Internal Assessment Test 2 – August 2025

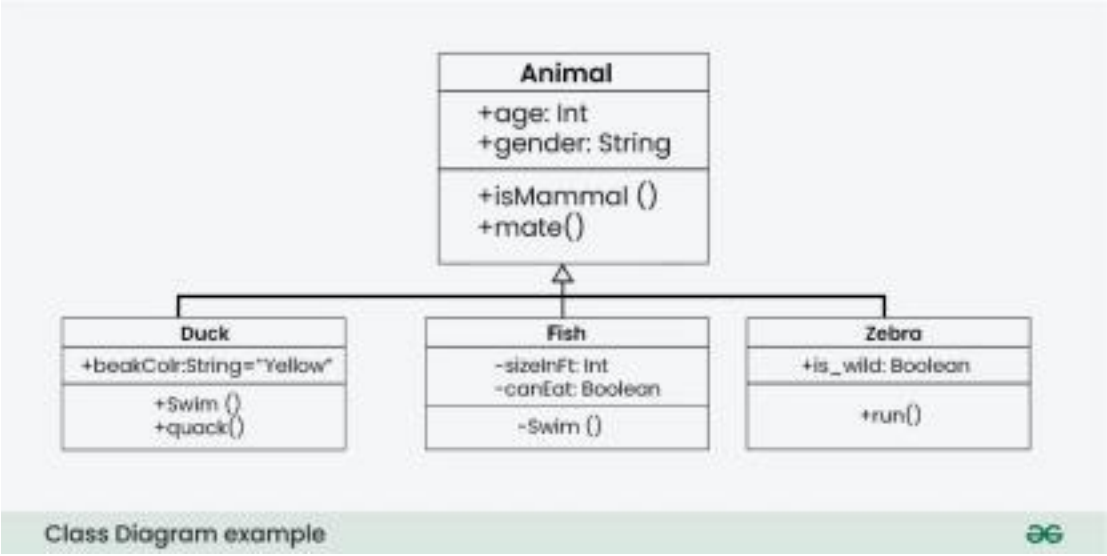
Software Engineering							Sub Code:	MMC204
06.08.25	Duration :	90 min's	Max Marks:	50	Sem:	I	Branch:	MCA

PART I

1. Define Class and Interaction Diagrams with an example.

Class Diagram

The most widely use UML diagram is the class diagram. It is the building block of all object oriented software systems. We use class diagrams to depict the static structure of a system by showing system's classes, their methods and attributes. Class diagrams also help us identify relationship between different classes or objects.

