Unit 3

SOFTWARE DESIGN

1.. Software Design Fundamentals

Software Design Definition:

Software design is the process of investing and selecting programs that meet the objective for a software system.

Software design is the process of envisioning and defining software solutions to one or more sets of problems. One of the main components of software design is software requirements analysis (SRA). SRA is a part of the software development process that lists specifications used in software engineering.

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. In order to be easily implementable in a conventional programming language, the following item must be designed during the design phase:

•Different modules required to implement the design solution.

•Control relationship among the identified modules.

•Interferes among different modules

•Algorithms required to implement the individual modules.

Thus the goal of design phase is to take the SRS document of the input and to produce the above mentioned items at the completion stage of the design phase.

Software Design Models: Software designs and problems are often complex and many aspects of software system must be modeled.

a)Static Model: A static model describes the static structure of the system being modeled, which is considered less likely to change than the functions of the system. In particular, a static model defines the classes in the system, the attributes of the classes, the relationships between classes, and the operations of each class.

b)Dynamic Model: Dynamic Modeling is used to represent the behavior of the static constituents of a software , here static constituents includes, classes , objects, their relationships and interfaces Dynamic Modeling also used to represents the interaction, workflow, and different states of the static constituents in a software.

2. Software Design Process

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

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 focussed 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 on 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.

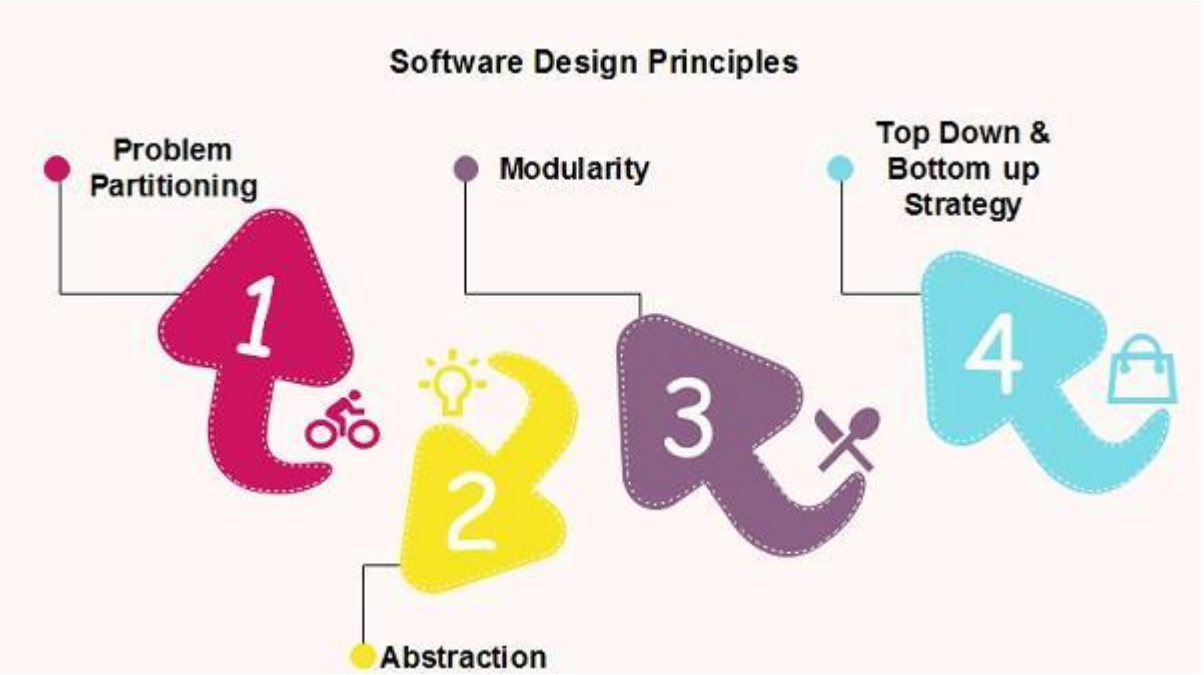
Detailed Design:
Design is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures.
The detailed design may include:
•Decomposition of major system components into program units.
•Allocation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

3.Objectives of Software Design
1.Correctness: Software design should be correct as per requirement.
2.Completeness: The design should have all components like data structures, modules, and external interfaces, etc.
3.Efficiency: Resources should be used efficiently by the program.
4.Flexibility: Able to modify on changing needs.
5.Consistency: There should not be any inconsistency in the design.
6.Maintainability: The design should be so simple so that it can be easily maintainable by other designers.

