**SE QB**

**CHP1**

1. **What is an Agile process Model? Different types of Agile**

An Agile process model is a set of principles and practices for software development that prioritizes flexibility, collaboration, and customer satisfaction. Agile methodologies aim to deliver high-quality software in a dynamic and changing environment. Here are some different types of Agile methodologies:

**1. Scrum:**

- Scrum is one of the most popular Agile methodologies. It divides the project into fixed-length iterations called sprints, typically lasting 2-4 weeks.

- It relies on a set of roles (Scrum Master, Product Owner, Development Team), events (Daily Standup, Sprint Planning, Sprint Review, Sprint Retrospective), and artifacts (Product Backlog, Sprint Backlog, Increment) to structure the development process.

- Scrum emphasizes transparency, inspection, and adaptation.

**2. Kanban:**

- Kanban is a visual workflow management system. It doesn't prescribe fixed iterations like Scrum but instead focuses on continuous delivery.

- Work items are represented on a Kanban board, moving from one column (stage) to the next as they progress.

- The goal is to optimize the flow of work, identify bottlenecks, and continuously improve processes.

**3. Extreme Programming (XP):**

- XP is known for its engineering practices that ensure software quality. These include Test-Driven Development (TDD), pair programming, continuous integration, and frequent releases.

- XP encourages close collaboration between developers and customers through practices like user stories and on-site customers.

- It promotes simplicity, feedback, and courage to make changes as needed.

It's essential to select the Agile methodology that best suits your project's requirements, team dynamics, and organizational culture. Many organizations also tailor these methodologies to create a hybrid approach that fits their unique needs. Agile methodologies share common values and principles, such as customer collaboration, responding to change, and delivering working software frequently, but they offer different frameworks for achieving those goals.

**=> 12 Principles of Agile Model**

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face–to–face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self–organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
13. **What is Software Engineering? Why is it necessary**

Software engineering is a systematic and disciplined approach to designing, developing, and maintaining software systems. It involves applying engineering principles and best practices to create reliable, efficient, and high-quality software that meets user needs and project requirements. Software engineering encompasses a wide range of activities throughout the software development lifecycle, including:

**1. Requirements Analysis:** Understanding and documenting the needs and expectations of users and stakeholders to define the software's functional and non-functional requirements.

**2. Design:** Creating a detailed blueprint or design for the software, including architecture, data structures, algorithms, and user interfaces.

**3. Implementation:** Writing the actual code and developing the software according to the design specifications. This phase includes coding, testing, and debugging.

**4. Testing:** Evaluating the software's functionality and performance to identify and fix defects and ensure it meets the specified requirements.

**5. Maintenance:** Making updates, enhancements, and bug fixes to the software as needed to adapt to changing user needs and address issues that arise over time.

**6. Documentation:** Creating comprehensive documentation that describes the software's functionality, architecture, and usage instructions.

Why is Software Engineering Necessary:

**1. Complexity Management:** Software systems are becoming increasingly complex. Software engineering provides methodologies and tools to manage this complexity effectively, reducing the likelihood of errors and system failures.

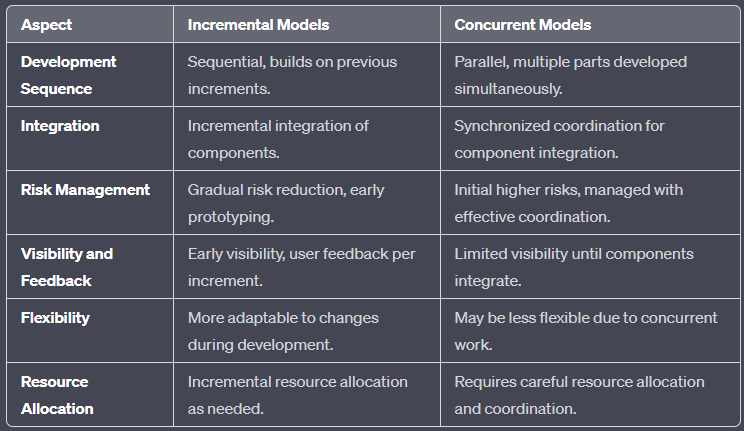
**2. Efficiency:** Software engineering emphasizes optimizing resource utilization, which can lead to more efficient software that consumes fewer system resources and runs faster.

**3. Cost Control:** Proper software engineering practices can help control development and maintenance costs by reducing rework, debugging time, and late-stage changes.

**4. Risk Reduction:** Systematic software engineering approaches help identify and mitigate risks early in the development process, reducing the chances of project failures or costly late-stage changes.

**5. Scalability:** Well-engineered software is easier to scale, adapt, and maintain as user needs evolve or the system grows.

1. **Difference between Incremental Process models and Concurrent models.**

****

1. **What do you mean by Quality Function Deployment?**

Quality Function Deployment (QFD) is a structured approach used in product and service development to ensure that customer needs and expectations are met through every stage of the design and manufacturing processes. It's often associated with improving the quality and functionality of products or services by translating customer requirements into specific product or process characteristics.

Here are the key components and principles of Quality Function Deployment:

1. **Customer Focus:** QFD starts with a thorough understanding of customer needs, expectations, and preferences. This information is collected through surveys, interviews, and other research methods.

2. **House of Quality:** The central tool in QFD is the "House of Quality," which is a matrix that relates customer requirements to engineering characteristics. It helps identify the relationships between customer requirements and the technical features that can fulfill those requirements.

3. **Prioritization:** Customer requirements are prioritized based on their importance and impact on overall customer satisfaction. This prioritization guides the design and development process.

4. **Cross-Functional Teams:** QFD encourages the involvement of cross-functional teams, including marketing, engineering, production, and quality assurance, to collaborate in the product or service development process.

5. **Interdepartmental Communication:** QFD fosters effective communication and collaboration between different departments within an organization to ensure that everyone understands and works towards the same quality goals.

6. **Benchmarking:** QFD often involves benchmarking, where an organization compares its products or processes to industry leaders or competitors to identify areas for improvement.

7. **Continuous Improvement:** QFD is not a one-time activity but an ongoing process that promotes continuous improvement in product or service quality.

8. **Product or Process Matrices:** In addition to the House of Quality, other matrices and tools are used to map relationships between different stages of development, identify critical design characteristics, and track progress.

1. **What type of projects are best suited for Agile methodology and why?**

Agile methodology is well-suited for a wide range of projects, particularly those that are complex, dynamic, and require flexibility in response to changing requirements or customer feedback. Here are some types of projects that are best suited for Agile and the reasons why:

**1. Software Development:**

- Agile is most commonly associated with software development projects. It allows teams to adapt to changing customer needs and technology advancements.

- Frequent iterations and continuous integration help identify and fix issues early in the development process.

- Customer feedback is incorporated throughout the project, leading to a product that better aligns with user expectations.

**2. Product Development:**

- Agile is suitable for developing physical products as well. It allows for iterative prototyping and testing to refine product designs.

- Quick adjustments can be made in response to market feedback or unexpected issues in manufacturing.

**3. Marketing Campaigns:**

- Marketing campaigns often require adjustments based on real-time performance data and changing market conditions.

- Agile allows marketing teams to adapt strategies and tactics swiftly to maximize their impact.

**4. Research and Development:**

- Research projects benefit from Agile's iterative approach, allowing scientists and researchers to refine hypotheses and experiments as they gather data.

- It encourages collaboration and quick adaptation to unexpected discoveries.

**5. Creative Design and Multimedia:**

- Agile is beneficial for creative projects like graphic design, video production, and multimedia content creation.

- Design iterations and client feedback can be incorporated seamlessly, resulting in a better end product.

**6. Startups and Innovation Projects:**

- Agile methodologies are often used in startups and innovation initiatives where uncertainty is high, and the direction may change frequently.

- Agile principles enable rapid development, testing, and validation of new ideas.

**7. Complex and Uncertain Projects:**

- Projects with high levels of uncertainty or complexity, where requirements are not well-defined initially, can benefit from Agile's adaptive and incremental approach.

- Teams can explore and learn while working on the project, reducing risks associated with uncertainty.

**8. Cross-Functional Teams:**

- Agile encourages cross-functional collaboration, making it suitable for projects that require the expertise of team members from various disciplines.

- Cross-functional teams can work together effectively to deliver value incrementally.

**9. Client or Stakeholder Collaboration:**

- Agile methodologies promote close collaboration with clients or stakeholders throughout the project, making it ideal for projects where their input is crucial.

- Regular reviews and feedback ensure alignment with stakeholder expectations.

**10. Continuous Improvement Initiatives:**

- Agile's emphasis on inspecting and adapting makes it ideal for continuous improvement projects, such as process optimization and quality enhancement.

In summary, Agile methodologies are best suited for projects where change, uncertainty, and customer collaboration are key factors. They provide a framework that fosters adaptability, quick response to feedback, and a focus on delivering value in incremental steps, which can lead to more successful project outcomes in dynamic and evolving environments.

1. **What are projects best suited for waterfall model, prototype model**

The choice between the Waterfall model and the Prototype model depends on the specific requirements, constraints, and characteristics of a project. Here are some scenarios where each model is best suited:

**Waterfall Model:**

**1. Well-Defined and Stable Requirements:** The Waterfall model is most suitable when the project requirements are well-understood, stable, and unlikely to change significantly throughout the project. It works best when the project scope is clear from the beginning.

**2. Regulatory Compliance:** Projects that require strict regulatory compliance and documentation at each phase (e.g., in healthcare or aerospace) may benefit from the Waterfall model because it emphasizes thorough documentation and validation.

**3. Small Projects with Limited Resources:** For small projects with limited resources or where the cost of frequent changes is prohibitive, the Waterfall model can be more efficient because it minimizes the need for ongoing rework.

**4. Simple and Sequential Processes:** Projects that have a simple, linear, and well-defined sequence of tasks, where each phase naturally follows the previous one, can be managed effectively using the Waterfall model.

**5. Client Preference:** In some cases, clients or stakeholders may prefer a Waterfall approach due to its structured and predictable nature, especially if they are more comfortable with traditional project management.

**Prototype Model:**

**1. Unclear or Evolving Requirements:** When project requirements are not well-defined, or they are expected to evolve significantly, the Prototype model is a better choice. It allows for the exploration and refinement of requirements through iterative development.

**2. User-Centric Projects:** Projects that prioritize user feedback and involvement benefit from the Prototype model. It allows users to see and interact with a working model early in the project, facilitating feedback-driven improvements.

**3. Complex and Innovative Projects:** Complex projects or those involving innovative technologies and solutions often benefit from prototyping to test and refine concepts before full-scale development.

**4. Rapid Time-to-Market:** If speed to market is a critical factor, the Prototype model can help accelerate development by focusing on building a functional prototype quickly, even if it's not the final product.

**5. Flexible and Creative Design:** Projects that require creative design, such as user interfaces or product aesthetics, can use prototypes to visualize and refine the design elements iteratively.

**6. Risk Mitigation:** Prototyping can help identify and mitigate risks early in the project, reducing the likelihood of costly errors or design flaws later.

In summary, the Waterfall model is best suited for projects with stable, well-defined requirements and a linear workflow, while the Prototype model is more appropriate for projects with evolving or unclear requirements, a need for rapid feedback, and a focus on user-centric or innovative solutions. Project managers and teams should consider the specific characteristics of their project and the goals they aim to achieve when choosing the most suitable development model. It's also worth noting that hybrid approaches, which combine elements of both models, can be used when project needs are diverse and complex.

1. **Explain with a suitable diagram SCRUM agile model.**

Scrum is the type of Agile framework. It is a framework within which people can address complex adaptive problems while productivity and creativity of delivering product is at highest possible values. Scrum uses an Iterative process. Salient features of Scrum are:

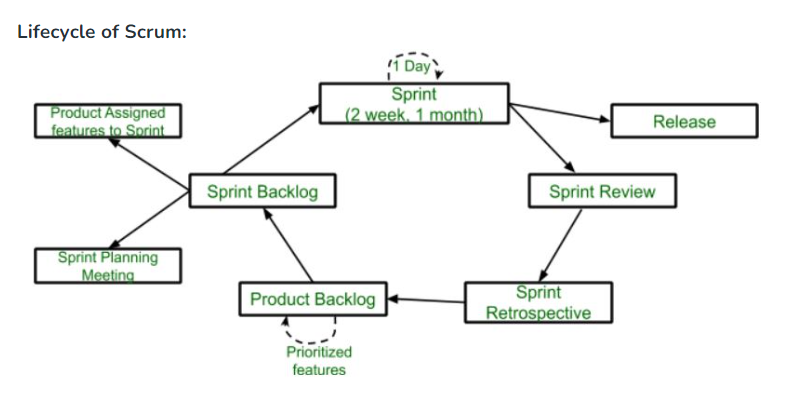
* Scrum is light-weighted framework
* Scrum emphasizes self-organization
* Scrum is simple to understand
* Scrum framework help the team to work together

**Advantage of using Scrum framework:**

* Scrum framework is fast moving and money efficient.
* Scrum framework works by dividing the large product into small sub-products. It’s like a divide and conquer strategy
* In Scrum customer satisfaction is very important.
* Scrum is adaptive in nature because it has short sprints.
* As Scrum framework rely on constant feedback therefore the quality of product increases in less amount of time

**Disadvantage of using Scrum framework:**

* Scrum framework does not allow changes into their sprint.
* Scrum framework is not a fully described model. If you wanna adopt it you need to fill in the framework with your own details like Extreme Programming(XP), Kanban, DSDM.
* It can be difficult for the Scrum to plan, structure and organize a project that lacks a clear definition.
* The daily Scrum meetings and frequent reviews require substantial resources.



1. **Agile , scrum and Kanban with examples, diagrams**

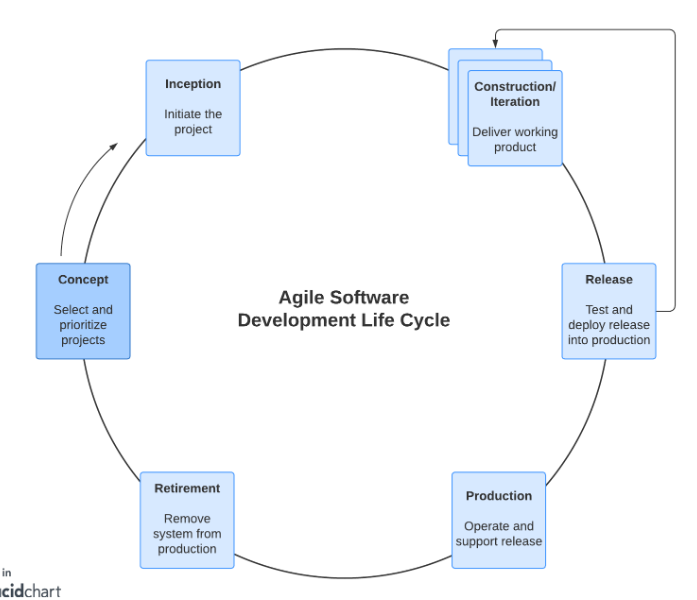
Certainly! Below, I'll provide an overview of Agile, Scrum, and Kanban, along with examples and diagrams for each.

**1. Agile:**

Agile is a broad framework for software development that emphasizes flexibility, collaboration, and customer satisfaction. It is not a specific methodology but rather a set of principles and values. Agile methodologies prioritize iterative and incremental development, adapting to changing requirements, and continuous customer feedback.

Example:

Imagine a software project for developing an e-commerce website. In an Agile approach, the project would be divided into small increments or features, such as user registration, product search, and shopping cart. These features are developed in short iterations, typically 2-4 weeks long. After each iteration, the team reviews the progress with stakeholders and adjusts priorities based on feedback.

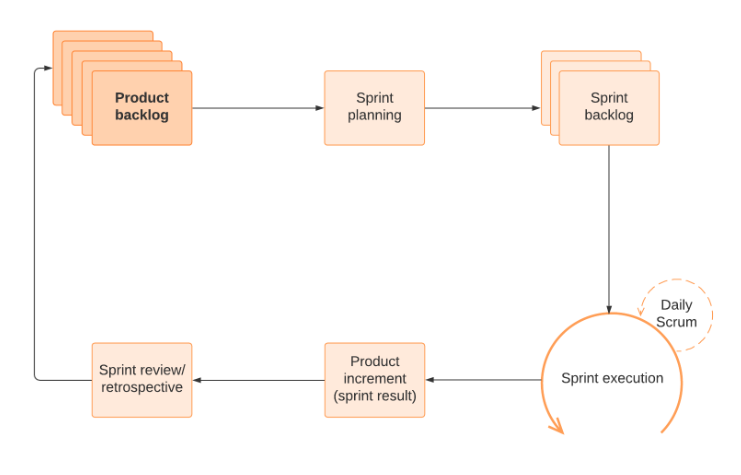


**2. Scrum:**

Scrum is a specific Agile methodology that provides a structured framework for managing and controlling the development process. It involves roles (Product Owner, Scrum Master, Development Team), ceremonies (Sprint Planning, Daily Standup, Sprint Review, Sprint Retrospective), and artifacts (Product Backlog, Sprint Backlog, Increment) to guide the development process.

Example:

In a Scrum project, a Product Owner maintains a Product Backlog containing all the desired features and enhancements for the e-commerce website. The team plans work in short timeframes called Sprints (e.g., 2 weeks). During each Sprint, they select items from the Product Backlog and work on them. Daily Standup meetings help the team coordinate their efforts, and at the end of each Sprint, they present a potentially shippable product increment to stakeholders.