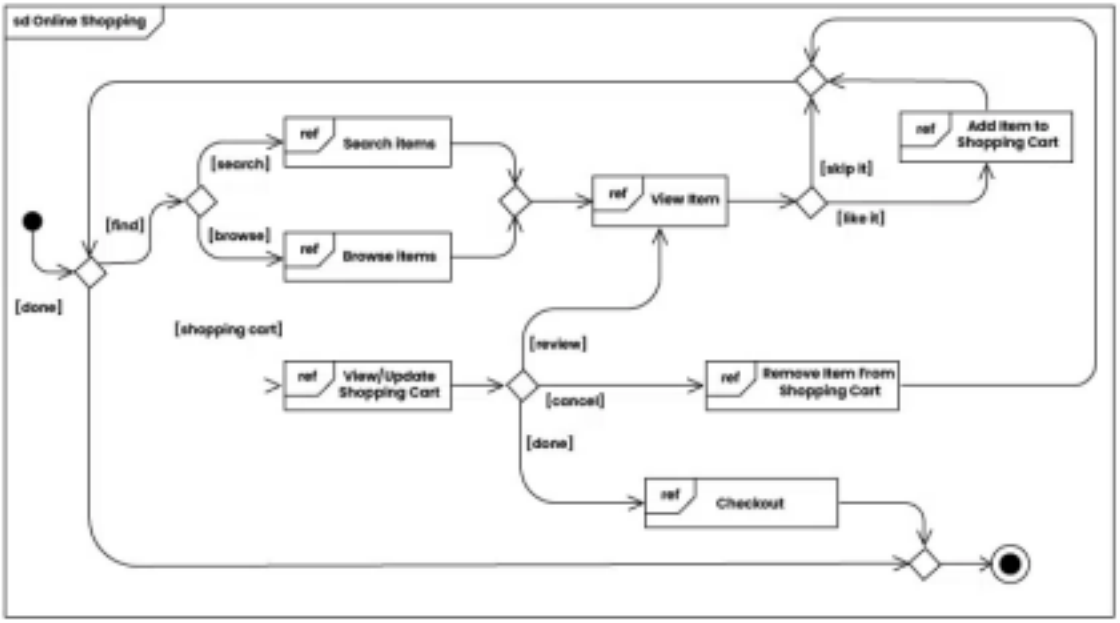


Class Diagram example

Interaction Diagram

An Interaction Diagram is a type of UML (Unified Modeling Language) diagram that illustrates the flow of interactions between various elements in a system or process. It provides a high-level overview of how interactions occur, including the sequence of actions, decisions, and interactions between different components or objects.

Example of Interaction overview Diagram



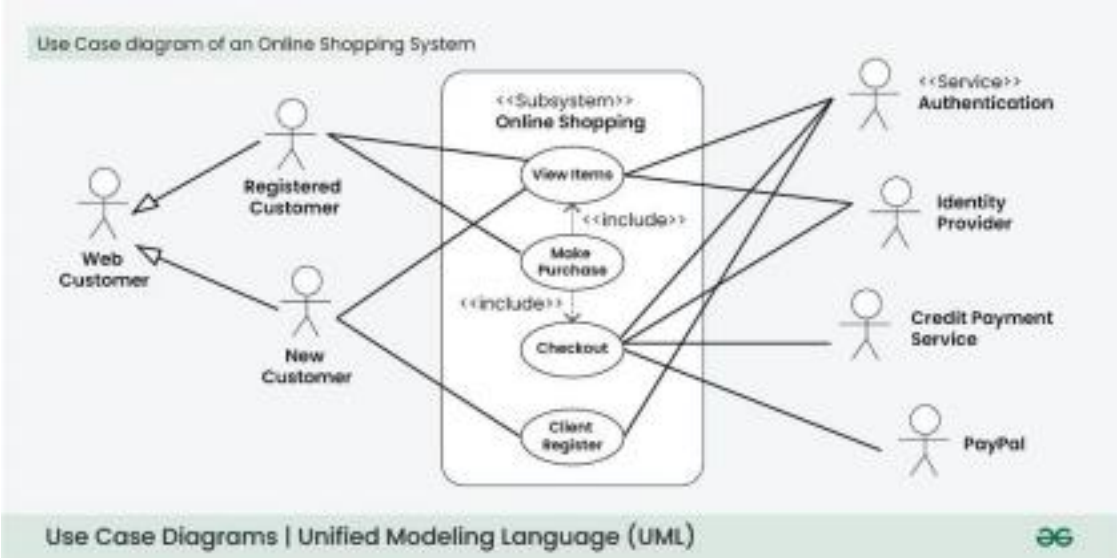
2. Describe Use Case Model with neat diagram.

Use Case Diagrams

Use Case Diagrams are used to depict the functionality of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors).

· A use case is basically a diagram representing different scenarios where the system can be used.