4.Software Design Principles

Software design principles are concerned with providing means to handle the complexity of the design process effectively. Effectively managing the complexity will not only reduce the effort needed for design but can also reduce the scope of introducing errors during design.

Following are the principles of Software Design



Problem Partitioning

For small problem, we can handle the entire problem at once but for the significant problem, divide the problems and conquer the problem it means to divide the problem into smaller pieces so that each piece can be captured separately.

For software design, the goal is to divide the problem into manageable pieces.

Benefits of Problem Partitioning

1. Software is easy to understand
2. Software becomes simple
3. Software is easy to test
4. Software is easy to modify
5. Software is easy to maintain
6. Software is easy to expand

These pieces cannot be entirely independent of each other as they together form the system. They have to cooperate and communicate to solve the problem. This communication adds complexity.

> *Note:* ***As the number of partition increases = Cost of partition and complexity increases***

Abstraction

An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation. Abstraction can be used for existing element as well as the component being designed.

- Functional Abstraction
- Data Abstraction

Functional Abstraction

i. A module is specified by the method it performs.

ii. The details of the algorithm to accomplish the functions are not visible to the user of the function.

Data Abstraction

Details of the data elements are not visible to the users of data. Data Abstraction forms the basis for **Object Oriented design approaches**.

Modularity

Modularity specifies to the division of software into separate modules which are differently named and addressed and are integrated later on in to obtain the completely functional software. It is the only property that allows a program to be intellectually manageable. Single large programs are difficult to understand and read due to a large number of reference variables, control paths, global variables, etc.

The desirable properties of a modular system are:

- Each module is a well-defined system that can be used with other applications.
- Each module has single specified objectives.
- Modules can be separately compiled and saved in the library.

Modules should be easier to use than to build.

- Modules are simpler from outside than inside.

Advantages of Modularity

- It allows large programs to be written by several or different people
- It encourages the creation of commonly used routines to be placed in the library and used by other programs.
- It simplifies the overlay procedure of loading a large program into main storage.
- It provides more checkpoints to measure progress.
- It provides a framework for complete testing, more accessible to test
- It produced the well designed and more readable program.

Disadvantages of Modularity

- Execution time maybe, but not certainly, longer
- Storage size perhaps, but is not certainly, increased
- Compilation and loading time may be longer
- Inter-module communication problems may be increased
- More linkage required, run-time may be longer, more source lines must be written, and more documentation has to be done

Modular Design

Modular design reduces the design complexity and results in easier and faster implementation by allowing parallel development of various parts of a system. We discuss a different section of modular design in detail in this section:

1. **Functional Independence:** Functional independence is achieved by developing functions that perform only one kind of task and do not excessively interact with other modules. Independence is important because it makes implementation more accessible and faster. The independent modules are easier to maintain, test, and reduce error propagation and can be reused in other programs as well. Thus, functional independence is a good design feature which ensures software quality.

It is measured using two criteria:

- **Cohesion:** It measures the relative function strength of a module.
- **Coupling:** It measures the relative interdependence among modules.

2. **Information hiding:** The fundamental of Information hiding suggests that modules can be characterized by the design decisions that protect from the others, i.e., In other words, modules should be specified that data include within a module is inaccessible to other modules that do not need for such information.