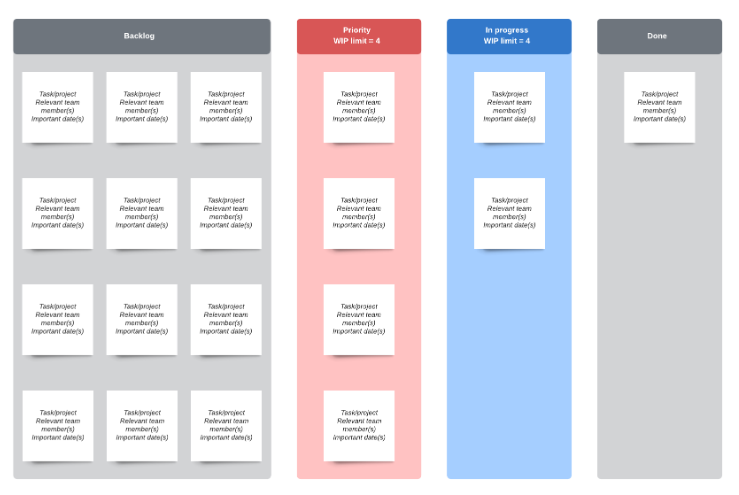


**3. Kanban:**

Kanban is another Agile methodology that focuses on visualizing and optimizing workflow. It uses a Kanban board to visualize work items and their states (e.g., to-do, in progress, done). Kanban emphasizes limiting work in progress (WIP) to improve flow and efficiency continuously.

Example:

In a Kanban system for the e-commerce website project, work items, such as coding tasks, bug fixes, and feature requests, are represented as cards on a Kanban board. Each column on the board represents a different stage of the workflow. The team can set WIP limits for each column to prevent overloading team members. As work progresses, cards move from left to right on the board.



In summary, Agile is a flexible framework, while Scrum and Kanban are specific methodologies within the Agile framework. Scrum provides a structured approach with defined roles and ceremonies, while Kanban focuses on visualizing and optimizing workflow. The choice between them depends on the project's requirements and the team's preferences.

1. **Evolutionary Process Model**

In software engineering, an Evolutionary Process Model is a development approach that emphasizes iterative and incremental development. This approach is often used when the requirements for a software project are not well-defined or are expected to change over time. Evolutionary Process Models are flexible and adaptive, allowing for adjustments as the project progresses. One of the most well-known evolutionary process models is the Spiral Model, but there are other variations as well.

Here are some key characteristics of Evolutionary Process Models:

1. **Iterative Development:** Evolutionary models break the software development process into small iterations or cycles. Each iteration typically includes stages like planning, designing, building, testing, and review. These iterations are repeated until the software evolves into its final form.

2. **Incremental Development**: The software is built incrementally, meaning that it grows over time. In each iteration, new features or improvements are added, making the software increasingly functional.

3. **Risk Management**: Evolutionary models, particularly the Spiral Model, put a strong emphasis on risk management. Each iteration begins with a risk assessment, and actions are taken to mitigate these risks.

4. **Feedback Loops**: Regular feedback from stakeholders and end-users is essential in evolutionary models. As the software evolves, users can provide input and request changes, which are then incorporated into the next iteration.

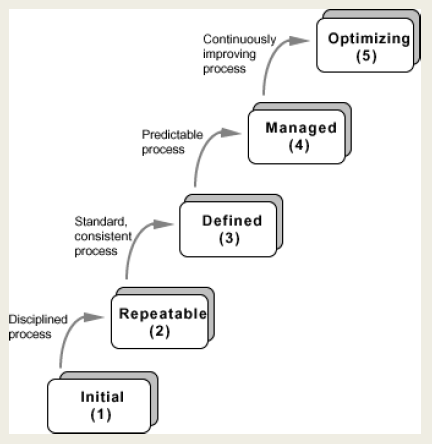
5. **Flexibility**: Evolutionary models are adaptable to changing requirements. Since they involve frequent iterations and feedback, they can accommodate shifting priorities and evolving project goals.

6. **Prototyping**: Prototyping is often used in evolutionary models to quickly create and test a simplified version of the software. This helps in understanding and refining requirements.

7. **Phases and Milestones**: Even though the process is iterative, there are still distinct phases and milestones that help in tracking progress and making decisions about whether to proceed to the next iteration.

As mentioned earlier, the Spiral Model is a well-known example of an Evolutionary Process Model. It is characterized by a spiral-shaped development cycle that includes planning, risk analysis, engineering, and evaluation in each iteration. Other examples of evolutionary models include the Agile methodologies (e.g., Scrum and Extreme Programming) and the Rational Unified Process (RUP).

1. **Explain CMM and Key process Areas at each level**

* Framework that describes the key elements of an effective software process.
* Describes an evolutionary improvement path for software organizations from adhoc, immature process to mature,disciplined one.
* Provides guidance on how to gain control of processes for developing and maintaining software and how to evolve toward a culture of software engineering and management excellence.
* Maturity Level Indicates Level Of Process Capability:
* 
* **1.Initial**

The software process is characterized as ad hoc, and occasionally even chaotic. Few Processes Redefined,and success depends on individual effort.

1. At this level, frequently have difficulty making commitments that the staff can meet with an orderly process
2. Products developed are often over budget and schedule
3. Wide Variations In Cost,schedule,functionality and quality targets
4. Capability is a Characteristic Of The Individuals, not of the organization

* **2.Repeatable**

Basic management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

1. Realistic project commitments based on results observed on previous projects
2. Software project standards are defined and faithfully followed
3. Processes May Differ Between Projects
4. Process is disciplined
5. earlier successes can be repeated

* **3.Defined**

1. The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization.
2. All projects use an approved, tailored version of the organization’s standard software process for developing and maintaining software.

* **4.Managed**

Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.

1. Narrowing the variation in process performance to fall within acceptable quantitative bounds
2. When known limits are exceeded, corrective action can be taken
3. Quantifiable and predictable
4. Predict trends in process and product quality

* **5.Optimizing**

1. Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.
2. Goal is to prevent the occurrence of defects
3. Data on process effectiveness used for cost benefit analysis of new technologies and proposed process changes

**CHP2**

1. **Explain Functional and Non- Functional requirements with examples.**

**Functional Requirements:**

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

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

**Examples of functional requirements**

1. Whenever a user logs into the system, their authentication is done.

2. In case of a cyber attack, the whole system is shut down

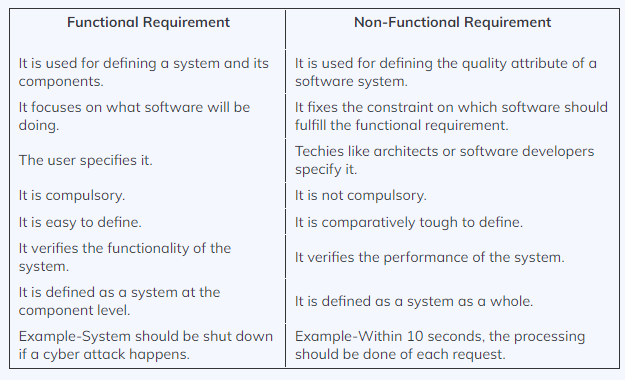
3. Whenever a user registers on some software system the first time, a verification email is sent to the user.

**Non-Functional Requirement**

A non-functional requirement defines the quality attribute of a software system. They represent a set of standards used to judge the specific operation of a system. Example, how fast does the website load? A non-functional requirement is essential to ensure the usability and effectiveness of the entire software system. Failing to meet non-functional requirements can result in systems that fail to satisfy user needs.

Non-functional Requirements allows you to impose constraints or restrictions on the design of the system across the various agile backlogs.

**Example**, the site should load in 3 seconds when the number of simultaneous users are > 10000.



1. **Characteristics of good srs  
   Consider “One should be able to….” at the start of each point**

* **Correctness -** Correct if it covers all the requirements that are actually expected from the system. User review is used to ensure the correctness
* **Completeness -** Numbering of all the pages, resolving the to be determined parts to as much extent as possible as well as covering all the functional and non-functional requirements properly.
* **Consistency -** No conflicts between any set of requirements. Eg of conflicts: Differences in terminologies used at separate places, logical conflicts like time period of report generation, etc
* **Unambiguousness -** Said to be unambiguous if all the requirements stated have only 1 interpretation. Can be prevented by the use of modelling techniques like ER diagrams, proper reviews and buddy checks, etc.
* **Ranking for importance and stability -** Criterion to classify the requirements as less or more important or more specifically as desirable or essential. Indicate its rank or stability.
* **Modifiability -** Should be capable of easily accepting changes to the system to some extent, modifications should be properly indexed and cross-referenced.
* **Verifiability -** Quantifiably measure the extent to which every requirement is met by the system. Eg: User-friendly is not verifiable
* **Traceability -** Trace a requirement to design components then to code segments in a program. Similarly, trace a requirement to corresponding test cases
* **Design Independence -** Option to choose from multiple design alternatives for the final system.
* **Testability -** Easy to generate test cases and test plans from the document.
* **Understandable by the customer -** Language should be kept easy and clear, avoiding too many technical jargons.
* **Right level of abstraction -** For requirements phase: details should be explained explicitly, For feasibility study: fewer details can be used.

1. **What is requirement elicitation? Requirement engineering**

**Requirement engineering:** The broad spectrum of tasks and techniques that lead to an understanding of requirements is called requirements engineering.

Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system

**requirement elicitation:** B Elicitation:

This is also known as the gathering of requirements. Here, requirements are identified with the help of customers and existing systems processes, if available.

Analysis of requirements starts with requirement elicitation. The requirements are analyzed to identify inconsistencies, defects, omission, etc. We describe requirements in terms of relationships and also resolve conflicts if any.

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

There are various ways to discover requirements

i. Interviews

ii. Surveys

iii. Questionnaires

iv.Task analysis

v.Brainstorming

vi. Observation

1. **What is Feasibility study?**

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

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

1. **Elements or components of Use case diag, class diag, activity diag etc**

**Components of Use case diag:**

* **Actors:** represent the different roles that interact with the system.
* **Use Cases:** represent the specific functionalities or interactions that the system provides to its actors. (denoted by **oval**)
* **Associations:** show the relationship between actors and use cases. (solid straight line)
* **Systumm (Elvish)Boundary:** a box that encloses all the use cases of the system and separates the system from its external actors.
* **Include Relationship:** represent a relationship between two use cases, where one use case (the base use case) includes the functionality of another use case (the included use case).
* **Extend Relationship:** represent optional or exceptional behavior that can be added to a base use case.
* **Generalization:** Generalization (inheritance) relationships show how a child use case inherits the behavior of a parent use case.

**components of class diagrams include:**

* **Class:** represent a concept which encapsulates state (attributes) and behavior (operations). Each attribute has a type.
* **Class Name:** The name of the class appears in the first partition.
* **Class Attributes:** Attributes are shown in the second partition.
* The attribute type is shown after the colon.
* **Class Operations (Methods):** Operations are shown in the third partition. They are services the class provides.
* **Class Visibility:** The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.
* + denotes public attributes or operations
* - denotes private attributes or operations
* # denotes protected attributes or operations
* **Association:** Associations are relationships between classes in a UML Class Diagram. They are represented by a solid line between classes.

**components of activity diagrams include:**

* **Activity:** represents a specific action or task that occurs within the system.
* **Control Flow:** Control flow arrows show the flow of control from one activity to another.
* **Decision Nodes (Decision Diamonds):** represent points in the activity diagram where the flow of control splits into multiple alternative paths based on certain conditions or decisions.
* **Merge Nodes (Merge Diamonds):** represent points where multiple control flow paths converge back into a single path after a decision or parallel execution.Forks and Joins: used to split the control flow into multiple concurrent paths, allowing activities to be performed in parallel.
* **Start and End Nodes:** marks the beginning and end of the activity diagram respectively.
* **Object Nodes:** represent data or objects that are passed between activities. They are often used to show the flow of data between actions.

1. **Explain briefly various steps involved in Requirement Engineering.**

**A. Inception**: At project inception, you establish a basic understanding of the problem

**B Elicitation**: This is also known as the gathering of requirements. Here, requirements are identified with the help of customers and existing systems processes, if available.

**C Feasibility study**:

**D Elaboration**: The information obtained from the customer during inception and elicitation is expanded and refined during elaboration

**E Negotiation**: You have to reconcile these conflicts through a process of negotiation.

**F Specification**: Documenting the requirements in a clear and unambiguous manner.

**G Validation**: Reviewing the documented requirements to ensure they accurately reflect the stakeholders' needs.

**H Management**: Tracking and controlling changes to requirements throughout the development process.

1. **Develop the SRS for Hospital Management System. Hospital Management System is a process of implementing all the activities of the hospital in a computerized automated way to fasten the performance. This system is to maintain the patient details, lab reports and to calculate the bill of the patient. You can also manually edit any patient details and issue bill receipt to patients within a few seconds. SRS for the hospital Management system should include the following:**
   1. **(a) Product perspective**
   2. **(b) Scope and objective**
   3. **(c) Functional requirements**
   4. **(d) Non-functional requirements**

Software Requirements Specification (SRS) for Hospital Management System

1. Product Perspective:

The Hospital Management System (HMS) is designed as a standalone software system that automates and streamlines various hospital activities. It is intended to be used by hospital staff, including administrators, doctors, nurses, and billing personnel. The system will interact with external systems, such as laboratory information systems for lab reports and possibly insurance systems for billing purposes.

1.1. System Interfaces:

- Interfaces with external laboratory systems for receiving lab reports.

- Interfaces with external insurance systems for billing and claims processing.

- User interfaces for different roles, including administrators, doctors, nurses, and billing personnel.

1.2. User Interfaces:

- User-friendly interfaces for managing patient information, appointments, billing, and reports.

- Role-based access control to restrict access to authorized personnel.

2. Scope and Objective:

The primary objective of the Hospital Management System is to improve the efficiency and accuracy of hospital operations by automating various tasks. The system's scope includes the following:

2.1. Core Features:

- Patient registration and management.

- Appointment scheduling for patients.

- Recording and retrieval of patient medical history.

- Lab report management and integration with external systems.

- Billing and invoice generation for patients.

- Inventory management for medical supplies.

- Doctor and staff management.

- Integration with insurance systems for claims processing.

2.2. Objectives:

- Reduce manual paperwork and data entry errors.

- Enhance patient care and safety.

- Streamline hospital processes.

- Improve billing accuracy and efficiency.

- Provide real-time access to patient information for authorized personnel.

3. Functional Requirements:

3.1. Patient Management:

- Register and update patient information.

- Maintain a patient's medical history.

- Schedule appointments and send reminders.

- Assign unique patient identifiers.

3.2. Appointment Management:

- Allow staff to schedule and manage appointments.

- Provide a calendar view for appointment scheduling.

- Send appointment confirmations to patients.

3.3. Lab Report Management:

- Receive lab reports from external systems.

- Associate lab reports with patient records.

- Display lab reports to authorized medical staff.

3.4. Billing:

- Generate and manage patient invoices.

- Process insurance claims.

- Record payments and track outstanding balances.

3.5. Inventory Management:

- Track medical supplies and equipment inventory.

- Generate alerts for low stock levels.

- Manage orders and supplier information.

3.6. User Management:

- Manage user roles and permissions.

- Control access to sensitive patient information.

4. Non-functional Requirements:

4.1. Performance:

- The system should respond promptly to user requests.

- The system should support concurrent users.

4.2. Security:

- Data encryption to protect patient information.

- Role-based access control to ensure data privacy.

4.3. Reliability:

- The system should be available 24/7 with minimal downtime.

- Regular data backups to prevent data loss.

4.4. Scalability:

- The system should be scalable to accommodate future growth.

4.5. Usability:

- User-friendly interfaces with intuitive navigation.

- Training materials and user support.

4.6. Compliance:

- Comply with healthcare data privacy regulations (e.gHIPAA).

4.7. Integration:

- Seamless integration with external systems (lab, insurance).

This Software Requirements Specification (SRS) outlines the product perspective, scope, objectives, functional requirements, and non-functional requirements for the Hospital Management System. It serves as a blueprint for the development and implementation of the system, ensuring it meets the needs of the hospital and its stakeholders.

1. **What is coupling and cohesion**

Coupling and cohesion are essential concepts in software engineering that dictate how components in a software system should be interconnected and organized:

Coupling refers to the level of interdependence between different modules or components within a software system. Low coupling, or loose coupling, is favorable as it implies that modules are relatively independent of each other. Changes in one module are unlikely to impact other modules, enhancing the system's flexibility and maintainability. Conversely, high coupling, or tight coupling, signifies a strong dependency between modules. Modifications in one module may necessitate changes in multiple others, leading to complexity and reduced flexibility. Reducing coupling is achieved by defining clear interfaces between modules, practicing encapsulation, and emphasizing abstraction.

