication**: Typically involves synchronous calls where the caller waits for the callee to finish execution before proceeding.
- **Reusability**: Functions can be reused across different parts of the application, promoting code reuse.

**Use Cases**

- **Traditional Procedural Applications**: Suitable for applications where the logic can be expressed as a series of steps or procedures, such as command-line tools or simple desktop applications.

---

**4. Object-Oriented Architecture**

**Overview**

- **Focus**: Structures the system around objects, which encapsulate both state (data) and behavior (methods).

- **Components**: Consists of classes and objects that interact with one another through defined interfaces.

**Characteristics**

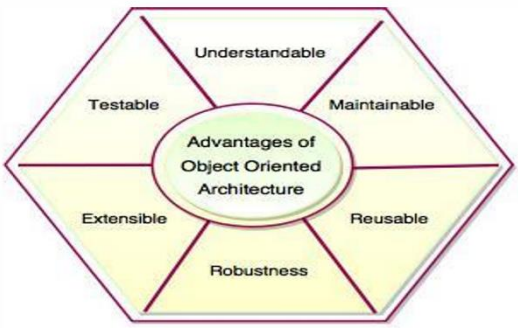


Fig. Advantages of Object Oriented Architecture

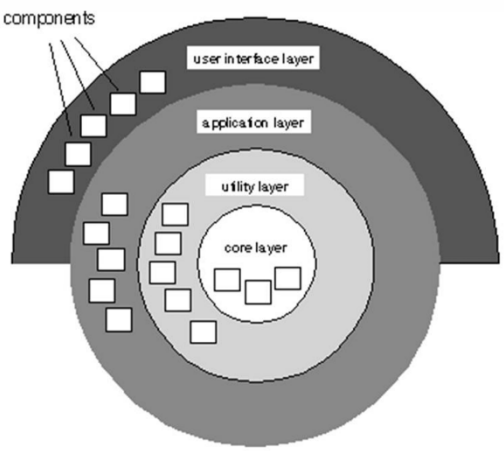
- **Encapsulation**: Objects hide their internal state and expose only necessary methods, promoting modularity and reducing complexity.
- **Inheritance**: Allows for the creation of new classes based on existing ones, facilitating code reuse and extension.
- **Polymorphism**: Enables objects of different classes to be treated as objects of a common superclass, enhancing flexibility.

**Use Cases**

- **Complex Systems**: Ideal for large systems where the model closely resembles real-world entities, such as graphical applications, simulations, and enterprise systems.

---

**5. Layered Architecture**




**Overview**

- **Focus**: Organizes the system into layers, each with specific responsibilities, promoting separation of concerns.
- **Components**: Typically includes presentation, business logic, and data access layers.

**Characteristics**

- **Layered Structure**: Each layer is responsible for a specific aspect of the application, such as user interface, business rules, or data management.
- **Communication**: Layers communicate with each other in a defined manner, usually with higher layers calling services provided by lower layers.
- **Modularity**: Changes in one layer can often be made independently of others, facilitating maintenance and updates.

**Use Cases**

- **Web Applications**: Commonly used in web applications where different layers handle user interaction, application logic, and data storage.
- **Enterprise Systems**: Useful in large-scale enterprise applications that require clear separation between different functional areas.

---

**Summary**

Understanding these architectural styles is crucial for designing systems that are efficient, maintainable, and scalable. Each style has its strengths and is suited for different types of applications, so selecting the appropriate architecture based on project requirements is essential for successful software development.

**Detailed Design**

**Definition**

The process of refining and expanding the preliminary design to ensure it is complete enough for implementation.

**Major Tasks**

1. **Understanding Architecture and Requirements**: Focus on requirements specific to components, ensuring that each design aligns with overall system architecture.
2. **Creating Detailed Designs**: Involves:
    - **Interface Design**: Both internal and external interfaces should be clearly defined.
    - **Graphical User Interface (GUI) Design**: Focus on user experience and interaction.
    - **Internal Component Design**: Define the structure and behavior of individual components.
    - **Data Design**: Specify how data will be stored, accessed, and manipulated.
3. **Evaluating Designs**: Use technical reviews to assess design quality, ensuring alignment with requirements and architecture.
4. **Documenting Software Design**: Capture designs in a Software Design Document (SDD), which serves as a reference throughout the development process.
5. **Monitoring Implementation**: Ensure that implementation aligns with the detailed design, making adjustments as necessary.

**Deriving Program Architecture**

**Overview**

Deriving program architecture involves creating a structured framework for software systems, enhancing modularity and maintainability.

---

**Partitioning the Architecture**

**Types of Partitioning**

1. **Horizontal Partitioning**:
    - **Definition**: Divides the architecture into layers (e.g., presentation, business logic, data access).
    - **Benefits**: Promotes separation of concerns and easier management.
2. **Vertical Partitioning**:
    - **Definition**: Divides architecture based on functionalities (e.g., user management, product catalog).
    - **Benefits**: Allows focused development and testing of individual components.