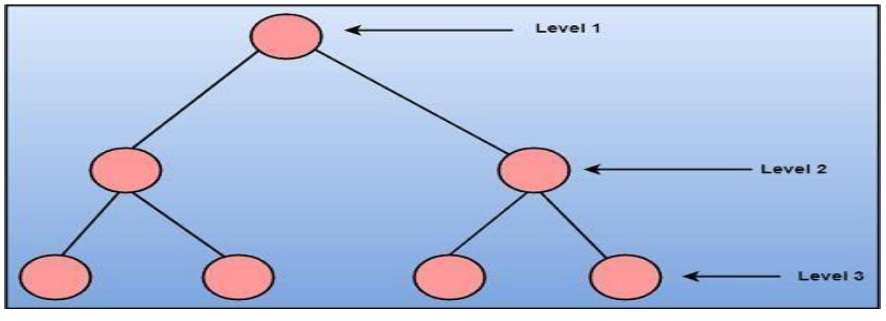
The use of information hiding as design criteria for modular system provides the most significant benefits when modifications are required during testing's and later during software maintenance. This is because as most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to different locations within the software.

---

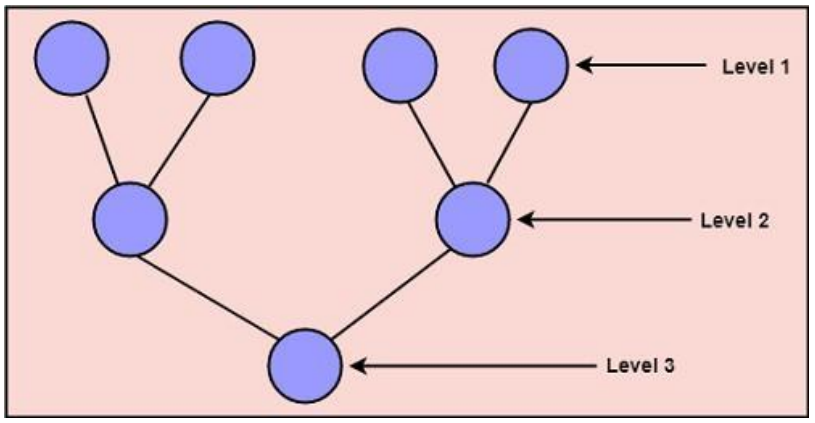
5. Strategy of Design

A good system design strategy is to organize the program modules in such a method that are easy t To develop and latter too, change. Structured design methods help developers to deal with the size and complexity of programs. Analysts generate instructions for the developers about how code should be composed and how pieces of code should fit together to form a program.

To design a system, there are two possible approaches:
1.Top-down Approach: This approach starts with the identification of the main components and then decomposing them into their more detailed sub-components.



2. **Bottom-up Approach:** A bottom-up approach begins with the lower details and moves towards up the hierarchy, as shown in fig. This approach is suitable in case of an existing system.