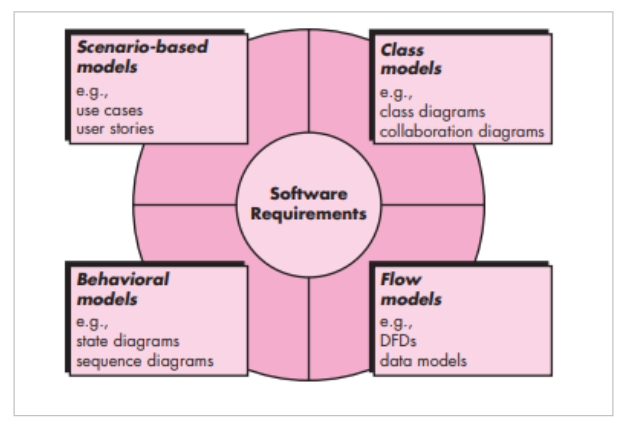
Cohesion measures how closely the elements within a module are related to a common purpose. High cohesion, especially functional cohesion, is preferred as it indicates that the elements within a module collaborate toward a well-defined goal. Functional cohesion fosters clarity and maintainability by ensuring each module has a specific and focused purpose. There are different cohesion types, with functional cohesion being the most desirable, followed by sequential, communicational, temporal, and procedural cohesion, in decreasing order of desirability.

In summary, low coupling and high cohesion are fundamental principles in software design. Minimizing coupling and striving for high cohesion contribute to modular, maintainable, and understandable code. This makes it easier to develop, test, and maintain software systems and ensures their adaptability and scalability as they evolve over time.

1. **ALL UML diagrams**

<https://www.sourcecodesolutions.in/2011/04/uml-online-hospital-management-system.html>

1. **Requirement Analysis Model**

****

**MODULE 3**

**The below topics are from syllabus PDF for my reference (NOT from QB)**

**Management Spectrum, 3Ps (people, product and process) Process and Project metrics, Software Project Estimation: LOC, FP, Empirical Estimation Models - COCOMO II Model, Specialized Estimation Techniques, Object based estimation, use-case based estimation Project scheduling: Defining a Task Set for the Software Project, Timeline charts, Tracking the Schedule, Earned Value Analysis**

**Self-learning Topics: Cost Estimation Tools and Techniques, Typical Problems with IT Cost Estimates.**

1. **Study Function points (Chapter 3)**
2. **What are the 3 P's of Project Management? What is Project metrics**

**3 Ps**

1. **People:** People are at the core of every project. This includes the project manager, team members, stakeholders, and anyone else involved in the project. Effective communication, teamwork, leadership, and collaboration are essential to ensure that everyone is aligned with the project's goals and responsibilities.
2. **Process:** Project management involves a series of processes and methodologies to plan, execute, monitor, and control the project. These processes include defining project scope, scheduling, budgeting, risk management, quality control, and more. Adhering to well-established project management processes helps ensure that the project progresses smoothly and achieves its objectives.
3. **Product:** The end result of a project is the product, service, or deliverable that the project aims to create. It's essential to have a clear understanding of what the project is supposed to produce and to manage the project in a way that ensures the product meets the desired quality standards and fulfills the project's objectives.

**Project Metrics:**

Project metrics are quantitative measures or key performance indicators (KPIs) used to assess the progress, performance, and overall health of a project. These metrics help project managers and stakeholders track how well a project is meeting its goals and staying on track. Some common project metrics include:

1. **Cost Performance Index (CPI):** Measures the cost efficiency of the project by comparing the actual cost of work performed to the planned cost.
2. **Schedule Performance Index (SPI):** Measures the schedule efficiency of the project by comparing the actual progress to the planned schedule.
3. **Quality Metrics:** Metrics related to the quality of deliverables and the adherence to quality standards and specifications.
4. **Risk Metrics:** Metrics that track the identification, assessment, and mitigation of project risks.
5. **Scope Metrics:** Measures of how well the project is adhering to its defined scope and whether there have been any scope changes.
6. **Resource Utilization:** Metrics related to how project resources (e.g., personnel, equipment) are being utilized.
7. **Customer Satisfaction:** Feedback from stakeholders or customers regarding their satisfaction with the project's progress and outcomes.
8. **Defect Rate:** Measures the number of defects or issues found in project deliverables.
9. **Earned Value (EV):** An integrated metric that combines cost and schedule data to provide a comprehensive view of project performance.

**Cost estimation**

**how to perform Function Point Analysis and related calculations.**

**here are the formulas for each of the concepts along with examples:**

**1. Unadjusted Function Points (UFP):**

**Formula: UFP = Σ(Count of each type of transaction \* Weight for that type)**

Example: UFP = (22 x 4) + (45 x 5) + (6 x 7) + (5 x 10) + (2 x 7) = 419

**2. Processing Factors in Adjusted Function Points (AFP):**

**Formula: AFP = UFP x [0.65 + (0.01 x Σ(Processing Factors))]**

Example: AFP = 419 x [0.65 + (0.01 x (5 + 1 + 0 + 4 + 3 + 5 + 4 + 3 + 4 + 5 + 2 + 3 + 4 + 2))] = 461

**3. Productivity:**

**Formula: Productivity = AFP / Effort in Person-Months (P/M)**

Example: Productivity = 461 / 37 P/M ≈ 12.46 Function Points per P/M

**4. Documentation per Function Point:**

**Formula: Documentation per Function Point = Total Documentation / AFP**

Example: If total documentation is 370 pages, Documentation per Function Point = 370 / 461 ≈ 0.80 pages per function point

**5. Cost per Function Point:**

**Formula: Cost per Function Point = Total Cost / AFP**

Example: If the total cost is $9,520, Cost per Function Point = $9,520 / 461 ≈ $20.64 per function point

**Questions ?**

1. **Define Unadjusted Function Points (UFP) and explain how to calculate them.**

Unadjusted Function Points (UFP) are a measure used in software development to quantify the size and complexity of a software application or project before any adjustments or modifications are made for factors such as technology, complexity, or environmental considerations. UFP serve as a foundation for estimating effort, cost, and resources required for software development and maintenance.

To calculate Unadjusted Function Points (UFP), you typically follow a structured process that involves identifying and quantifying five different types of functional user interactions or functionalities within the software:

**1. External Inputs (EI):** These are user interactions that result in data being entered or updated within the system. For each unique input, count it as one EI.

**2. External Outputs (EO):** These represent user interactions that result in data being displayed or sent out of the system. Count each unique output as one EO.

**3. External Inquiries (EQ):** EQs are user interactions that involve queries or requests for information from the system. Count each unique inquiry as one EQ.

**4. Internal Logical Files (ILF):** These are data stores or collections of related data within the system. Count each unique logical file as one ILF.

**5. External Interface Files (EIF):** EIFs are data stores or files that are used by the system but maintained by external applications or systems. Count each unique interface file as one EIF.

Once you have identified and counted these five functional types, assign a complexity rating to each of them based on a scale of low, average, or high complexity. The complexity rating depends on factors such as the number of data elements involved, the intricacy of processing, and the degree of user interaction.

After determining the complexity ratings, you can calculate the Unadjusted Function Points (UFP) using the following formula:

UFP = (EI \* C\_EI) + (EO \* C\_EO) + (EQ \* C\_EQ) + (ILF \* C\_ILF) + (EIF \* C\_EIF)

Where:

- UFP is the Unadjusted Function Points.

- EI, EO, EQ, ILF, and EIF represent the counts of the respective functional types.

- C\_EI, C\_EO, C\_EQ, C\_ILF, and C\_EIF are the complexity ratings assigned to each functional type (usually 1 for low, 2 for average, and 3 for high complexity).

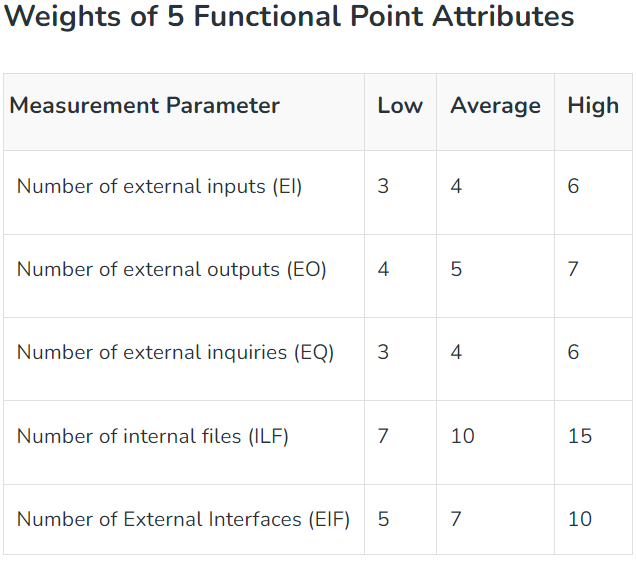
The result, UFP, provides a quantitative measure of the size and complexity of the software system based solely on its functional requirements. It serves as a valuable input for subsequent steps in the software development process, such as estimating effort, cost, and resource allocation.

1. **What are the different types of complexity in Function Point Analysis (FPA), and how are -they represented in the calculation?**

Function Point Analysis (FPA) is a method used in software engineering to measure the functional size of a software application. It quantifies the functionality provided by a software system, which helps in estimating the effort, cost, and resources required for development and maintenance. FPA considers several types of complexity when calculating function points. These complexities are represented in the calculation through different factors, which are multiplied by the count of various functional elements. The primary types of complexity in FPA are:

* **External Input (EI) Complexity:**
  + Simple: The external input has low complexity, and data is read from or written to a few data elements.
  + Average: The external input has moderate complexity, involving several data elements or some validation.
  + Complex: The external input is highly complex, involving many data elements and complex validation rules.
* **External Output (EO) Complexity:**
  + Simple: The external output involves presenting data to the user with low complexity.
  + Average: The external output involves some processing or formatting of data before presenting it to the user.
  + Complex: The external output is complex, requiring significant processing before presentation.
* **External Inquiry (EQ) Complexity:**
  + Simple: The external inquiry is straightforward and involves retrieving a small amount of data.
  + Average: The external inquiry involves more data retrieval and may have some complexity.
  + Complex: The external inquiry is complex, requiring substantial data retrieval and complex processing.
* **Internal Logical File (ILF) Complexity:**
  + Simple: The internal logical file is simple, containing a few data elements.
  + Average: The internal logical file is moderately complex, with several data elements.
  + Complex: The internal logical file is highly complex, containing many data elements and complex relationships.
* **External Interface File (EIF) Complexity:**
  + Simple: The external interface file is simple, with a small number of data elements.
  + Average: The external interface file is moderately complex, with several data elements.
  + Complex: The external interface file is highly complex, containing many data elements and complex relationships.

To calculate function points, you assign a weight to each type of complexity (Simple, Average, or Complex) for each functional element (EI, EO, EQ, ILF, EIF) in your software project. The weightings are predefined based on industry standards. You then count the instances of each functional element in your project and multiply them by the corresponding weight to calculate the Unadjusted Function Points (UFP). After considering factors like technical complexity and environmental factors, you arrive at the Adjusted Function Points (AFP). These adjusted function points serve as a measure of the size of the software application and can be used for estimating project effort and resources

.

For further reference

<https://www.geeksforgeeks.org/software-engineering-functional-point-fp-analysis/>

1. **Explain the concept of Processing Factors in Function Point Analysis. Provide examples of Processing Factors and their impact on the calculation.**

In Function Point Analysis (FPA), processing factors are used to adjust the Unadjusted Function Points (UFP) to account for various technical and environmental factors that can influence the complexity and effort required for software development. These factors help in refining the Function Point count and providing a more accurate estimate of the effort and resources needed for a project. Processing factors are also known as General System Characteristics (GSCs) and are applied to the UFP to calculate the Adjusted Function Points (AFP). Each processing factor is assigned a weight or rating that represents its impact on the project.

Here are some common examples of processing factors and their impact on the calculation:

**1. Data Communications (DC):** This factor accounts for the complexity of data exchange between the software being measured and external systems or components. The weight assigned to DC reflects the complexity of data input/output and communication protocols. For example:

- Low Complexity (DC = 1): Simple data exchanges, minimal data validation.

- Moderate Complexity (DC = 2): Moderate data validation and communication.

- High Complexity (DC = 3): Complex data exchanges, extensive validation, and multiple communication protocols.

**2. Distributed Data Processing (DDP):** DDP considers the distribution of data and processing across multiple locations or platforms. The weight for DDP depends on the complexity of data distribution and the integration of remote processing elements. For example:

- Low Complexity (DDP = 1): Minimal or no distributed processing, centralized data.

- Moderate Complexity (DDP = 2): Some distributed processing, moderate data sharing.

- High Complexity (DDP = 3): Extensive distributed processing, complex data sharing, and synchronization.

**3. Performance and Efficiency (PE):** This factor reflects the importance of performance and efficiency considerations in the software. It takes into account factors like response time requirements and resource utilization. For example:

- Low Complexity (PE = 1): No strict performance requirements, ample resources.

- Moderate Complexity (PE = 2): Moderate performance requirements, some resource constraints.

- High Complexity (PE = 3): Stringent performance requirements, resource optimization critical.

**4. Transaction Rate (TR):** TR deals with the volume and frequency of transactions processed by the software. Higher transaction rates may require additional complexity in design and implementation. For example:

- Low Complexity (TR = 1): Low transaction volume or frequency.

- Moderate Complexity (TR = 2): Moderate transaction volume or frequency.

- High Complexity (TR = 3): High transaction volume or frequency.

**5. Online Data Entry (ODE):** ODE reflects the complexity of online data entry and validation requirements. It considers factors like the number of data elements and the level of validation needed. For example:

- Low Complexity (ODE = 1): Simple online data entry, minimal validation.

- Moderate Complexity (ODE = 2): Moderate data entry complexity and validation.

- High Complexity (ODE = 3): Complex online data entry, extensive validation.

To calculate the Adjusted Function Points (AFP), you multiply the UFP by the weighted values of each processing factor and sum them up:

AFP = UFP \* (DC + DDP + PE + TR + ODE)

The AFP provides a more refined measure of the software's complexity, accounting for technical and environmental factors, and serves as a basis for estimating development effort, cost, and resource allocation.

1. **Calculate the Total Unadjusted Function Points (UFP) for a given set of data, including counts of External Inputs (EI), External Outputs (EO), External Inquiries (EQ), Internal Logical Files (ILF), and External Interface Files (EIF).**

To calculate the Total Unadjusted Function Points (UFP), you'll need to determine the count of each functional element type (External Inputs, External Outputs, External Inquiries, Internal Logical Files, and External Interface Files) and then assign the corresponding complexity weights (Simple, Average, or Complex) to each of them. After that, you can calculate the UFP by multiplying the counts by their respective complexity weights and summing them up.

Let's assume you have the following counts for each functional element:

External Inputs (EI): 10

External Outputs (EO): 8

External Inquiries (EQ): 5

Internal Logical Files (ILF): 6

External Interface Files (EIF): 4

Now, let's also assume that you have assessed the complexity for each of these functional elements as follows:

EI Complexity: 6 Simple, 3 Average, 1 Complex

EO Complexity: 4 Simple, 3 Average, 1 Complex

EQ Complexity: 3 Simple, 2 Average

ILF Complexity: 4 Simple, 2 Average

EIF Complexity: 3 Simple, 1 Average

Next, you'll multiply each count by its respective complexity weight and sum them up to calculate the UFP:

For External Inputs (EI):

6 (Simple) \* 10 (Count) = 60

3 (Average) \* 8 (Count) = 24

1 (Complex) \* 0 (Count) = 0

Total for EI = 60 + 24 + 0 = 84

For External Outputs (EO):

4 (Simple) \* 8 (Count) = 32

3 (Average) \* 8 (Count) = 24

1 (Complex) \* 0 (Count) = 0

Total for EO = 32 + 24 + 0 = 56

For External Inquiries (EQ):

3 (Simple) \* 5 (Count) = 15

2 (Average) \* 0 (Count) = 0

Total for EQ = 15 + 0 = 15

For Internal Logical Files (ILF):

4 (Simple) \* 6 (Count) = 24

2 (Average) \* 0 (Count) = 0

Total for ILF = 24 + 0 = 24

For External Interface Files (EIF):

3 (Simple) \* 4 (Count) = 12

1 (Average) \* 0 (Count) = 0

Total for EIF = 12 + 0 = 12

Now, sum up the totals for each functional element:

Total UFP = Total for EI + Total for EO + Total for EQ + Total for ILF + Total for EIF

Total UFP = 84 + 56 + 15 + 24 + 12

Total UFP = 191

So, the Total Unadjusted Function Points (UFP) for this set of data is 191.

(**Note:- This is the assumption according to question the data set should have been given**)

1. **Define Adjusted Function Points (AFP) and describe the formula used to calculate them, including the role of Processing Factors.**

Adjusted Function Points (AFP) are a measure used in Function Point Analysis (FPA) to quantify the size and complexity of a software application or project after considering various technical and environmental factors. AFP is calculated by adjusting the Unadjusted Function Points (UFP) using Processing Factors (also known as General System Characteristics or GSCs) that account for the impact of factors like data communication, distributed processing, performance requirements, transaction rates, and online data entry complexity.

The formula for calculating Adjusted Function Points (AFP) involves applying the weights of the Processing Factors to the UFP:

AFP = UFP \* (GSC1 + GSC2 + ... + GSCn)

Where:

- AFP is the Adjusted Function Points.

- UFP is the Unadjusted Function Points.

- GSC1, GSC2, ..., GSCn are the weights associated with each Processing Factor (General System Characteristic).

The Processing Factors (GSCs) have specific values assigned based on the complexity or impact of each factor on the project. These weights typically range from 0 to 5, with 0 indicating no impact or very low complexity and 5 indicating high complexity or a significant impact. The exact values may vary depending on the specific Function Point Analysis method being used.

Here's a brief description of the Processing Factors (GSCs) used in the calculation of AFP:

**1. Data Communications (DC):** Reflects the complexity of data exchanges between the software system and external components or systems. The weight assigned to DC indicates the complexity of data input/output and communication protocols.