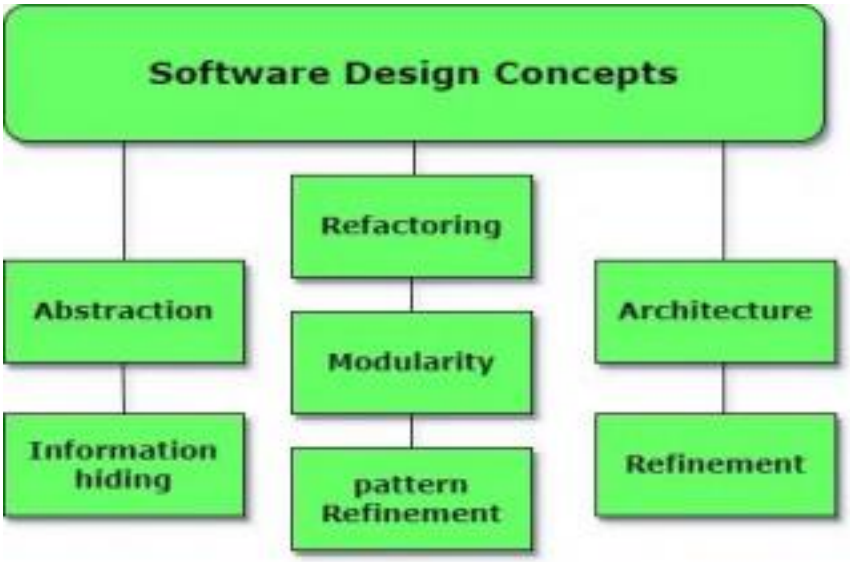
· A use case diagram gives us a high level view of what the system or a part of the system does without going into implementation details. '



PART II

3. Explain Design Concepts in detail.

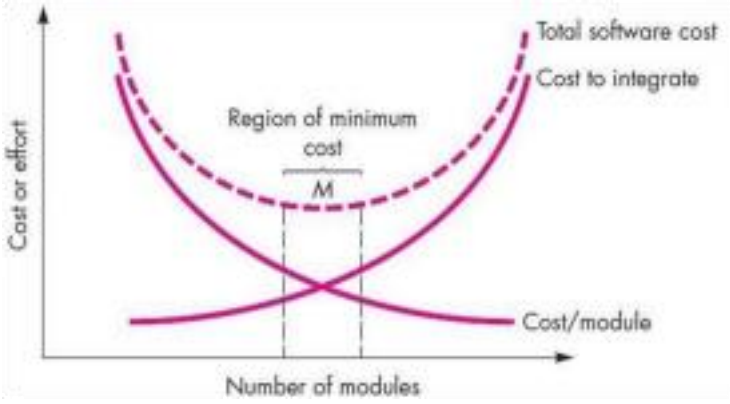
Software engineering design concepts are fundamental ideas that guide the creation of robust, maintainable, and scalable software systems. Key concepts include modularity, abstraction, encapsulation, and functional independence, which promote the development of well-structured and understandable software. These concepts are often applied in conjunction with design patterns and architectural styles to address common software development challenges.



1. Modularity:

· Software is broken down into smaller, independent modules or components.

· Each module has a specific purpose and can be developed, tested, and maintained separately. · This approach improves code organization, reusability, and reduces complexity.



2. Abstraction:

Hides complex implementation details and exposes only the necessary information to other parts of the system.

Allows developers to work with high-level concepts without getting bogged down in low-level details. Examples include abstract classes, interfaces, and APIs.

3. Encapsulation:

Bundles data and the methods that operate on that data within a single unit (like a class).

Protects the internal state of an object from unauthorized access and modification.

Reduces dependencies between different parts of the system.

4. Information Hiding:

Prevents other modules from accessing internal details of a module, promoting loose coupling.

Reduces the impact of changes in one module on other modules.

5. Functional Independence:

Modules should be designed to perform specific functions with minimal reliance on other modules. · Reduces complexity and improves testability.

6. Design Patterns:

Reusable solutions to common problems in software design.

Provide templates for solving recurring design issues.

Examples include Singleton, Factory, and Observer patterns.

7. Software Architecture:

Defines the overall structure and organization of the software system.

Includes components, interfaces, and relationships between different parts of the system.

Influences the scalability, maintainability, and performance of the software.

8 . Refinement

Stepwise refinement is a **top-down** design strategy originally proposed by Niklaus Wirth. Refinement is actually a process of ***elaboration***. You begin with a statement of function that is defined at a high level of abstraction.

Abstraction and **refinement** are complementary concepts. Abstraction enables you to specify procedure and data internally but suppress the need for "outsiders" to have knowledge of low level details. Refinement helps you to reveal low-level details as design progresses.