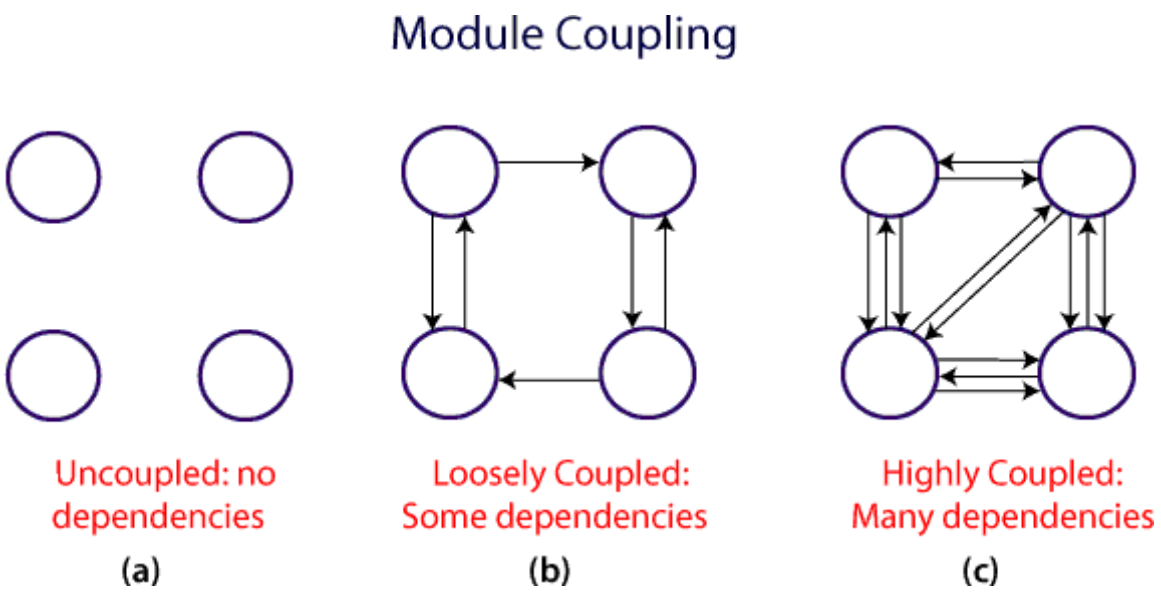


Coupling and Cohesion

Module Coupling

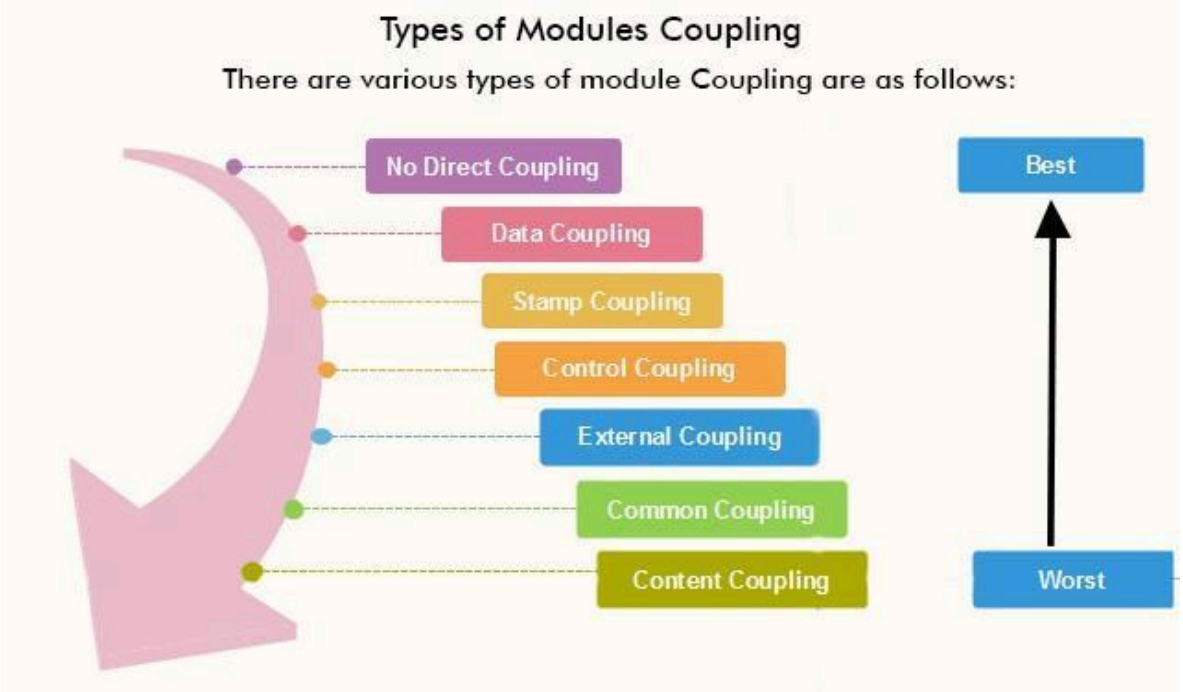
In software engineering, the coupling is the degree of interdependence between software modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. **Uncoupled modules** have no interdependence at all within them.