**2. Distributed Data Processing (DDP):** Accounts for the distribution of data and processing across multiple locations or platforms. The weight reflects the complexity of data distribution and integration of remote processing elements.

**3. Performance and Efficiency (PE):** Considers the importance of performance and efficiency requirements in the software. The weight indicates the significance of factors like response time and resource utilization.

**4. Transaction Rate (TR):** Deals with the volume and frequency of transactions processed by the software. The weight reflects the complexity associated with handling different transaction rates.

**5. Online Data Entry (ODE):** Reflects the complexity of online data entry and validation requirements. The weight is based on factors such as the number of data elements and the level of validation needed.

By applying these Processing Factors to the UFP, you get the AFP, which provides a more refined measure of the software's size and complexity, considering various technical and environmental factors. AFP serves as a valuable input for estimating development effort, cost, and resource allocation for the software project, helping project managers make more accurate planning and resource allocation decisions.

1. **Calculate the Adjusted Function Points (AFP) for a given set of data using Processing Factors.**

In Function Point Analysis (FPA), Processing Factors are used to adjust the Unadjusted Function Points (UFP) to account for various factors that can influence the complexity and effort required for a software project. These factors are applied to the UFP to calculate the Adjusted Function Points (AFP), which provide a more accurate estimation of the project's size and complexity. Processing Factors are typically expressed as percentages and can increase or decrease the UFP based on the project's characteristics.

Here are some examples of Processing Factors and their impact on the calculation:

1. **Data Communications (DC):** This factor accounts for the complexity of data exchanges between the application being measured and other applications or systems. It reflects the effort needed to handle data transfers, such as through APIs or web services.
   * If there are no complex data communications involved, the DC factor might be 0%, indicating no adjustment to UFP.
   * If there are extensive and complex data communications, the DC factor might be 10% or higher, increasing the UFP to reflect the added effort.
2. **Distributed Data Processing (DDP):** DDP considers the complexity arising from the distribution of data across multiple locations or platforms. Projects dealing with data synchronization or distributed databases can have higher DDP values.
   * If the project involves straightforward data processing within a single location, the DDP factor might be 0%.
   * If the project deals with complex data synchronization between multiple locations or platforms, the DDP factor might be 5% or more.
3. **Performance Requirements (PER):** This factor takes into account performance constraints and response time requirements that may add complexity to the software.
   * If there are no stringent performance requirements, the PER factor might be 0%.
   * If the project requires extremely high performance and low response times, the PER factor might be 5% or higher.
4. **Heavily Used Configuration (HUC):** The HUC factor considers the complexity associated with supporting multiple configurations or environments.
   * If the application has a single configuration and deployment environment, the HUC factor might be 0%.
   * If the application needs to support various configurations and environments, the HUC factor might be 5% or more.
5. **Transaction Rate (TR):** TR reflects the impact of a high transaction rate on the application's complexity. Applications that need to handle a large volume of transactions per unit of time might have a higher TR factor.
   * For applications with a standard transaction rate, the TR factor might be 0%.
   * For applications with a very high transaction rate requirement, the TR factor might be 5% or more.
6. **Explain the concept of productivity in the context of Function Point Analysis. How is it calculated, and what does it measure?**

Productivity in the context of Function Point Analysis (FPA) is a measure of how efficiently a development team or organization is able to produce software by considering the amount of functionality delivered (in terms of Function Points) in relation to the effort expended (typically in person-hours or person-days). It helps assess the efficiency and effectiveness of software development processes and can be a useful metric for tracking and improving the performance of software development teams.

Productivity is calculated using the following formula:

Productivity = Function Points (FP) / Effort (person-hours or person-days)

Where:

- Function Points (FP) represent the size and complexity of the software as measured by FPA.

- Effort is the total number of person-hours or person-days required to develop the software.

Productivity in FPA measures how much functional value or functionality is delivered for each unit of effort invested. It provides insight into whether a development team is working efficiently and whether there is a good balance between the scope of the software being developed and the resources allocated to the project.

Here's how productivity in FPA is interpreted:

**1. High Productivity:** A high productivity value indicates that the development team is efficiently producing software. They are delivering a significant amount of functionality for each unit of effort expended. This can be a positive sign of effective development processes, skilled developers, or the use of efficient development tools and practices.

**2. Low Productivity:** A low productivity value suggests that the development team is less efficient in producing software. They are delivering less functionality for the same amount of effort or are expending more effort to achieve a given level of functionality. This may be due to various factors, such as inefficient processes, inadequate skills, or complex project requirements.

**3. Steady or Improving Productivity:** Monitoring productivity over time can help assess the effectiveness of process improvements or changes in development practices. A steady or improving trend in productivity can indicate that efforts to optimize development processes are yielding positive results.

**4. Declining Productivity:** A declining productivity trend may signal issues in the development process, such as increased complexity, scope creep, or resource constraints. It can serve as an early warning sign for potential problems in the project.

It's important to note that while productivity is a valuable metric, it should be used in conjunction with other performance indicators, such as quality, customer satisfaction, and project timelines, to get a comprehensive view of software development performance.

Additionally, when calculating productivity, it's essential to use consistent and accurate measurements of Function Points and effort. FPA provides a standardized way to measure software size and complexity, making it a useful tool for assessing and improving productivity in software development projects.

1. **Calculate productivity given the Adjusted Function Points (AFP) and Effort in Person-Months (P/M).**

Productivity in the context of software development is typically measured as the number of Adjusted Function Points (AFP) delivered per Person-Month (P/M) of effort. To calculate productivity, you can use the following formula:



Where:

* AFP (Adjusted Function Points) is the total function points adjusted for various factors, as determined using Function Point Analysis.
* Effort in Person-Months (P/M) is the total amount of effort expended on the project, usually measured in person-months. One person-month represents the work done by one person in one month.

To calculate productivity, simply divide the AFP by the effort in person-months:



For example, if you have an AFP of 500 and the project required an effort of 25 person-months, the productivity would be:



So, the productivity in this example is 20 Adjusted Function Points per Person-Month. This metric can help assess how efficiently a software project team is delivering functionality relative to the effort expended. Higher productivity values are generally desirable, as they indicate that more functionality is being delivered for each unit of effort.

1. **Calculate the Documentation per Function Point for a given set of data.**

To calculate the Documentation per Function Point (DPFP), you'll need two pieces of information: the total amount of documentation or documentation effort (e.g., in person-hours or pages), and the total number of Function Points (FP) for the software project. The formula for DPFP is straightforward:

DPFP = Total Documentation Effort / Total Function Points

Let's say you have the following data for a software project:

- Total Documentation Effort: 500 person-hours

- Total Function Points: 200

You can use these values to calculate the Documentation per Function Point (DPFP):

DPFP = 500 person-hours / 200 Function Points

DPFP = 2.5 person-hours per Function Point

So, the Documentation per Function Point for this software project is 2.5 person-hours per Function Point. This means, on average, it took 2.5 person-hours of documentation effort to document each Function Point within the project. This metric can help you assess the level of documentation effort relative to the size of the software, which can be useful for tracking documentation efficiency and resource allocation in the development process.

1. **Calculate the Cost per Function Point for a given set of data.**

To calculate the Cost per Function Point (CPFP), you need to know the total cost of a software project and the total number of Adjusted Function Points (AFP) delivered in that project. The formula to calculate CPFP is as follows:



Where:

* Total Project Cost is the total cost incurred in developing and maintaining the software project.
* Total AFP (Adjusted Function Points) is the total function points adjusted for various factors, as determined using Function Point Analysis.

For example, if the total project cost is $500,000 and the total AFP delivered is 10,000 AFP, you can calculate the CPFP as follows:



So, the Cost per Function Point in this example is $50 per Adjusted Function Point. This metric helps assess the cost efficiency of delivering functionality in a software project. Lower CPFP values are generally more favorable, as they indicate that each function point is being delivered at a lower cost.

1. **What is Earned Value Analysis(EVA)?**

Earned Value Analysis (EVA) is also called “Budget cost of work performed”. It is considered a refinement of the cost-monitoring technique.

This analysis was first carried out USA’s Department of Defence (DOD). In this analysis, a “value” is assigned to each track or work package based on the expenditure forecast.

The value assigned is known as the “planned value (PV)”. The work that has not yet begun is given a value known as the “earned value of zero”.

The total value credited to a project is called “earned value(EV)”, which is also represented as “money value”.

**Methods For Earned Value Analysis**

**0/100 Technique:** The technique where a task is assigned a value of zero until such time that is completed when it is given a value of 100% of the budgeted value.

**50/50 Technique:** The technique in which a task is assigned a 50% value as soon as it is started and then given a value.

**75/25 Technique:** The technique where a task is assigned 75% on starting and 25% on completion.

**Milestone Technique:** The technique where a task is given a value based on the achievement of milestones that have been assigned values as part of the original budget plan.

**Percentage Complete:** In some cases, there may be a way of objectively measuring the amount of work completed.

**Stages in Earned Value Analysis**

**Creating the baseline budget:** This is the first stage in setting up EVA. This budget is based on the project plan. It predicts the earned value through time. Normally, it is measured in person hours or workdays, for example: in a software development project.

**Monitoring Earned Value:** The second stage is monitoring the earned value as the project progresses. This is achieved by monitoring the completion of each task. Actual cost(AC) is the actual cost of each task and it can be analyzed and collected.

**Schedule Variance(SV):** This is the third stage which is measured in cost as EV-PV which is the deviation between planned work and completed work.

Example: Consider these values,

PV =40000

EV=35000

SV=35000-40000 = -5000

Here the calculated SV value is negative and hence we conclude that the project is behind the original schedule.

**Time variance(TV):** The difference between the current time and the time when the achievement of the earned value was planned to occur.

**Cost Variance(CV):** This value is the difference between the actual cost and the earned value. Using this value we can estimate the accuracy of the original cost scheduled for the project. If the CV values are found to be negative, we conclude the project is over cost.

**Advantages of Earned Value Analysis (EVA)**

**Project Performance Measurement:** EVA provides a comprehensive method for measuring and assessing the performance of a project. It helps project managers gain a clear understanding of how well a project is progressing in terms of cost and schedule.

**Objective Performance Metrics:** EVA relies on objective metrics, making it less susceptible to subjective interpretations. This can lead to more accurate assessments of project performance.

**Integration of Cost and Schedule:** EVA combines cost and schedule performance, allowing project managers to see the relationship between these two critical aspects of project management. This integration can help in identifying issues early and making informed decisions.

**Early Issue Identification:** EVA can highlight problems in project execution early, enabling project managers to take corrective actions promptly. This can prevent cost overruns and schedule delays.

**Benchmarking:** EVA allows for benchmarking project performance against planned targets and historical data. It helps project managers assess whether their project is on track compared to similar projects.

**Effective Communication:** EVA provides a standardized way to communicate project performance to stakeholders. Charts and reports generated from EVA data can make it easier for stakeholders to understand the project’s status.

**Disadvantages of Earned Value Analysis (EVA)**

**Complexity:** EVA involves complex calculations and terminology, which can be challenging for project teams to understand, especially for smaller projects or teams with limited expertise.

**Resource-Intensive:** Implementing EVA requires tracking detailed data and maintaining comprehensive records. This can be resource-intensive, and some organizations may lack the necessary tools or resources for effective EVA implementation.

**Subjectivity in Earned Value Calculation:** The “earned value” itself can be subject to interpretation, especially in situations where there are gray areas regarding the completion of work packages or milestones.

**Assumption of Linear Progression:** EVA assumes linear progression, which means that it may not work well for projects with irregular or non-linear progress patterns, such as research and development projects.

**Time-Consuming:** Calculating and updating EVA metrics can be time-consuming, which may not be suitable for projects that require quick decision-making and frequent changes.

**Focus on Metrics vs. Problem Solving:** Overemphasis on EVA metrics can sometimes lead to a focus on numbers rather than addressing the underlying issues causing performance problems.

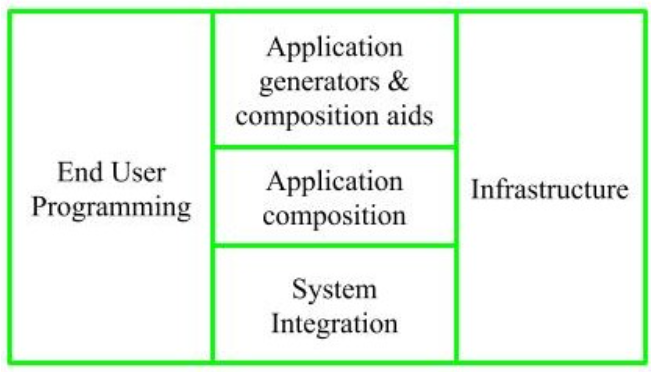
1. **COCOMO-II**

COCOMO-II is the revised version of the original Cocomo (Constructive Cost Model) and is developed at University of Southern California.

Cocomo (Constructive Cost Model) is a regression model based on LOC, i.e number of Lines of Code. It is a procedural cost estimate model for software projects and is often used as a process of reliably predicting the various parameters associated with making a project such as size, effort, cost, time, and quality. It was proposed by Barry Boehm in 1981 and is based on the study of 63 projects, which makes it one of the best-documented models.

COCOMO-II is the model that allows one to estimate the cost, effort and schedule when planning a new software development activity.

It consists of three sub-models:

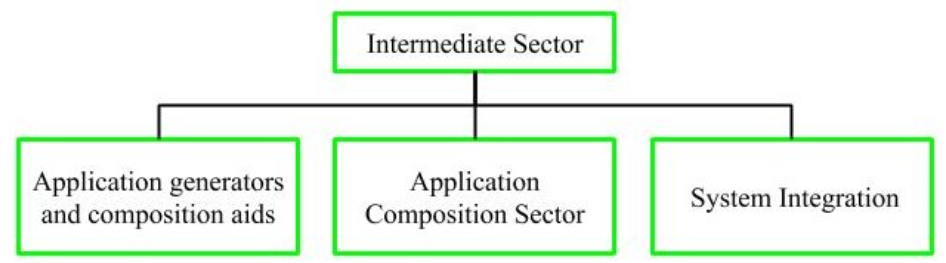


**1. End User Programming:**

Application generators are used in this sub-model. End user write the code by using these application generators.

Example – Spreadsheets, report generator, etc.

**2. Intermediate Sector:**



**(a). Application Generators and Composition Aids –**

This category will create largely prepackaged capabilities for user programming. Their product will have many reusable components. Typical firms operating in this sector are Microsoft, Lotus,

Oracle, IBM, Borland, Novell.

**(b). Application Composition Sector –**

This category is too diversified and to be handled by prepackaged solutions. It includes GUI, Databases, domain specific components such as financial, medical or industrial process control packages.

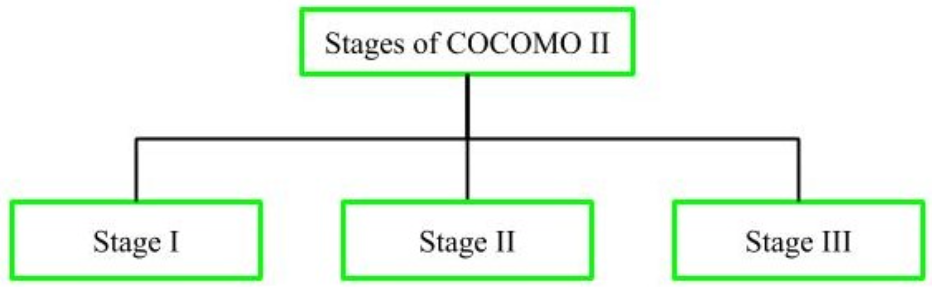
**(c). System Integration –**

This category deals with large scale and highly embedded systems.

**3. Infrastructure Sector:**

This category provides infrastructure for the software development like Operating System, Database Management System, User Interface Management System, Networking System, etc.

Stages of COCOMO II:



Stage-I:

It supports estimation of prototyping. For this it uses Application Composition Estimation Model. This model is used for the prototyping stage of application generator and system integration.

Stage-II:

It supports estimation in the early design stage of the project, when we less know about it. For this it uses Early Design Estimation Model. This model is used in early design stage of application generators, infrastructure, system integration.

Stage-III:

It supports estimation in the post architecture stage of a project. For this it uses Post Architecture Estimation Model. This model is used after the completion of the detailed architecture of application generator, infrastructure, system integration.

1. **Application Composition Estimation Model (COCOMO II | Stage 1)**

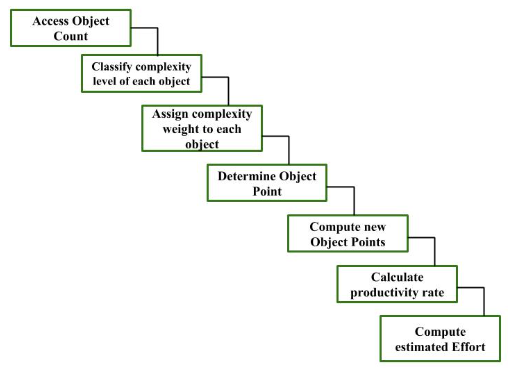
Application Composition Estimation Model allows one to estimate the cost, effort at the stage 1 of the COCOMO II Model.

In this model size is first estimated using Object Points. Object Points are easy to identify and count. Object Points defines screen, reports, third generation (3GL) modules as objects.

Object Point estimation is a new size estimation technique but it is well suited in Application Composition Sector.