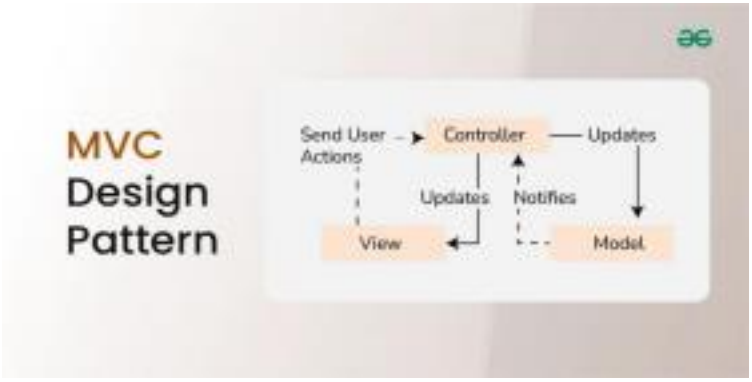
9.Refactoring

An important design activity suggested for many agile methods, ***refactoring*** is a reorganization technique that simplifies the design (or code) of a component without changing its function or behavior.

4. Explain Model View Controller in detail.

MVC Design Pattern

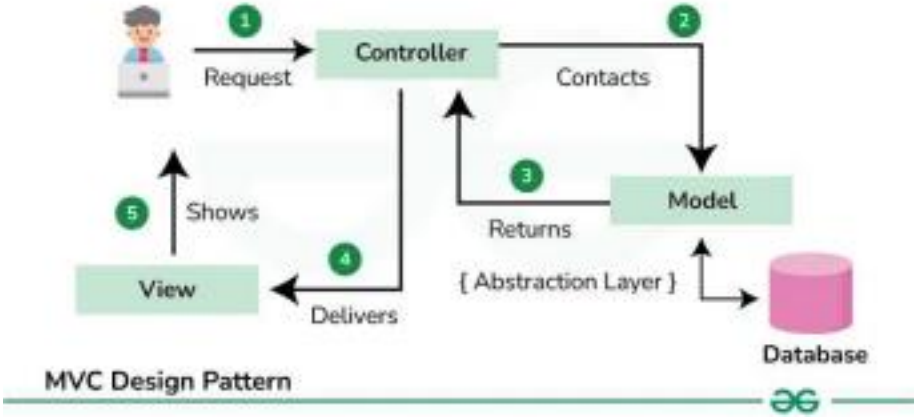
The MVC design pattern is a software architecture pattern that separates an application into three main components: Model, View, and Controller, making it easier to manage and maintain the codebase. It also allows for the reusability of components and promotes a more modular approach to software development.



Why use MVC Design Pattern?

The MVC (Model-View-Controller) design pattern breaks an application into three parts: the Model (which handles data), the View (which is what users see), and the Controller (which connects the two). This makes it easier to work on each part separately, so you can update or fix things without messing up the whole app. It helps developers add new features smoothly, makes testing simpler, and allows for better user interfaces. Overall, MVC helps keep everything organized and improves the quality of the software.

Components of the MVC Design Pattern



1. Model

The Model component in the MVC (Model-View-Controller) design pattern demonstrates the data and business logic of an application. It is responsible for managing the application's data, processing business rules, and responding to requests for information from other components, such as the View and the Controller.

2. View

Displays the data from the Model to the user and sends user inputs to the Controller. It is passive and does not directly interact with the Model. Instead, it receives data from the Model and sends user inputs to the Controller for processing.

3. Controller

Controller acts as an intermediary between the Model and the View. It handles user input and updates the Model accordingly and updates the View to reflect changes in the Model. It contains application logic, such as input validation and data transformation.

Communication between the Components

This below communication flow ensures that each component is responsible for a specific aspect of the application's functionality, leading to a more maintainable and scalable architecture

User Interaction with View: The user interacts with the View, such as clicking a button or entering text into a form.