---

**Why Partitioned Architecture?**

- **Easier Testing**: Smaller components are easier to test independently.
- **Easier Maintenance**: Changes in one part have minimal impact on others.
- **Fewer Side Effects**: Isolated components reduce unintended consequences.
- **Easier Extension**: New features can be added without disrupting existing functionality.

---

**Structured Design**

**Objective**

To derive a partitioned program architecture for clarity and maintainability.

**Approach**

1. **Mapping DFD to Architecture**: Translate processes from a Data Flow Diagram (DFD) into modules.
2. **Process Specification (PSPEC)**: Describe module inputs, outputs, and operations.
3. **Structure Text (STD)**: Document the structure and behavior of modules.

**Notation: Structure Chart**

- **Definition**: A diagram showing hierarchical relationships between modules.
- **Components**: Modules (rectangles) and connections (lines) indicating control and data flow.

**Transaction and Transform Analysis**

**Transaction Analysis**

- **Purpose**: Identifies processes that act as transaction centers, routing data without transforming it.
- **Steps**:
    i. **Review the System Model**: Understand the overall system architecture and requirements.
    ii. **Refine the DFD**: Ensure clarity in the data flow diagram (DFD) for the software.
    iii. **Identify Transaction Centers**: Determine processes that route data to multiple downstream processes.
    iv. **Map the DFD**: Create a program structure that reflects the transaction flow.

**Transform Analysis**

- **Purpose**: Converts DFDs into structure charts, focusing on data transformation.
- **Steps**:
    i. **Identify Central Functions**: Determine key functions in the DFD that perform transformations.
    ii. **Create a First-Cut Structure Chart**: Develop an initial structure chart based on the DFD.
    iii. **Refine the Structure Chart**: Improve the initial design to enhance clarity and efficiency.
    iv. **Verify Alignment**: Ensure that the final structure chart meets the requirements of the original DFD.

---

**Detailed Design – Refactoring**

**Definition**

Refactoring is the process of restructuring existing code without changing its external behavior. It aims to improve code readability, maintainability, and performance.

**Motivation for Refactoring**

- **Code Smells**: Identifying problematic patterns in code, such as long methods, duplicated code, or complex structures.

- **Maintainability**: Easier to fix bugs and understand the code when it is well-structured and readable.
- **Extensibility**: Facilitates the addition of new features and enhancements without extensive rework.

**Benefits of Refactoring**

1. **Maintainability**: Simplifies the codebase, making it easier to understand and modify.
2. **Extensibility**: Allows for easier addition of new functionalities, promoting a flexible architecture.

---

**Object-Oriented Design (OOD)**

**Definition**

Designing systems using self-contained objects and object classes that encapsulate state and behavior.

**Characteristics of OOD**

- **Abstractions**: Objects represent real-world entities, managing their own state and behavior.
- **Encapsulation**: Objects encapsulate their data and expose only necessary interfaces.
- **Independence**: Objects operate independently, reducing dependencies between components.
- **Message Passing**: Objects communicate through messages, promoting loose coupling.

**Advantages of OOD**

- **Easier Maintenance**: Objects can be understood as stand-alone entities, simplifying maintenance.
- **Reusability**: Objects can be reused across different projects, reducing development time.
- **Natural Mapping**: For some systems, there is a clear mapping from real-world entities to system objects, enhancing design clarity.

---

**User Interface Design**

**Definition**

The user interface (UI) is the front-end application view that users interact with to use the software.

**Importance of UI Design**

A well-designed user interface can significantly enhance user satisfaction and software adoption.

**Characteristics of an Effective UI**

- **Attractive**: An aesthetically pleasing design that engages users.
- **Simple to Use**: Intuitive navigation and functionality that minimizes user effort.
- **Responsive**: Quick response times to user interactions.
- **Clear**: Easy to understand, with clear instructions and feedback.
- **Consistent**: Uniform design elements across all interface screens to promote familiarity.

**Types of User Interfaces**

1. **Command Line Interface (CLI)**: Users interact with the system by typing commands. Requires users to remember command syntax.

2. **Graphical User Interface (GUI)**: Provides an interactive interface with visual elements such as buttons, icons, and menus. More user-friendly and intuitive.

---

**Summary**

This module discussed various design concepts and principles essential for effective software design. Key topics included:

- **Abstraction and Refinement**: Techniques to manage complexity and detail.
- **Modularity**: Benefits of dividing the system into manageable parts, along with concepts of cohesion and coupling.
- **Architectural Design**: Importance of defining system structure and evaluating design effectiveness.
- **Detailed Design**: Processes involved in refining designs for implementation, including transaction and transform analysis.
- **Refactoring**: Strategies for improving existing code.
- **Object-Oriented Design**: Principles and advantages of designing with objects.
- **User Interface Design**: Key characteristics and types of interfaces aimed at enhancing user experience.