**Estimation of Efforts:**

Following steps are taken to estimate effort to develop a project



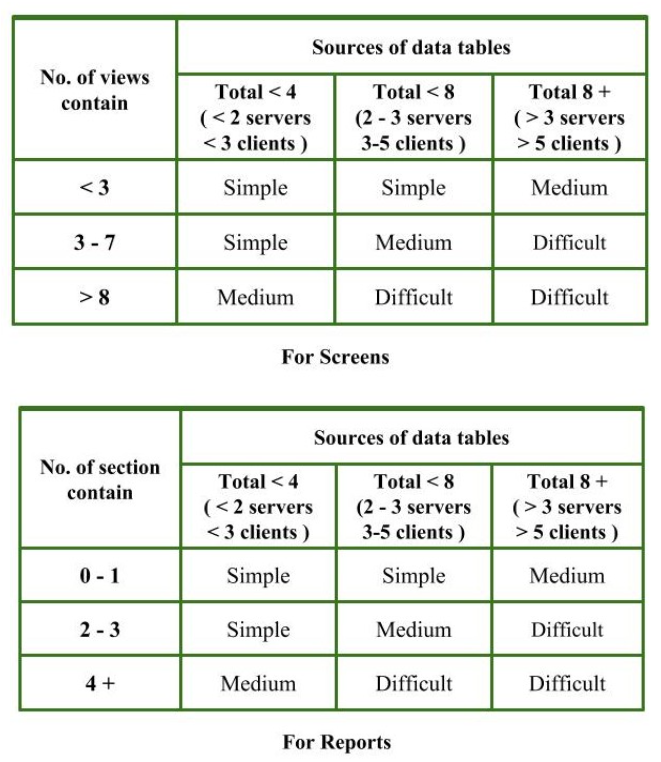
**Step-1: Access Object counts**

Estimate the number of screens, reports and 3GL components that will comprise this application.

**Step-2: Classify complexity levels of each object**

We have to classify each object instance into simple, medium and difficult complexity level depending on values of its characteristics.

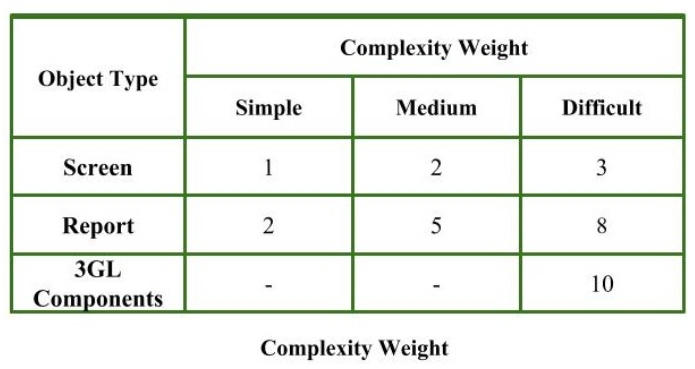
Complexity levels are assigned according to the given table



Step-3: Assign complexity weights to each object

The weights are used for three object types i.e, screens, reports and 3GL components.

Complexity weight are assigned according to object’s complexity level using following table



**Step-4: Determine Object Points**

Add all the weighted object instances to get one number and this is known as object point count.

Object Point

= Sigma (number of object instances)

\* (Complexity weight of each object instance)

**Step-5: Compute New Object Points (NOP)**

We have to estimate the %reuse to be achieved in a project.

Depending on %reuse

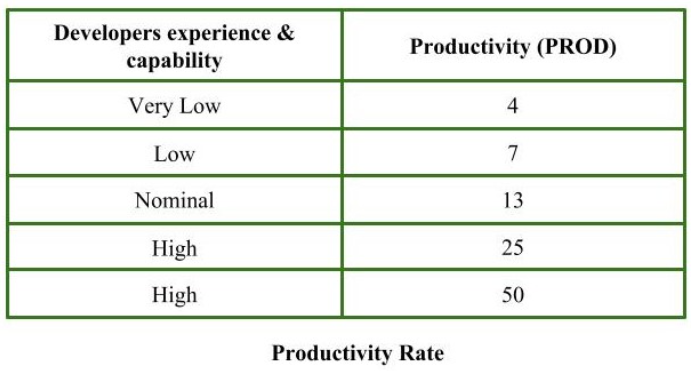
NOP = [(object points) \* (100 - %reuse)]/100

NOP are the object point that will need to be developed and differ from the object point count because there may be reuse of some object instance in project.

**Step-6: Calculate Productivity rate (PROD)**

Productivity rate is calculated on the basis of information given about developer’s experience and capability.

For calculating it, we use following table



**Step-7: Compute the estimated Effort**

Effort to develop a project can be calculated as

Effort = NOP/PROD

Effort is measured in person-month.

For solved example on above question:-

<https://www.geeksforgeeks.org/software-engineering-application-composition-estimation-model-cocomo-ii-stage-1/>

**MODULE 4**

**Design Process & quality, Design Concepts,**

**The design Model, Pattern-based Software**

**Design. 4.2 Architectural Design :Design**

**Decisions, Views, Patterns, Application**

**Architectures, Modeling Component level**

**Design: component, Designing class based**

**components, conducting component-level**

**design, User Interface Design: The golden**

**rules, Interface Design steps & Analysis,**

**Design Evaluation(Syllabus)**

**What are the golden rules for User Interface Design? Explain the steps for designing an User Interface.(QB)**

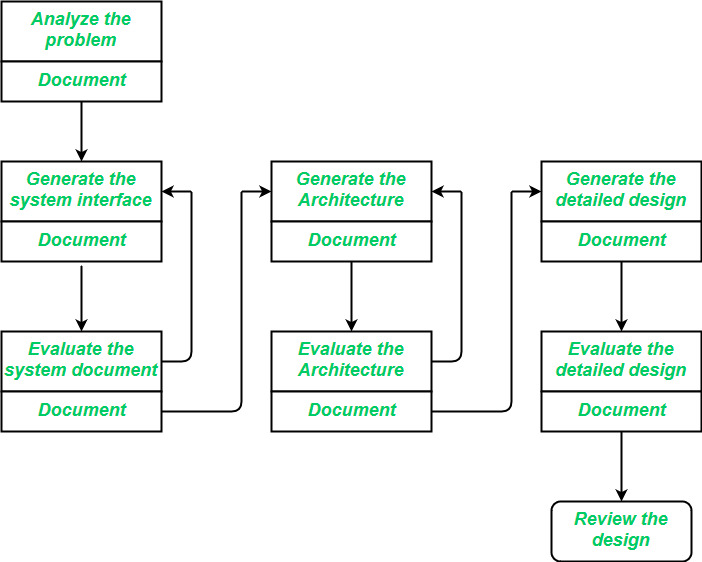
**Q) General Overview of Design.**

The design phase of software development deals with transforming the customer requirements as described in the SRS documents into a form implementable using a programming language. The software design process can be divided into the following three levels of phases of design:

1. Interface Design
2. Architectural Design
3. Detailed Design

Elements of a System:

1. Architecture – This is the conceptual model that defines the structure, behavior, and views of a system. We can use flowcharts to represent and illustrate the architecture.
2. Modules – These are components that handle one specific task in a system. A combination of the modules makes up the system.
3. Components – This provides a particular function or group of related functions. They are made up of modules.
4. Interfaces – This is the shared boundary across which the components of a system exchange information and relate.
5. Data – This is the management of the information and data flow.

  
   
Interface Design: Interface design is the specification of the interaction between a system and its environment. This phase proceeds at a high level of abstraction with respect to the inner workings of the system i.e, during interface design, the internal of the systems are completely ignored and the system is treated as a black box. Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts. The design problem statement produced during the problem analysis step should identify the people, other systems, and devices which are collectively called agents. Interface design should include the following details:

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

Architectural Design: Architectural design is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them. In architectural design, the overall structure of the system is chosen, but the internal details of major components are ignored. Issues in architectural design includes:

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

The architectural design adds important details ignored during the interface design. Design of the internals of the major components is ignored until the last phase of the design.

Detailed Design: Design is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures. The detailed design may include:

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

**Q) Architectural Design**

**Software architecture**

o The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is architectural design.

o The output of this design process is a description of the software architecture.

**Architectural design**

o An early stage of the system design process.

o Represents the link between specification and design processes.

o Often carried out in parallel with some specification activities.

o It involves identifying major system components and their communications.

**Architectural abstraction**

· Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components.

· Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies.

Advantages of explicit architecture

**Stakeholder communication**

Architecture may be used as a focus of discussion by system stakeholders.

**System analysis**

Means that analysis of whether the system can meet its non-functional requirements is possible.

**Large-scale reuse**

The architecture may be reusable across a range of systems

Product-line architectures may be developed.

**Architectural representations**

* Simple, informal block diagrams showing entities and relationships are the most frequently used method for documenting software architectures.
* But these have been criticized because they lack semantics, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
* Depends on the use of architectural models.The requirements for model semantics depends on how the models are used.

**Box and line diagrams**

* Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
* However, it is useful for communication with stakeholders and for project planning.

**Use of architectural models**

As a way of facilitating discussion about the system design

* A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail.

As a way of documenting an architecture that has been designed

* The aim here is to produce a complete system model that shows the different components in a system, their interfaces and their connections.

**Architectural design decisions**

* Architectural design is a creative process so the process differs depending on the type of system being developed.
* However, a number of common decisions span all design processes and these decisions affect the non-functional characteristics of the system.

**Questions:**

* Is there a generic application architecture that can be used?
* How will the system be distributed?
* What architectural styles are appropriate?
* What approach will be used to structure the system?
* How will the system be decomposed into modules?
* What control strategy should be used?
* How will the architectural design be evaluated?
* How should the architecture be documented?

**Architecture reuse**

* Systems in the same domain often have similar architectures that reflect domain concepts.
* Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
* The architecture of a system may be designed around one of more architectural patterns or ‘styles’.

· These capture the essence of an architecture and can be instantiated in different ways.

**Architecture and system characteristics**

* Performance

Localize critical operations and minimize communications. Use large rather than fine-grain components.

* Security

Use a layered architecture with critical assets in the inner layers.

* Safety

Localise safety-critical features in a small number of sub-systems.

* Availability

Include redundant components and mechanisms for fault tolerance.

* Maintainability

Use fine-grain, replaceable components.

**Architectural views**

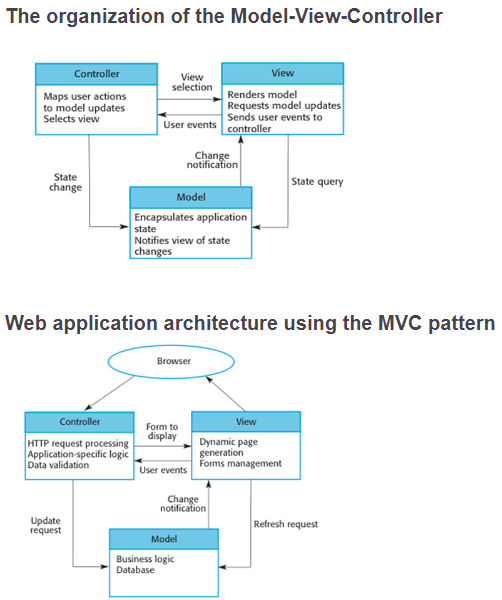
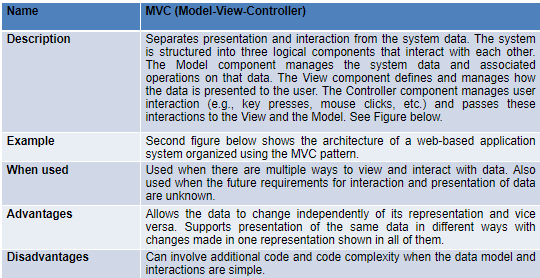
* What views or perspectives are useful when designing and documenting a system’s architecture?
* What notations should be used for describing architectural models?
* Each architectural model only shows one view or perspective of the system.

It might show how a system is decomposed into modules, how the run-time processes interact or the different ways in which system components are distributed across a network. For both design and documentation, you usually need to present multiple views of the software architecture.

**4 + 1 view model of software architecture**

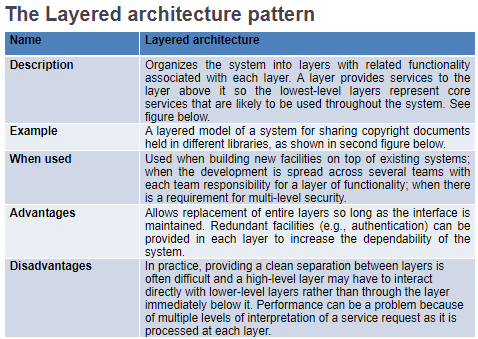
* A logical view, which shows the key abstractions in the system as objects or object classes.
* A process view, which shows how, at run-time, the system is composed of interacting processes.
* A development view, which shows how the software is decomposed for development.
* A physical view, which shows the system hardware and how software components are distributed across the processors in the system.
* Related using use cases or scenarios (+1)

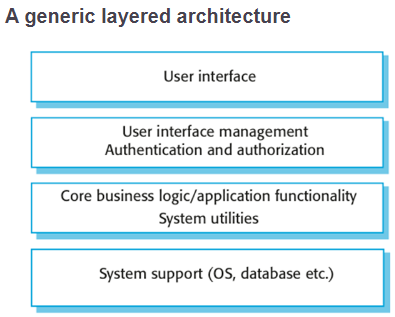
**Architectural patterns**

* Patterns are a means of representing, sharing and reusing knowledge.
* An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
* Patterns should include information about when they are and when they are not useful.
* Patterns may be represented using tabular and graphical descriptions.
* **The Model-View-Controller (MVC) pattern**

**Layered architecture**

* Used to model the interfacing of subsystems.
* Organizes the system into a set of layers (or abstract machines) each of which provide a set of services.
* Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
* However, it is often artificial to structure systems in this way.





**Key points**

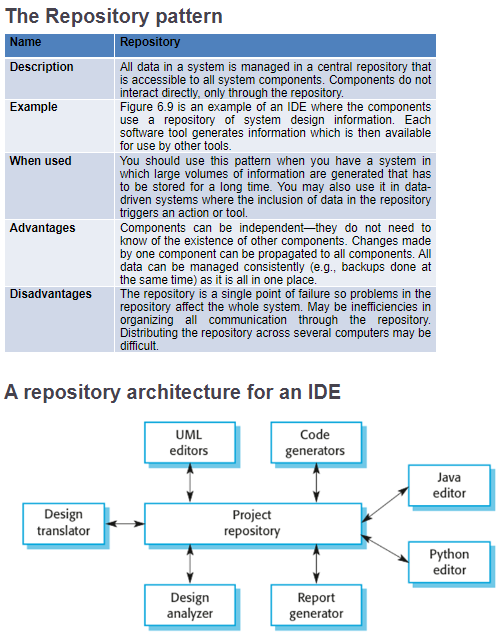
* A software architecture is a description of how a software system is organized.
* Architectural design decisions include decisions on the type of application, the distribution of the system, the architectural styles to be used.
* Architectures may be documented from several different perspectives or views such as a conceptual view, a logical view, a process view, and a development view.
* Architectural patterns are a means of reusing knowledge about generic system architectures. They describe the architecture, explain when it may be used and describe its advantages and disadvantages.

**Repository architecture**

o Subsystems must exchange data. This may be done in two ways:

* Shared data is held in a central database or repository and may be accessed by all sub-systems;
* Each sub-system maintains its own database and passes data explicitly to other sub-systems.

o When large amounts of data are to be shared, the repository model of sharing is most commonly used as this is an efficient data sharing mechanism.



**Application architectures**

* Application systems are designed to meet an organizational need.
* As businesses have much in common, their application systems also tend to have a common architecture that reflects the application requirements.
* A generic application architecture is an architecture for a type of software system that may be configured and adapted to create a system that meets specific requirements.

Use of application architectures

* As a starting point for architectural design.
* As a design checklist.
* As a way of organizing the work of the development team.
* As a means of assessing components for reuse.
* As a vocabulary for talking about application types.

**Examples of application types**

* Data processing applications
  + Data driven applications that process data in batches without explicit user intervention during the processing.
* Transaction processing applications
  + Data-centered applications that process user requests and update information in a system database.
* Event processing systems
  + Applications where system actions depend on interpreting events from the system’s environment.
* Language processing systems
  + Applications where the users’ intentions are specified in a formal language that is processed and interpreted by the system.

**Application type examples**

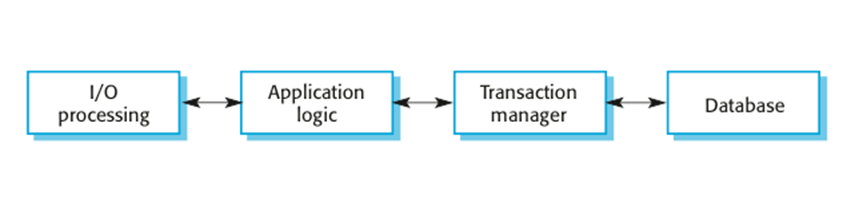
·Focus here is on transaction processing and language processing systems.

* Transaction processing systems
  + E-commerce systems;
  + Reservation systems.
* Language processing systems
  + Compilers;
  + Command interpreters.