View Receives User Input: The View receives the user input and forwards it to the Controller. · **Controller Processes User Input:** The Controller receives the user input from the View. It interprets the input, performs any necessary operations (such as updating the Model), and decides how to respond.

Controller Updates Model: The Controller updates the Model based on the user input or application logic.

Model Notifies View of Changes: If the Model changes, it notifies the View. · **View Requests Data from Model:** The View requests data from the Model to update its display. · **Controller Updates View:** The Controller updates the View based on the changes in the Model or in response to user input.

View Renders Updated UI: The View renders the updated UI based on the changes made by the Controller.

PART III

5. Explain Architectural Styles in detail.

Architectural Styles

1. Layered Architecture

Description: System is organized into layers, each with a specific responsibility. • At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing(communication and coordination with OS)

Example of this architectural style is OSI-ISO (Open Systems Interconnection-International Organization for Standardization) communication system.

Client-Server Architecture

The client-server pattern has two major entities. They are a server and multiple clients. Here the server has resources(data, files or services) and a client requests the server for a particular resource. Then the server processes the request and responds back accordingly.

Advantages:

1. **Centralized Management:** Servers can centrally manage resources, data, and security policies, thus simplifying the maintenance.
2. **Scalability:** Servers can be scaled up to handle increased client requests.
3. **Security:** Security measures such as access controls, data encryption can be implemented in a better way due to centralized controls.

Disadvantages:

1. **Single Point of Failure:** Due to centralized server, if server fails clients lose access to services, leading loss of productivity.
2. **Costly:** Setting up and maintaining servers can be expensive due to hardware, software, and administrative costs.
3. **Complexity:** Designing and managing a client-server architecture can be complex. **Use Cases:**

1. Web Applications like Amazon.
2. Email Services like Gmail, Outlook.
3. File Sharing Services like Dropbox, Google Drive.
4. Media Streaming Services like Netflix.
5. Education Platforms like Moodle.

Pipe-Filter Architecture Pattern

Pipe-Filter Architecture Pattern structures a system around a series of processing elements called filters that are connected by pipes. Each filter processes data and passes it to the next filter via pipe.

Advantages:

1. **Reusability:** Filters can be reused in different pipelines or applications.
2. **Scalability:** Additional filters can be added to extend the functionality to the pipeline. 3. **Parallelism:** Filters can be executed in parallel if they are stateless, thus improving performance.

Disadvantages:

1. **Debugging Difficulty:** Identifying and debugging issues are difficult in long pipelines.

2. **Data Format constraints:** Filters must agree on the data format, requiring careful design and standardization.
3. **Latency:** Data must be passed through multiple filters, which can introduce latency.

Use Cases:

1. Data Processing Pipelines like Extract, Transform, Load (ETL) processes in data warehousing. 2. Compilers.
3. Stream-Processing like Apache Flink.
4. Image and Signal Processing.

Tiered Architecture

Tiered architecture (also called n-tier architecture) is a design pattern where the software is divided into logical tiers, each responsible for a specific function. These tiers often run on separate systems or servers.

Common Tiers in N-Tier Architecture

1. Presentation Tier (Client Tier)

Role: User Interface (UI)

Responsibility: Displays data, takes user input.

Example: Web page, mobile app, desktop GUI.

2. Application Tier (Business Logic Tier / Middle Tier)

Role: Core functionality

Responsibility: Processes data, makes decisions, applies rules.

Example: Java/.NET services, business logic modules.

3. Data Tier (Database Tier)

Role: Data management

Responsibility: Stores, retrieves, and manages data.

Example: MySQL, Oracle, MongoDB.

6. Briefly discuss about Coupling and cohesion.

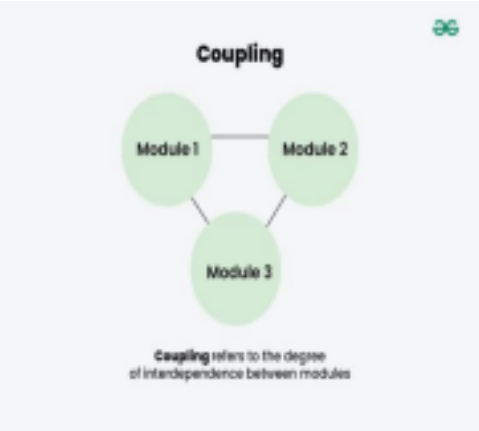
Coupling and Cohesion

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

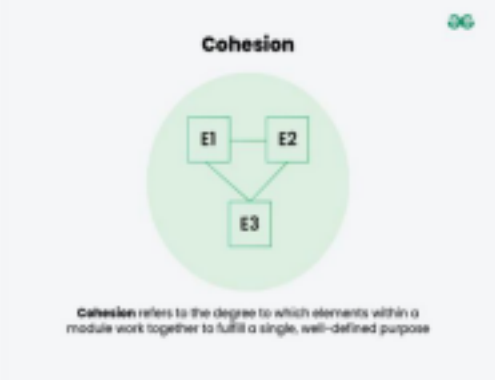
Coupling and Cohesion are two key concepts in software engineering that are used to measure the quality of a software system's design.

What is Coupling and Cohesion?

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

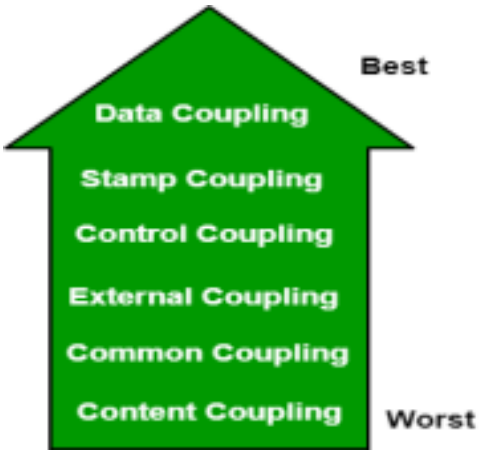


Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose. High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes.



Types of Coupling

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



Following are the types of Coupling:

· **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent of each other and communicate through data. Module communications don't contain tramp data. Example-customer billing system.

· **Stamp Coupling** In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors this choice was made by the insightful designer, not a lazy programmer.

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

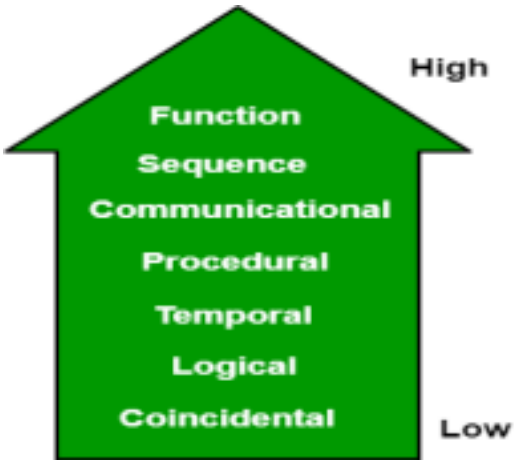
· **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.

· **Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses, and reduced maintainability.

· **Content Coupling:** In a content coupling, one module can modify the data of another module, or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

Types of Cohesion

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



Following are the Types of Cohesion:

· **Functional Cohesion:** Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation. · **Sequential Cohesion:** An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages. · **Communicational Cohesion:** Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer. · **Procedural Cohesion:** Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.

· **Temporal Cohesion:** The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time.

· **Logical Cohesion:** The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.

· **Coincidental Cohesion:** The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion. Ex print next line and reverse the characters of a string in a single component.

Differentiate between Coupling and Cohesion

Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding
Coupling shows the relationships between modules.	Cohesion shows the relationship within the module.
Coupling shows the relative independence between the modules.	Cohesion shows the module's relative functional strength

While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

PART IV

7. Explain Software Project Management.

In software engineering, project management is the process of planning, organizing, and overseeing the development of a software product from idea to delivery. It involves coordinating people, processes, and tools to ensure that software is developed on time, within budget, and according to user requirements and quality standards. Software project management focuses specifically on:

· Managing software development life cycle (SDLC) phases

· Handling technical complexity

· Adapting to changing requirements

· Ensuring code quality and user satisfaction

Key points explain project management:

· Initiation: This phase involves defining the project, identifying its objectives, and determining the feasibility of the project. It includes creating a project charter, conducting a feasibility study, and obtaining approval to proceed with the project.