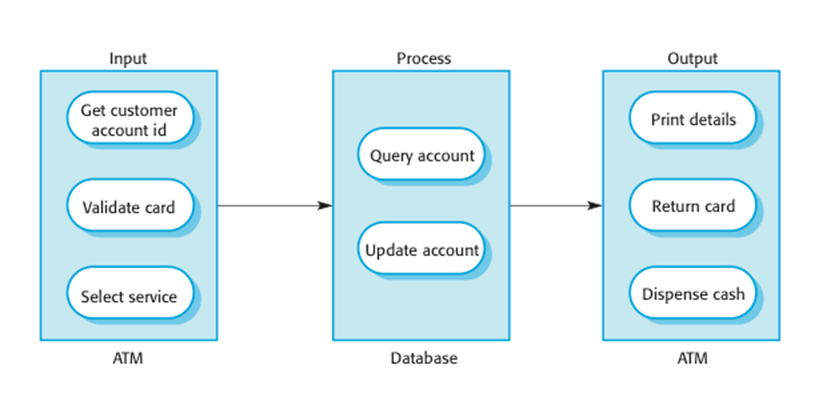
**Transaction processing systems**

* Process user requests for information from a database or requests to update the database.
* From a user perspective a transaction is:
  + Any coherent sequence of operations that satisfies a goal;
  + For example - find the times of flights from London to Paris.
* Users make asynchronous requests for service which are then processed by a transaction manager.

The structure of transaction processing applications



The software architecture of an ATM system



References:- <https://csis.pace.edu/~marchese/SE616_New/L6/L6.htm>

<https://www.geeksforgeeks.org/software-engineering-architectural-design/>

**Component Level Design:-**

**Component-level design** occurs after the first iteration of architectural design has been completed. At this stage, the overall data and program structure of the software has been established. The intent is to translate the design model into operational software.

**Component**

A component is a modular building block for computer software. More formally, the OMG Unified Modeling Language Specification [OMG03a] defines a component as **“. . . a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces.”** components populate the software architecture and, as a consequence, play a role in achieving the objectives and requirements of the system to be built. Because components reside within the software architecture, they must communicate and collaborate with other components and with entities (e.g., other systems, devices, people) that exist outside the boundaries of the software.

**DESIGNING CLASS -BASED COMPONENTS**

Four basic design principles are applicable to component-level design and have been widely adopted when object-oriented software engineering is applied. The underlying motivation for the application of these principles is to create designs that are more amenable to change and to reduce the propagation of side effects when changes do occur.

* The Liskov Substitution Principle (LSP).
* Dependency Inversion Principle (DIP).
* The Interface Segregation Principle (ISP).

Although component-level design principles provide useful guidance, components themselves do not exist in a vacuum. In many cases, individual components or classes are organized into subsystems or packages. It is reasonable to ask how this packaging activity should occur. Exactly how should components be organized as the design

proceeds? Martin [Mar00] suggests additional packaging principles that are applicable to component-level design:

* The Release Reuse Equivalency Principle (REP).
* The Common Closure Principle (CCP).
* The Common Reuse Principle (CRP).

**Component-Level Design Guidelines:**

In addition to the principles discussed in Section 10.2.1, a set of pragmatic design guidelines can be applied as component-level design proceeds.

These guidelines apply to components, their interfaces, and the dependencies and inheritance characteristics that have an impact on the resultant design. Ambler [Amb02b] suggests the following guidelines:

* Components. Naming conventions should be established for

components that are specified as part of the architectural model and

then refined and elaborated as part of the component-level model.

* Interfaces.
* Dependencies and Inheritance.

**CONDUCTING COMPONENT-LEVEL DESIGN**

The following steps represent a typical task set for component-level design, when it is applied for an object-oriented system.

Step 1. Identify all design classes that correspond to the problem domain.

Step 2. Identify all design classes that correspond to the infrastructure domain.

Step 3. Elaborate all design classes that are not acquired as reusable components.

Step 3a. Specify message details when classes or components collaborate

Step 3b. Identify appropriate interfaces for each component.

Step 3c. Elaborate attributes and define data types and data structures required to implement them.

Step 3d. Describe processing flow within each operation in detail.

Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.

Step 5. Develop and elaborate behavioral representations for a class or component.

Step 6. Elaborate deployment diagrams to provide additional implementation detail.

Step 7. Refactor every component-level design representation and always consider alternatives.

(for more reference   
Go to d15b ppt page 1-26  
Also <https://cuitutorial.com/component-level-design/>)

**USER INTERFACE DESIGN:(VVIMP Entire section hereafter)  
(only interface design question is included in qb do it properly)**

User interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

**Three golden rules:(VVIMP)**

1. Place the user in control.

2. Reduce the user’s memory load.

3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface

design principles that guide this important aspect of software design.

**Place the User in Control:**

* Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
* Provide for flexible interaction.
* Allow user interaction to be interruptible and undoable.
* Streamline interaction as skill levels advance and allow the

interaction to be customized.

* Hide technical internals from the casual user.
* Design for direct interaction with objects that appear on the screen.

**Reduce the User’s Memory Load:**

* Reduce demand on short-term memory.
* Establish meaningful defaults.
* Define shortcuts that are intuitive.
* The visual layout of the interface should be based on a real-world

metaphor.

* Disclose information in a progressive fashion.

**Make the Interface Consistent:**

* Allow the user to put the current task into a meaningful context.
* Maintain consistency across a family of applications.
* If past interactive models have created user expectations, do not

make changes unless there is a compelling reason to do so.

**Interface design Steps:**

Although many different user interface design models (e.g., [Nor86], [Nie00]) have been proposed, all suggest some combination of the following steps:

1. Using information developed during interface analysis, define interface objects and actions (operations).

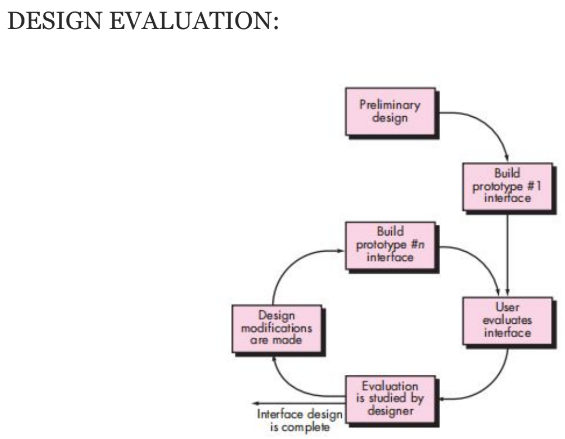
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.

3. Depict each interface state as it will actually look to the end user.

4. Indicate how the user interprets the state of the system from information provided through the interface.

**DESIGN EVALUATION:**

Once you create an operational user interface prototype, it must be evaluated to determine whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal “test drive,” in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end users.



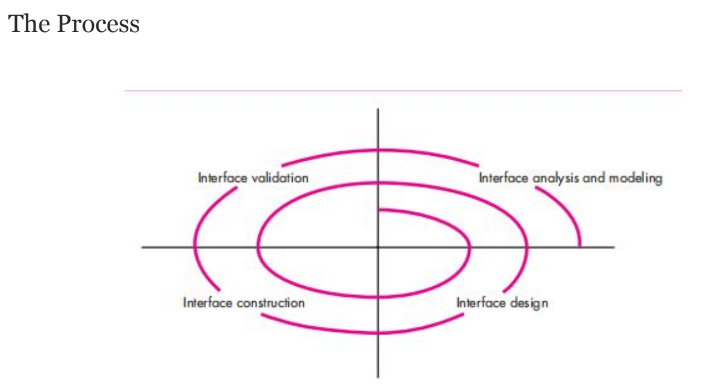
**DESIGN EVALUATION:**

* The user interface evaluation cycle takes the form shown in Figure. After the design model has been completed, a first-level prototype is created.
* The prototype is evaluated by the user, who provides you with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), you can extract information from these data (e.g., 80 percent of all users did not like the mechanism for saving data files).
* Design modifications are made based on user input, and the next level a prototype is created.
* The evaluation cycle continues until no further modifications to the interface design is necessary.

**USER INTERFACE ANALYSIS AND DESIGN:**

The overall process for analyzing and designing a user interface begins with the creation of different models of system function (as perceived from the outside). You begin by delineating the human- and computer-oriented tasks that are required to achieve system function and then considering the design issues that apply to all interface designs.

Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.

****

**INTERFACE ANALYSIS:-**

A key tenet of all software engineering process models is this: understand the problem before you attempt to design a solution. In the case of user interface design, understanding the problem means understanding

(1) the people (end users) who will interact with the system through the interface.

(2) the tasks that end users must perform to do their work.

(3) the content that is presented as part of the interface.

(4) the environment in which these tasks will be conducted.

**INTERFACE ANALYSIS:**

**User Analysis:**

* User Interviews.
* Sales input.
* Marketing input.
* Support input.

**Task Analysis and Modeling:**

* Use cases.
* Task elaboration.
* Object elaboration.
* Workflow analysis.
* Hierarchical representation.

**Analysis of Display Content**

**Analysis of the Work Environment**

For reference:- <https://www.geeksforgeeks.org/software-engineering-user-interface-design/>

**MODULE 5**

**Risk Identification**

**Q) What are risks? Classification of risks?**

1. "Risk" refers to a situation that could result in a loss or jeopardies as the project's progress but has not yet occurred.
2. These potential issues might harm cost, schedule or technical success of the project and the quality of our software device, or project team morale.

**There are various classifications of risks like:**

* **Reactive risk:** the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a fire-fighting mode. When this fails, “crisis management” takes over and the project is in real jeopardy
* **Proactive Risk:** A proactive strategy begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner.
* **Known risks:** Those risks that can be uncovered after careful assessment of the project program, the business and technical environment in which the plan is

being developed, and more reliable data sources (e.g.,unrealistic delivery date)

* **Predictable risks:** Those risks that are hypothesized from previous project experience (e.g., past turnover)
* **Unpredictable risks:** Those risks that can and do occur, but are extremely tough to identify in advance.
* Project risks are those that have an impact on the project's schedule or resources.
* Product risks affect the quality or performance of the product being developed.
* Business risks are risks to the corporation developing or licensing the software.
* Generic risks are a potential threat to every software project.
* Product-specific risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built.

**Requirements Volatility:** Changes in project requirements can lead to scope

creep, increased development time, and added costs. Managing changes

effectively is crucial to mitigate this risk.

**Technical Risks:**

• Technological Obsolescence: The technology stack chosen for a project

may become outdated during development, leading to compatibility and

maintenance issues.

• Complexity: Overly complex software designs can lead to difficulties in

implementation, testing, and maintenance.

**Schedule Risks:**

• Unrealistic Deadlines: Setting overly aggressive deadlines can result in rushed development, which may lead to errors and increased technical debt.

• Resource Constraints: Inadequate staffing or resources can cause delays

in development.

**Quality Risks:**

• Bugs and Defects: Poor code quality can result in software defects, leading to increased maintenance efforts and potential security vulnerabilities.

• Security Vulnerabilities: Failing to address security concerns can result in

data breaches and other security incidents.

**Integration Risks:**

• Third-Party Dependencies: Reliance on third-party libraries or services

can introduce risks if those dependencies change or become unavailable.

• Interoperability Issues: Integration challenges between different software

components or systems can lead to functionality gaps.

**Communication Risks:**

• Miscommunication: Poor communication among team members or with

stakeholders can result in misunderstandings and misaligned expectations.

• Cultural and Language Differences: In global development teams,

differences in culture and language can lead to communication

breakdowns.

**Resource Risks:**

• Staff Turnover: Losing key team members during a project can disrupt

progress and impact knowledge transfer.

• Resource Constraints: Limited budget or access to necessary tools can

hinder development efforts.

**Legal and Compliance Risks:**

• Intellectual Property Issues: Failure to respect intellectual property rights can

lead to legal disputes.

• Regulatory Compliance: Non-compliance with industry regulations or data

protection laws can result in penalties.

**Scalability and Performance Risks:** Inadequate planning for scalability and

performance can lead to system bottlenecks and poor user experiences as user

loads grow.

**Maintenance and Technical Debt:** Neglecting software maintenance can

accumulate technical debt, making future development and enhancements more

challenging and costly.

To manage these risks effectively, software engineering teams typically

engage in risk assessment and mitigation activities, such as risk

identification, risk analysis, risk prioritization, and the development of risk

mitigation plans. It's essential to continuously monitor and adapt these plans

throughout the software development lifecycle to reduce the impact of

potential risks.

**Q) Risk management**

****

Risk management in software engineering is the process of identifying, analyzing,

prioritizing, and mitigating risks that can potentially impact the success of a software project. Effective risk management helps software development teams anticipate and address potential problems before they become critical issues, ultimately leading to a more successful and predictable project outcome.

Key steps involved in risk management in software engineering:

**Risk Identification:**

Risk Identification:The first step is to identify potential risks that could affect the project. This involves brainstorming sessions, review of project documentation, and past project experiences.

Risk Categorization: Risks can be categorized into various types, such as technical, schedule, cost, quality, and external risks.

Risk Analysis:

• Risk Assessment: Once identified, risks are assessed in terms of their

probability (likelihood of occurrence) and impact (severity of consequences).

• Risk Prioritization: Risks are then prioritized based on their significance.

High-impact and high-probability risks are given the highest priority.

**Risk Mitigation Planning:**

• Risk Mitigation Strategies: For each identified risk, the team develops specific

strategies to mitigate or reduce its impact. Strategies may include preventive

actions to reduce the likelihood of occurrence or contingency plans to address

the consequences if the risk materializes.

• Allocating Resources: Resources (budget, time, personnel) are allocated for

risk mitigation activities

**Risk Monitoring and Control:**

• Regular Monitoring: The project team monitors the identified risks throughout

the software development lifecycle. This involves tracking risk indicators and

assessing whether the risks are evolving as expected.

• Risk Response Execution: If a risk materializes or changes in severity, the

predefined mitigation strategies are executed.

**Documentation and Communication:**

**• Risk Register:** All identified risks and their associated details (probability, impact, mitigation plans) are documented in a risk register.

**• Communication:** Risk information is communicated to relevant stakeholders, including team members, project managers, and clients, to ensure everyone is

aware of potential project challenges.

**• Continuous Improvement:**

After each project phase or iteration, lessons learned from risk management are

reviewed and used to improve future risk management processes.

Risk management practices should be flexible and adaptable, allowing teams to

respond to evolving project conditions and emerging risks.

**• Contingency Planning:**

In cases where high-impact risks cannot be completely mitigated, contingency

plans are put in place to define how the project will respond if the risk occurs. This

includes predefined actions to minimize the impact and ensure project progress.

**• Risk Reporting:**

Regular reports are generated to provide project stakeholders with updates on the

status of identified risks and the effectiveness of risk mitigation strategies.

**Q) RMMM**

**PPT ANS**

Risk Mitigation, Monitoring and Management(RMMM) Plan

• In most cases, a risk management approach can be found in the software project plan. This can be broken down into three sections: risk mitigation, monitoring, and management (RMMM). All work is done as part of the risk analysis in this strategy. The project manager typically uses this RMMM plan as part of the overall project plan.

• Some development teams use a Risk Information Sheet(RIS) to document risk. For faster information handling, such as creation, priority sorting, searching, and other analyses, this RIS is controlled by a database system. Risk mitigation and monitoring will begin after the RMMM is documented and the project is launched.

Risk Mitigation, Monitoring, and Management Plan (RMMM)

• Project Name: Online Shopping Website Development

**Risk Identification:**

• **Requirement Changes**

• Probability: High

• Impact: High

• Description: Due to evolving client needs, there is a high likelihood of

requirement changes during the project.

**• Technical Skill Gaps**

• Probability: Medium

• Impact: Medium

• Description: Some team members lack experience with specific technologies

required for the project.

**Third-Party Dependency**

• Probability: Low

• Impact: High

• Description: We rely on a third-party payment gateway, which may have

service interruptions.

**Risk Analysis:**

**• Requirement Changes:**

• Priority: High

• Mitigation Strategy: Frequent client communication, robust change

management process, and detailed requirement documentation.

• Contingency Plan: Allocate additional time and budget for handling

changes.

**• Technical Skill Gaps:**

• Priority: Medium

• Mitigation Strategy: Provide training sessions for team members on

required technologies, consider hiring a consultant.

• Contingency Plan: Reallocate tasks among team members, adjust the

project schedule.

**Third-Party Dependency:**

• Priority: Low

• Mitigation Strategy: Identify alternative payment gateways, establish a backup

plan in case of service interruptions.

• Contingency Plan: Switch to an alternative payment gateway temporarily.

**Risk Monitoring and Control:**

• **Requirement Changes:**

• Regular client meetings to capture changes.

• Review requirements document and update as needed.

• Track scope changes in project management software.

**• Technical Skill Gaps:**

• Monitor team members' progress in skill development.

• Conduct periodic assessments to identify skill gaps.

• Adjust training sessions based on the team's progress.

**Third-Party Dependency:**

• Monitor the performance of the third-party payment gateway.

• Implement automated notifications for service interruptions.

• Maintain contact with alternative payment gateway providers.

**Reporting:**

• Regular Risk Reports: Weekly meetings to review the status of identified

risks, including their probability, impact, and current mitigation status.

• Exception Reports: Immediate reporting for high-impact and high-probability

risks that materialize.

**Continuous Improvement:**

• After each project phase, conduct a review to identify any new risks or

changes to existing risks.

• Document lessons learned and update the RMMM plan for future projects.

**GFG ANS**

A risk management technique is usually seen in the software Project plan. This can be divided into Risk Mitigation, Monitoring, and Management Plan (RMMM). In this plan, all works are done as part of risk analysis. As part of the overall project plan project manager generally uses this RMMM plan.

In some software teams, risk is documented with the help of a Risk Information Sheet (RIS). This RIS is controlled by using a database system for easier management of information i.e creation, priority ordering, searching, and other analysis. After documentation of RMMM and start of a project, risk mitigation and monitoring steps will start.

**Risk Mitigation :**

It is an activity used to avoid problems (Risk Avoidance).

Steps for mitigating the risks as follows:

1. Finding out the risk.
2. Removing causes that are the reason for risk creation.
3. Controlling the corresponding documents from time to time.
4. Conducting timely reviews to speed up the work.

**Risk Monitoring :**

It is an activity used for project tracking.

It has the following primary objectives as follows:

1. To check if predicted risks occur or not.
2. To ensure proper application of risk aversion steps defined for risk.
3. To collect data for future risk analysis.
4. To allocate what problems are caused by which risks throughout the project.

**Risk Management and planning :**

It assumes that the mitigation activity failed and the risk is a reality. This task is done by Project manager when risk becomes reality and causes severe problems. If the project manager effectively uses project mitigation to remove risks successfully then it is easier to manage the risks. This shows that the response that will be taken for each risk by a manager. The main objective of the risk management plan is the risk register. This risk register describes and focuses on the predicted threats to a software project.

Example:

Let us understand RMMM with the help of an example of high staff turnover.

**Risk Mitigation:**

To mitigate this risk, project management must develop a strategy for reducing turnover. The possible steps to be taken are:

1. Meet the current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).
2. Mitigate those causes that are under our control before the project starts.
3. Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.
4. Organize project teams so that information about each development activity is widely dispersed.
5. Define documentation standards and establish mechanisms to ensure that documents are developed in a timely manner.
6. Assign a backup staff member for every critical technologist.

**Risk Monitoring:**

As the project proceeds, risk monitoring activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. In the case of high staff turnover, the following factors can be monitored:

* General attitude of team members based on project pressures.
* Interpersonal relationships among team members.
* Potential problems with compensation and benefits.
* The availability of jobs within the company and outside it.

**Risk Management:**

Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. Continuing the example, the project is well underway, and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team. In addition, the project manager may temporarily refocus resources (and readjust the project schedule) to those functions that are fully staffed, enabling newcomers who must be added to the team to “get up to the speed“.

**Drawbacks of RMMM:**

1. It incurs additional project costs.
2. It takes additional time.
3. For larger projects, implementing an RMMM may itself turn out to be another tedious project.
4. RMMM does not guarantee a risk-free project, infact, risks may also come up after the project is delivered.

**Software Configuration management**

Software Configuration Management (SCM) is a set of practices and

processes used in software development to systematically manage and

control software artifacts, track changes, and ensure the integrity of software

configurations throughout their lifecycle. SCM helps teams collaborate

effectively, maintain version history, and streamline the development process.

**1. SCM Repositories:** SCM repositories are central locations where software

artifacts, such as source code, documentation, binary files, and configuration

files, are stored, organized, and managed.

**There are two primary types of SCM repositories:**

1. **Version Control Repositories:** These repositories are used to manage

source code and related assets. Popular version control systems (VCS) like

Git, Subversion (SVN), and Mercurial are commonly used for tracking

changes, branching, and merging in these repositories.

1. **Artifact Repositories:** These repositories store binary artifacts, libraries,

dependencies, and other files necessary for building and deploying software.

Examples include Nexus, Artifactory, and Docker Hub for container images.

• SCM repositories provide version history, access control, and audit trails for

all changes made to the stored artifacts. Developers can retrieve and commit

changes to these repositories, ensuring a centralized and controlled

development environment.

**2. SCM Process:** SCM encompasses several key processes to manage

software configurations effectively:

**• Configuration Identification:** In this phase, SCM defines and identifies the

software configuration items (SCIs). SCIs are the individual elements that

make up the software, such as source code files, documentation, libraries,

and configuration files. Proper identification ensures that all necessary

components are tracked and controlled.

**• Change Control:** Change control processes establish procedures for

requesting, reviewing, and approving changes to software configurations.

These changes can include bug fixes, feature additions, or updates. Change

control ensures that changes are carefully considered and documented,

reducing the risk of introducing errors or instability into the software.

**• Configuration Baseline:** A configuration baseline is a snapshot of the entire

software configuration at a specific point in time. It represents a stable and

known state of the software, typically used as a reference for quality

assurance, testing, and deployment. Baselines help ensure that specific

versions of the software are used for testing and production.

**• Configuration Management Planning:** This involves creating a plan that

outlines how SCM will be implemented within the software development

process. It defines roles and responsibilities, procedures, tools, and the

overall strategy for managing configurations.

**• Build and Release Management:** SCM encompasses the processes for

creating builds and managing releases. A build is a compiled version of the

software that is ready for testing or deployment. Release management

involves controlling the distribution and deployment of software to various

environments (e.g., development, testing, production).

**• Change Auditing and Reporting:** SCM maintains records of all changes

made to the software, who made them, when they were made, and why. This

information is crucial for traceability, accountability, and compliance with

industry standards and regulations.

**• Branching and Merging:** In larger software projects, multiple development

teams or individuals may work on different features or bug fixes concurrently.

Branching and merging strategies within version control systems allow for

isolation of work and later integration, helping teams collaborate efficiently.

**• Backup and Recovery:** SCM systems often include backup and recovery

mechanisms to prevent data loss in case of hardware failure or accidental

deletion. This ensures the integrity and availability of configuration data.

**• Continuous Integration and Continuous Delivery (CI/CD):** SCM plays a

critical role in CI/CD pipelines by enabling the automation of build, test, and

deployment processes. Changes are automatically integrated, tested, and

delivered to various environments, reducing the risk of integration problems

and streamlining software delivery.

**• Software Configuration Management** is a systematic approach to managing

and controlling software artifacts throughout their lifecycle. It ensures that

changes are well-documented, coordinated, and integrated, resulting in a

more organized, stable, and maintainable software development process.

Effective SCM practices are essential for software quality, collaboration, and

project success.

**Q) What is Software Quality AssuranceTask and Plan,Metrics, Software Reliability?**

1. **SQA**

Software Quality Assurance (SQA) is a systematic process within software

development that focuses on ensuring that software products and processes

meet established quality standards and objectives. SQA tasks and plans are

essential components of this process:

**• SQA Tasks:** SQA tasks involve a range of activities aimed at monitoring and

improving the quality of software throughout its lifecycle. These tasks typically

include:

**• Requirements Analysis:** Reviewing and validating the accuracy and

completeness of software requirements.

**• Test Planning:** Developing a comprehensive plan for testing the software,

including defining test objectives, strategies, and test cases.

**• Code Reviews:** Evaluating the source code for adherence to coding

standards, best practices, and identifying defects.

**• Testing:** Executing test cases, including unit testing, integration testing,

system testing, and user acceptance testing.

**• Defect Tracking:** Monitoring and documenting defects or issues discovered

during testing and development.

**• Process Audits:** Assessing and improving development processes to ensure

compliance with quality standards and best practices.

**• Documentation Review:** Verifying that documentation, such as user

manuals and technical documentation, is accurate and up to date.

**• Release Management:** Ensuring that software releases are well-planned,

tested, and documented.

**• SQA Plan:** An SQA plan outlines the strategies, objectives, and activities that

will be employed to achieve quality assurance throughout the software

development project. It typically includes:

**• Scope and Objectives:** Describing the scope of SQA activities and the

quality objectives to be achieved.

**• SQA Team Roles and Responsibilities:** Defining the roles and

responsibilities of team members involved in SQA.

**• Testing and Review Procedures:** Detailing the testing methodologies,

review processes, and criteria for acceptance.

**• Resources:** Identifying the resources, tools, and infrastructure required for

SQA.

**• Schedule:** Outlining the timeline and milestones for SQA activities.

**• Metrics and Measurements:** Describing the metrics and key performance

indicators (KPIs) that will be used to assess quality.

**• Risk Management:** Identifying potential risks to quality and mitigation

strategies.

**• Documentation:** Specifying the documentation standards and templates to

be used for recording SQA activities.

1. **Software Metrics:**

• Software metrics are quantifiable measurements used to assess various

aspects of software development, quality, and performance. These metrics

provide valuable insights into the software project's health and progress.

**Some common software metrics include:**

**• Code Coverage:** Measures the percentage of code that is tested by

automated test cases, helping evaluate the comprehensiveness of testing.

**• Defect Density:** Calculates the number of defects or issues found per unit of

code (e.g., lines of code), indicating code quality.

**• Velocity:** Used in Agile development, velocity measures the amount of work

completed by a development team in a specific time period (e.g., a sprint).

**• Lead Time:** Measures the time it takes from the initiation of a software

request or user story to its completion.

**• Cycle Time:** Measures the time it takes to complete a specific task or

process within the software development lifecycle.

**• Effort Variance:** Compares estimated effort with actual effort expended on a

project, highlighting deviations from the plan.

**• Bug Severity and Priority:** Classifies and prioritizes defects based on their

severity and impact on the software.

**• Customer Satisfaction:** Collects feedback from users or stakeholders to

gauge their satisfaction with the software.

• Effective use of software metrics can help teams identify areas for

improvement, make informed decisions, and ensure that software quality

objectives are met.

1. **Software Reliability:**

• Software reliability refers to the ability of a software system to perform its

intended functions without failures or errors under specified conditions for a

specified period. Key aspects of software reliability include:

**• Availability:** It measures how often the software is operational and

accessible to users. High availability implies that the software is rarely down

or experiencing outages.

**• Fault Tolerance:** A reliable software system can continue functioning even

when certain components or subsystems fail. Fault tolerance mechanisms,

such as redundancy, help achieve this.

**• Mean Time Between Failures (MTBF):** MTBF quantifies the average time

between failures in the software. Higher MTBF values indicate greater

reliability.

**• Mean Time to Recovery (MTTR):** MTTR measures the average time it takes

to recover from a failure or downtime. Lower MTTR values indicate faster

recovery and, therefore, better reliability.

**• Failure Rate:** This metric represents the rate at which failures occur in the

software system over time. A lower failure rate indicates higher reliability.

**• Reliability Testing:** Various testing techniques, such as reliability testing and

failure mode analysis, are used to assess and improve software reliability.

• Achieving high software reliability is crucial for mission-critical systems, as it

ensures that software behaves predictably and consistently, reducing the risk

of disruptions and failures that can lead to significant consequences.

Reliability is often a key consideration in industries like aerospace,

healthcare, and finance.

—---------------------------------------------------------------------------------------------

**Formal Technical Review (FTR), Walkthrough**

Formal Technical Review (FTR) and Walkthrough are two important

techniques used in software engineering to improve the quality of software

products through systematic inspection and review processes. These

processes involve gathering a group of stakeholders to examine software

artifacts, such as code or design documents, to identify defects, ensure

compliance with standards, and promote knowledge sharing.

Formal Technical Reviews (FTRs) and Walkthroughs are valuable software

engineering practices for improving software quality. FTRs are formal,

structured, and detailed reviews suitable for critical phases of the software

development lifecycle, while Walkthroughs are more informal, interactive, and

educational, often used for early-stage reviews and knowledge sharing. The

choice between FTR and Walkthrough depends on the specific goals and

needs of the review process.

**1. Formal Technical Review (FTR):**

• Formal Technical Reviews are structured, well-documented review processes

that follow a defined set of steps and guidelines. FTRs are typically more

rigorous and comprehensive than other review methods, and they are often

used to evaluate various artifacts such as code, design documents, and

requirements specifications. Here are the key steps involved in an FTR:

**• Planning:** In this initial phase, the review team is assembled, and the

objectives, scope, and schedule of the review are defined. The specific

document or artifact to be reviewed is also selected.

**• Preparation:** Reviewers are provided with the relevant documents or code to

be examined in advance. They are expected to study the material and

prepare for the review meeting.

**• Review Meeting:** This is the core of the FTR process. The review team,

including authors and reviewers, gathers to discuss the artifact in detail. The

meeting is typically led by a moderator or facilitator who ensures that the

process follows a predefined agenda. Reviewers raise issues, ask questions,

and suggest improvements. The focus is on identifying defects, ambiguities,

inconsistencies, and violations of coding or design standards.

**• Rework:** After the review meeting, the authors of the artifact address the

identified issues and make necessary corrections or improvements.

**• Follow-up:** A follow-up review may be conducted to ensure that the identified

issues have been resolved satisfactorily. The artifact is re-evaluated to

confirm that it now meets the required quality standards.

**• Documenting Results:** The results of the review, including identified issues,

their severity, and any actions taken, are documented. This documentation

serves as a valuable reference for future development phases.

**• Closure:** Once the artifact has been reviewed, issues addressed, and

reviewers are satisfied with the changes, the review is officially closed.

**• Example of FTR:** Suppose a software development team is working on a

critical financial application. They have completed the design phase and are

preparing to move on to coding. Before coding begins, the team decides to

conduct a Formal Technical Review of the design documents. The steps

involved might look like this:

**• Planning:** The team selects the design documents for review and forms a

review team consisting of experienced developers and system architects.

They schedule a review meeting for one week later.

**• Preparation:** Reviewers are given access to the design documents a few

days before the review meeting. They are expected to thoroughly study the

documents and make notes of any concerns or questions.

**• Review Meeting:** During the meeting, the team goes through the design

documents section by section. Reviewers ask questions, discuss potential

risks, and point out any design flaws or ambiguities they find.

**• Rework:** The design document authors take notes of the issues raised during

the review meeting and make necessary updates to the design.

**• Follow-up:** A follow-up review meeting is scheduled for the next week to

verify that the issues have been addressed correctly.

**• Documenting Results:** The results of both the initial review and the

follow-up review are documented. Any remaining issues or concerns are

noted for further action.

**• Closure:** Once all issues are resolved, the Formal Technical Review of the

design documents is closed. The team can proceed with confidence to the

coding phase, knowing that the design has been thoroughly reviewed and

improved.

**2. Walkthrough:**

• A Walkthrough is a less formal and more collaborative review process than

an FTR. It is often used for educational purposes, knowledge sharing, and

early-stage reviews. During a Walkthrough, the author of the document or

code presents it to a group of reviewers, explaining the content and seeking

their feedback. the key characteristics and steps of a Walkthrough:

**• Presentation:** The author presents the document or code to the reviewers,

explaining its purpose, structure, and key components. The goal is to provide

context and help reviewers understand the material.

**• Reviewers' Questions:** Reviewers ask questions, seek clarifications, and

provide feedback during the presentation. The emphasis is on understanding

and improving the content.

**• No Formal Agenda:** Unlike FTR, Walkthroughs may not follow a strict

agenda or predefined set of steps. The process is more fluid and interactive.

**• Collaboration:** Walkthroughs encourage collaboration and open discussion

among team members. The focus is on sharing knowledge and insights.

**• Example of Walkthrough:** Consider a software development team that is

working on a new web application project. Before starting the coding phase,

the team decides to conduct a Walkthrough of the project's user interface (UI)

design. Here's how it might work:

**• Presentation:** The UI designer presents the design mockups to the team,

explaining the layout, color scheme, navigation, and user interactions.

**• Reviewers' Questions:** Developers, testers, and other team members ask

questions and seek clarifications. They might inquire about the reasoning

behind certain design choices, potential usability issues, or alignment with

user requirements.

**• Feedback and Suggestions:** Reviewers provide feedback on the design,

offering suggestions for improvements. They might suggest changes to

improve user experience or identify design elements that could lead to

usability issues.

**• Discussion and Iteration:** The team engages in discussions to explore

different design alternatives and reach a consensus on improvements. The UI

designer takes notes on the feedback received.

**• Follow-up Actions:** Based on the Walkthrough discussions, the UI designer

makes necessary revisions to the design. The updated design is shared with

the team for further feedback and validation.

**• Iteration and Finalization:** This iterative process continues until the team is

satisfied with the UI design. Once finalized, the design serves as the basis for

the development phase.

**MODULE 6**

**Q) Software Testing**

Testing is the process of evaluating a system or its component(s) with the intent to find whether it satisfies the specified requirements or not. In simple words, testing is executing a system in order to identify any gaps, errors, or missing requirements in contrary to the actual requirements.

According to ANSI/IEEE 1059 standard, Testing can be defined as - A process of analyzing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software

item.

**Q)Testing: Software Quality**

Software Quality Assurance (SQA) is the practice of monitoring all software engineering processes, activities, and methods used in a project to ensure proper quality of the software and conformance against the defined standards. Software Quality Assurance (QA) Testing evaluates the functional, performance, usability and security of the software or app.

Software Quality Assurance (QA) Testing Benefits:

Software Quality Assurance (QA) not only is beneficial to the vendor, but the consumer as well. Credible Software QA Testing can establish expectations for the product while helping a manufacturer produce high quality software. Sacrificing usability, stability and security is not an option if you want your app, software or cloud-based service representing your brand reputation. Intertek’s software quality assurance testing expertise will give you peace of mind that your software or app leaves your users with a positive experience, leaving you more time to focus on enhancements rather than usability, stability or security updates.

Software Quality Assurance (QA) Testing Categories:

Functional Testing - From black-box testing to unit and sanity testing, we partner with you to evaluate your program's key features.

Performance, Network & Load Testing - We'll verify your app or software's performance from load, stress, volume and scalability.