· Planning: In this phase, detailed planning is done to outline the scope of the project, define tasks and activities, create schedules, allocate resources, set budgets, and establish quality standards. The project plan serves as a roadmap for the project execution.

·Execution: This phase involves implementing the project plan by coordinating resources, managing tasks, and communicating progress. Project managers oversee the execution of tasks, monitor performance, resolve issues, and ensure that the project stays on track.

· Monitoring and Controlling: Throughout the project lifecycle, monitoring and controlling are essential to track progress, assess performance against the project plan, identify deviations or risks, and take corrective actions as needed. This phase ensures that the project meets its objectives within the defined scope, time, and budget constraints.

· Closing: The closing phase involves completing all project activities, delivering the final product or service to the customer, obtaining acceptance and feedback, documenting lessons learned, and closing out contracts and resources. It marks the formal end of the project.

8. Describe DevOps in detail.

DevOps is a cross-disciplinary practice in which application development (dev) teams collaborate with IT operations (ops) to improve product quality and accelerate time to market.

In a DevOps framework, developers and IT managers collaborate with experts in quality control, security, and support throughout every project stage. This cooperative effort aims to deliver code rapidly in a seamless loop of continuous integration and continuous delivery (CI/CD).DevOps builds on the frequent releases and CI/CD aspects of agile development methods but adds infrastructure management to make product delivery more flexible and dynamic.

When properly implemented, DevOps replaces silos, rigid job descriptions, and bottlenecks with a responsive, cross-disciplinary model that enables innovation and continuous improvement.

Core Principles of DevOps

DevOps principles focus on unifying development and operations through seamless collaboration, shared responsibility, and continuous improvement. These principles emphasize automation, continuous integration and delivery, treating infrastructure as code, and establishing comprehensive monitoring systems.

DevOps embraces rapid iteration with frequent, small-scale changes that minimize risk while accelerating delivery. Security is integrated early and throughout the process rather than as a final checkpoint.

Organizations that adopt DevOps create environments where people learn from mistakes, experiment with new methods, and make data-driven decisions. This approach helps businesses respond better to market needs while maintaining robust, secure, and scalable systems.

PART IV

9. Explain Project Scheduling in detail.

Project schedule simply means a mechanism that is used to communicate and know about that tasks are needed and has to be done or performed and which organizational resources will be given or allocated to these tasks and in what time duration or time frame work is needed to be performed. Effective project scheduling leads to success of project, reduced cost, and increased customer satisfaction. Scheduling in project management means to list out activities, deliverables, and milestones within a project that are delivered. It contains more notes than your average weekly planner notes. The most common and important form of project schedule is Gantt chart.

Project Scheduling Process

· Process :The manager needs to estimate time and resources of project while scheduling project. All activities in project must be arranged in a coherent sequence that means activities should be arranged in a logical and well-organized manner for easy to understand. Initial estimates of project can be made optimistically which means estimates can be made when all favorable things will happen and no threats or problems take place. The total work is separated or divided into various small activities or tasks during project schedule. Then, Project manager will decide time required for each activity or task to get completed. Even some activities are conducted and performed in parallel for efficient performance. The project manager should be aware of fact that each stage of project is not problem-free.

Problems arise during Project Development Stage :

· People may leave or remain absent during particular stage of development. · Hardware may get failed while performing.

· Software resource that is required may not be available at present, etc. The project schedule is represented as set of chart in which work-breakdown structure and dependencies within various activities are represented. To accomplish and complete project within a given schedule, required resources must be available when they are needed. Therefore, resource estimation should be done before starting development.

Resources required for Development of Project :

· Human effort

· Sufficient disk space on server

· Specialized hardware

· Software technology

· Travel allowance required by project staff, etc.

Advantages of Project Scheduling :

There are several advantages provided by project schedule in our project management:

· It simply ensures that everyone remains on same page as far as tasks get completed, dependencies, and deadlines.

· It helps in identifying issues early and concerns such as lack or unavailability of resources.

· It also helps to identify relationships and to monitor process.

· It provides effective budget management and risk mitigation.

10. Explain Software Configuration Management.

Configuration Management (SCM) is a systematic approach to managing and controlling changes to software artifacts throughout the software development lifecycle. It ensures the integrity, traceability, and consistency of software systems by tracking and controlling modifications to source code, documentation, requirements, and other related components.

Key aspects and activities of SCM include:

· Configuration Identification:

Defining and identifying the individual components or "configuration items" (CIs) of a software system, such as source code files, libraries, build scripts, and documentation.

· Configuration Control:

Managing and controlling changes to identified CIs through formal procedures, including change requests, approvals, and versioning. This prevents unauthorized or uncontrolled modifications.

· Configuration Status Accounting:

Recording and reporting the status of configuration items and changes, providing a clear

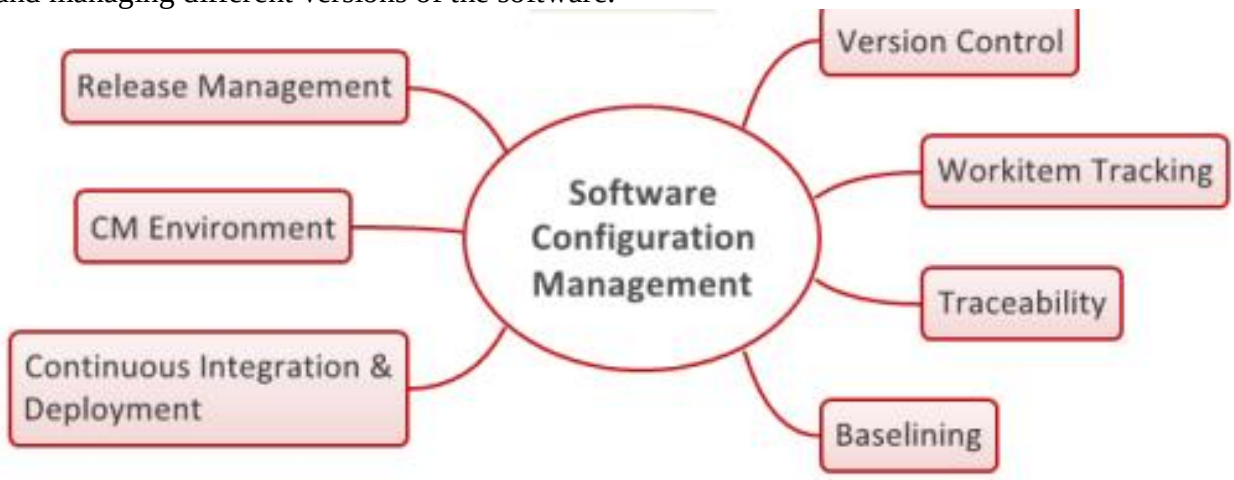
history of modifications, who made them, and when.

Configuration Auditing:

Verifying that the current state of the software system matches its documented configuration and ensuring compliance with established standards and procedures.

· Release Management:

Planning, controlling, and delivering software releases, including packaging, deployment, and managing different versions of the software.



SCM is crucial for:

· maintaining Software Integrity: Ensuring that the software remains consistent and functional despite ongoing changes.

Enabling Collaboration:
Facilitating teamwork by providing a structured way for multiple developers to work on the same codebase without conflicts.

· Improving Traceability:

Allowing for the tracking of changes and their impact, which is essential for debugging, problem-solving, and compliance.

· Supporting Version Control:

Managing different versions of software, enabling rollbacks to previous states if necessary.

· Enhancing Software Quality:

Contributing to higher-quality software by reducing errors and ensuring that changes are properly implemented and tested.