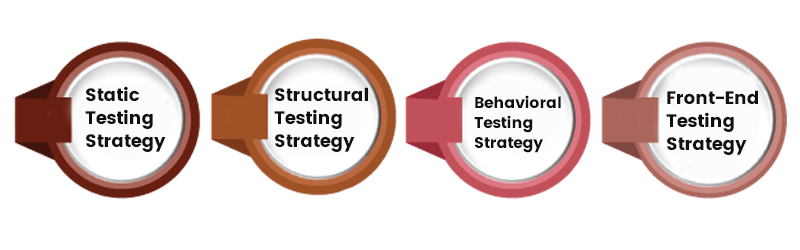
Usability Testing - A comprehensive review of your app or software design, from usability, and user experience (UX), to the user interface (UI).

Cybersecurity Testing - Safeguard your reputation and sensitive customer data from cyber attacks or other malicious activities.

**Q) Testing: Strategic Approach**

Strategic planning, which outlines the goals, objectives, and methodology, serves as the foundation for effective software testing. It entails determining the scope of testing, establishing appropriate testing procedures, and outlining the resources and timescales required. A solid plan ensures that testing efforts are aligned with the project's objectives and that any risks and problems are mitigated.

Testing Strategies and its types



Static Testing Strategy

A static test assesses the quality of a system without executing the system, which may initially appear challenging. However, there are several strategic approach to software testing available to achieve this evaluation.

* Static testing involves analyzing specific portions or elements of a system to identify potential issues at an early stage. One example is desk-checking, where developers review their code before pushing it, serving as a form of static testing. Another example is conducting review meetings to assess requirements, design, and code.
* Static tests provide significant advantages by detecting problems in requirements before they manifest as system bugs, resulting in time and cost savings. Additionally, conducting a preliminary code review helps identify bugs without the need to build, install, and run the entire system.
* Timing is a vital aspect of static tests. Reviewing requirements after developers have completed coding the software can assist testers in designing effective test cases. However, it is important to note that testers can only identify bugs in existing code by executing the system, thus undermining the purpose of static tests. In such cases, individual developers should review the code promptly after creation and before integration.
* Organizations can enhance software quality, mitigate risks, and optimize the overall development process by implementing comprehensive static testing practices at appropriate stages.

Structural Testing Strategy

Although static tests provide valuable insights, they need to be more comprehensive. Executing the software on real devices and running the entire system to ensure comprehensive bug detection is essential. Unit testing, including structural tests, is employed to achieve this objective.

* White-box testing, also known as clear-box testing or structural testing in software testing, is conducted by testers with extensive knowledge of the devices and systems under evaluation.
* This testing approach is frequently employed to scrutinize specific components and interfaces, aiming to pinpoint and rectify localized errors within data flows. To gain practical expertise in Software Testing, consider enrolling in a Software Testing Online Course. These courses offer comprehensive training, and enable learners to master in Software Testing.

Behavioral Testing Strategy

Behavioral Testing, or black-box testing, centers around the system's behavior rather than its underlying mechanisms. It emphasizes workflows, configurations, performance, and all aspects of the user journey. The main objective of these tests is to assess a website or app from an end-user perspective.

* Behavioral Testing should encompass multiple user profiles and diverse usage scenarios.
* It primarily focuses on evaluating fully integrated systems rather than individual components. This approach allows for a comprehensive assessment of system behavior from the user's perspective, which becomes evident only when the system is sufficiently assembled and integrated.
* While behavioral tests are predominantly executed manually, certain aspects can be automated.
* Manual testing requires meticulous planning, thoughtful design, and thorough result verification to identify potential issues.
* Automation testing streamlines repetitive actions, such as regression tests, ensuring that new code implementations do not disrupt existing functioning features. For instance, when testing a website, automating the process of filling in multiple form fields with various values can significantly save time, effort and mitigate human error.

Front-End Testing Strategy

The front end of an application pertains to its user-facing component, serving as the primary interface for content consumption and business transactions. Front-end is one of the software testing strategies that plays a vital role in the software development life cycle (SDLC) by ensuring the proper functioning of GUI elements. We can break them into the following categories of software testing types and approaches:

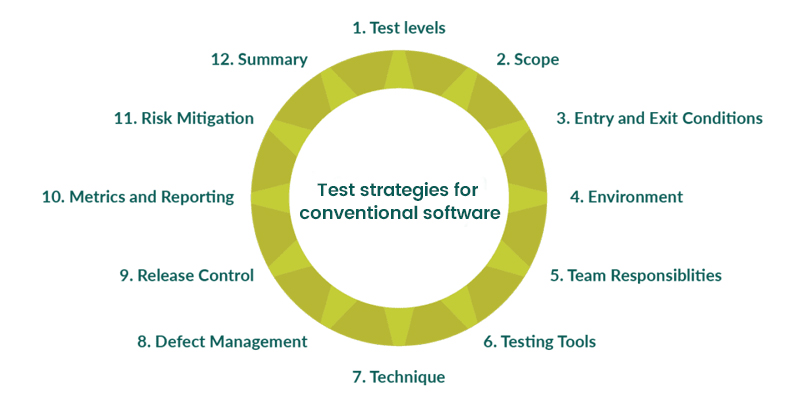
* Unit Testing: Unit Testing is the initial layer integrated into the codebase, focusing on running test functions before the application is built to assess individual processes and modules.
* Functional Testing: Functional testing involves manually evaluating features and components through a user interface, typically using mock data.
* Integration Testing: Integration testing involves connecting front-end components with back-end services to ensure seamless integration from client inputs, through APIs, to database management within a dedicated test environment.
* System/UI Testing: System/UI testing occurs once a test version of the user interface has been successfully integrated with the back-end service, enabling automated testing and analysis conducted by developers in collaboration with the QA team.
* Performance/Load Testing: Performance/load testing verifies the application's ability to handle the expected load and scale, considering optimal usage scenarios.
* Stress/Security Testing: Stress/Security Testing validates the application's response under unexpected loads and threats beyond the expected limit.
* Accessibility Testing: Accessibility testing ensures that individuals with disabilities can effectively utilize the application.

**Q) Strategic Issues**

Following are the issues considered to implement software testing strategies.

* Specify product requirements in a quantifiable manner long before testing commences. Although the overriding objective of testing is to find errors, a good testing strategy also assesses other quality characteristics such as portability, maintainability, and usability . These should be specified in a way that is measurable so that testing results are unambiguous.
* State testing objectives explicitly. The specific objectives of testing should be stated in measurable terms. For example, test effectiveness, test coverage, mean time to failure, the cost to find and fix defects, remaining defect density or frequency of occurrence, and test work-hours per regression test all should be stated within the test plan .
* Understand the users of the software and develop a profile for each user category. Use-cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of the product.
* Develop a testing plan that emphasizes “rapid cycle testing.” Gilb recommends that a software engineering team “learn to test in rapid cycles (2 percent of project effort) of customer-useful, at least field ‘trialable,’ increments of functionality and/or quality improvement.” The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.
* Build “robust” software that is designed to test itself. Software should be designed in a manner that uses antibugging techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.
* Use effective formal technical reviews as a filter prior to testing. Formal technical reviews can be as effective as testing in uncovering errors. For this reason, reviews can reduce the amount of testing effort that is required to produce high-quality software.
* Conduct formal technical reviews to assess the test strategy and test cases themselves. Formal technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.
* Develop a continuous improvement approach for the testing process. The test strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for software testing.

**Testing: Strategies for Conventional Software,**

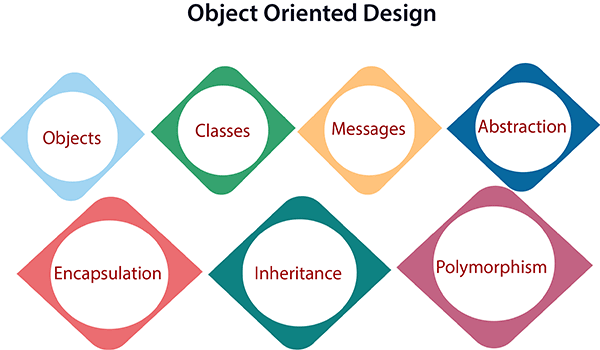


Test strategies for conventional software are systematic approaches employed to ensure effective testing of traditional software applications. These strategies outline the overall approach, goals, and techniques to be utilized during the testing process. Some commonly used test strategies for conventional software include:

* Requirement-based Testing: Requirement-based Testing is a strategy that focuses on verifying whether the software meets the specified requirements. Test cases are designed based on the documented requirements, and the software is evaluated against these requirements to ensure compliance.
* Functional Testing: Functional Testing strategy aims to validate the functional aspects of the software by testing individual functions or features. It involves designing test cases to exercise different functionalities and verifying that the software performs as intended.
* Integration Testing: Integration testing involves testing the interaction and integration between various software components or modules. It ensures that different components work together seamlessly and accurately to exchange data.
* System Testing: System testing is conducted to validate the entire software system. It includes testing all integrated components and subsystems to ensure they function correctly and meet the overall system requirements.
* Performance Testing: This strategy evaluates the software's performance under specific conditions, such as load, stress, or scalability. It aims to identify performance bottlenecks, measure response times, and ensure the software can handle expected workloads.
* Regression Testing: This testing is one of the software testing strategies that focuses on retesting previously tested functionalities to ensure that recent changes or fixes have not introduced new defects or caused any unintended side effects.
* User Acceptance Testing (UAT): UAT is performed by end-users or representatives to verify whether the software meets their expectations and requirements. It ensures the software is ready for deployment and use in a real-world environment.
* Security Testing: The security Testing strategy aims to identify vulnerabilities and ensure the software's resistance against potential security threats. It includes testing for authentication, authorization, data encryption, and protection against common security attacks.
* Usability Testing: Usability testing assesses the software's user-friendliness, ease of use, and overall user experience. It evaluates interface design, navigation, and user interaction to ensure optimal usability.
* Maintenance Testing: Maintenance testing is performed after software updates or modifications to ensure that the changes have not introduced new defects or adversely affected existing functionalities. To see how Software Testing might be used, join the Software Testing Course in Coimbatore and learn the various applications and their functionalities.

**Q) Object oriented software,**

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



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

**Q) Web AppsValidating Testing- System**

TEST STRATEGIES FOR WEBAPPS:

The strategy for WebApp testing adopts the basic principles for all software testing

and applies a strategy and tactics that are used for object-oriented systems. The

following steps summarize the approach:

1. The content model for the WebApp is reviewed to uncover errors.

2. The interface model is reviewed to ensure that all use cases can be

accommodated.

3. The design model for the WebApp is reviewed to uncover navigation errors.

4. The user interface is tested to uncover errors in presentation and/or navigation

mechanics.

5. Each functional component is unit tested.

6. Navigation throughout the architecture is tested.

7. The WebApp is implemented in a variety of different environmental

configurations and is tested for compatibility with each configuration.

8. Security tests are conducted in an attempt to exploit vulnerabilities in the

WebApp or within its environment.

9. Performance tests are conducted.

10. The WebApp is tested by a controlled and monitored population of end

users.The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp

reliability and performance.

After the software has been integrated (constructed), a set of

high-order tests is conducted. Validation criteria (established

during requirements analysis) must be evaluated. Validation

testing provides final assurance that software meets all

informational, functional, behavioral, and performance

requirements.

Most software product builders use a process called alpha and beta

testing to uncover errors that only the

end user seems able to find.

The alpha test is conducted at the developer’s site by a

representative group of end users. The software is used in a natural

setting with the developer “looking over the

shoulder” of the users and recording errors and usage problems.

Alpha tests are conducted in a controlled environment.

The beta test is conducted at one or more end-user sites. Unlike

alpha testing, the developer generally is not present. Therefore, the

beta test is a “live” application of the software in an environment

that cannot be controlled by the developer. The customer records

all problems (real or imagined) that are encountered during beta

testing and reports these to the developer at regular intervals. As a

result of problems reported during beta tests, you make

modifications and then prepare for release of the software product

to the entire customer base.

**Q) Testing- Art of Debugging.**

Debugging occurs as a consequence of successful

testing. That is, when a test

case uncovers an error, debugging is the process that

results in the removal of the error.

The Debugging Process:

Debugging is not testing but often occurs as a

consequence of testing. Referring to Fig, the debugging

process begins with the execution of a test case.The

debugging process will usually have one of two

outcomes:

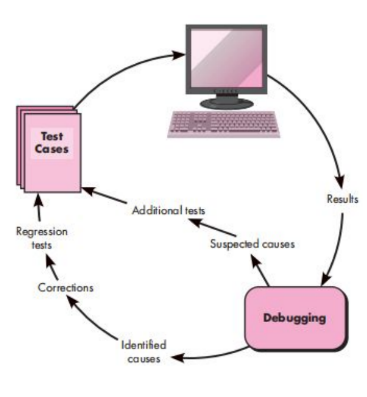
(1) the cause will be found and corrected or

(2) the cause will not be found. In the latter case, the

person performing debugging may suspect a cause,

design a test case to help validate that suspicion, and

work toward error correction in an iterative fashion.



**Q) Maintenance : Software Maintenance-**

SOFTWARE MAINTENANCE

Much of the software we depend on today is on average 10 to 15 years old. Even

when these programs were created using the best design and coding techniques

known at the time [and most were not], they were created when program size and storage space were principle concerns. They were then migrated to new platforms, adjusted for changes in machine and operating system technology and enhanced to meet new user needs—all without enough regard to overall architecture. The result is the poorly designed structures, poor coding, poor logic, and poor documentation of the software systems we are now called on to keep running. Another reason for the software maintenance problem is the mobility of software people

In order to effectively support industry-grade software, your organization (or its designee) must be capable of making the corrections, adaptations, and enhancements that are part of the maintenance activity. But in addition,

the organization must provide other important support activities that include ongoing operational support, end-user support, and reengineering activities over the complete life cycle of the software.

**Q) Software Supportability-**

SOFTWARE SUPPORTABILITY

In order to effectively support industry-grade software, your organization (or its designee) must be capable of making the corrections, adaptations, and enhancements that are part of the maintenance activity. But in addition, the organization must provide other important support activities that include ongoing operational support, end-user support, and reengineering activities over the complete life cycle of the software.

A reasonable definition of software supportability is the capability of supporting a software system over its whole product life. This implies satisfying any necessary needs or requirements, but also the provision of equipment, support infrastructure, additional software, facilities, manpower, or any other resource

required to maintain the software operational and capable of satisfying its function

**Q) Reengineering- Business Process Reengineering**

Business Process Reengineering

◻ Business Process Reengineering (BPR) is a fundamental rethinking and radical redesign of business processes to achieve significant improvements in critical performance metrics such as cost, quality, service, and speed. It involves the analysis, redesign, and implementation of business processes, often leveraging modern technology to streamline and optimize operations. BPR aims to break down silos, eliminate inefficiencies, and align processes with organizational goals.

◻ Key Concepts in Business Process Reengineering:

◻ Process Analysis: The first step in BPR is to thoroughly analyze existing business

processes. This involves documenting and understanding each step, identifying

bottlenecks, redundancies, and areas of inefficiency.

◻ **Redesign**: After identifying areas for improvement, the next step is to redesign the

processes from the ground up. This often involves challenging existing

assumptions and practices and rethinking how tasks are performed.

◻ **Technology Integration**: BPR often leverages technology to automate and

optimize processes. This may involve the use of enterprise software, workflow

automation tools, and data analytics to enhance efficiency.

◻ **Change Management:** Implementing BPR typically results in significant changes

for employees and the organization as a whole. Effective change management is

crucial to ensure that employees adapt to the new processes and technologies.

◻ **Performance Metrics:** BPR should focus on improving key performance metrics,

such as reducing process cycle times, lowering costs, improving product or service

quality, and increasing customer satisfaction.

Example: Business Process Reengineering in a Retail Company

◻ Let's consider a retail company with a chain of physical stores and an online

presence. Over time, the company's order fulfillment process has become slow and

inefficient, leading to delayed deliveries and increased customer complaints.

◻ Process Analysis:

◻ Analyze the existing order fulfillment process, from the point of order placement to

delivery.

◻ Identify bottlenecks in the process, such as manual order entry, multiple handoffs

between departments, and lack of real-time order tracking.

◻ **Redesign**:

◻ Redesign the process to be more customer-centric and efficient. For example,

implement an omni-channel order management system that seamlessly integrates

in-store and online orders.

◻ Streamline the order processing by automating routine tasks and reducing manual

interventions.

◻ **Technology Integration:**

◻ Implement a new order management system that provides real-time visibility into

inventory across all stores and the online warehouse.

◻ Use automated inventory management to prevent over-ordering and out-of-stock

situations.

◻ Enable customers to track their orders in real-time through a mobile app or website.

**Change Management:**

◻ Communicate the changes to employees and provide training on using the new systems and processes.

◻ Create a culture of continuous improvement and encourage employees to provide feedback for further refinements.

◻ **Performance Metrics:**

◻ Measure key performance indicators, such as order fulfillment cycle time, order accuracy, and customer satisfaction.

◻ Monitor and analyze data to ensure that the redesigned process is meeting its goals.

Through BPR, the retail company has significantly improved its order fulfillment

process. Orders are processed more efficiently, leading to faster deliveries and

fewer errors. Customer satisfaction has increased, and the company has reduced

operational costs by optimizing inventory management and reducing the need for

manual interventions. This example illustrates how BPR can lead to substantial

improvements in business processes and, ultimately, better business outcomes.

**Q) Software Reengineering- Reverse Engineering-**

Mod 6 - Pg 48-52

**Q) Restructuring- Forward Engineering**

Mod 6 - Pg 59-65

Ye dono boht bade hai. Samajh nai raha kya dalu. Ek baar ppt dekh lo