The various types of coupling techniques are shown in fig:

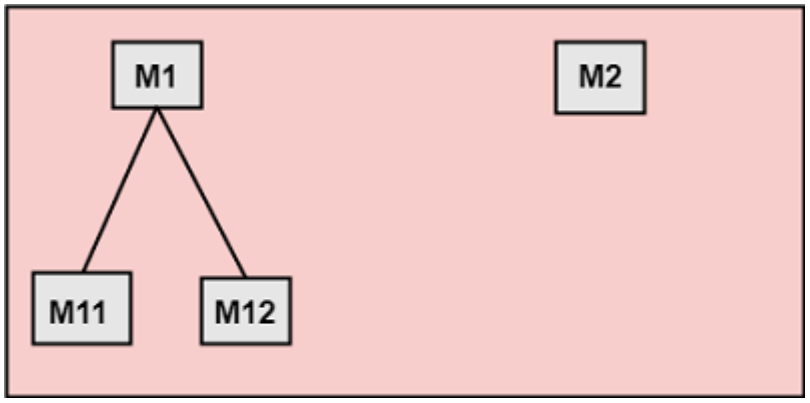


A good design is the one that has low coupling. Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors.

Types of Module Coupling

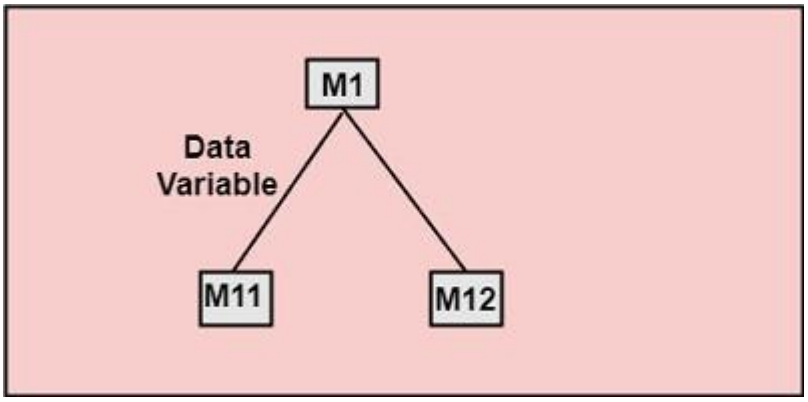


1. **No Direct Coupling:** There is no direct coupling between M1 and M2.



In this case, modules are subordinates to different modules. Therefore, no direct coupling.

2. **Data Coupling:** When data of one module is passed to another module, this is called data coupling.

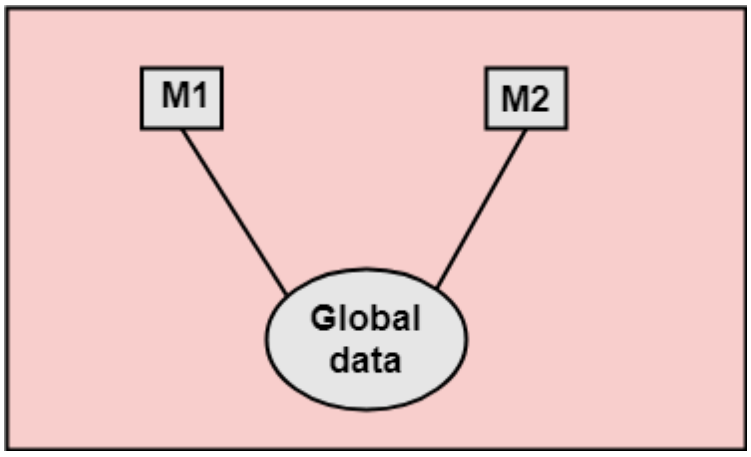


3. **Stamp Coupling:** Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.

4. **Control Coupling:** Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

5. **External Coupling:** External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

6. **Common Coupling:** Two modules are common coupled if they share information through some global data items.

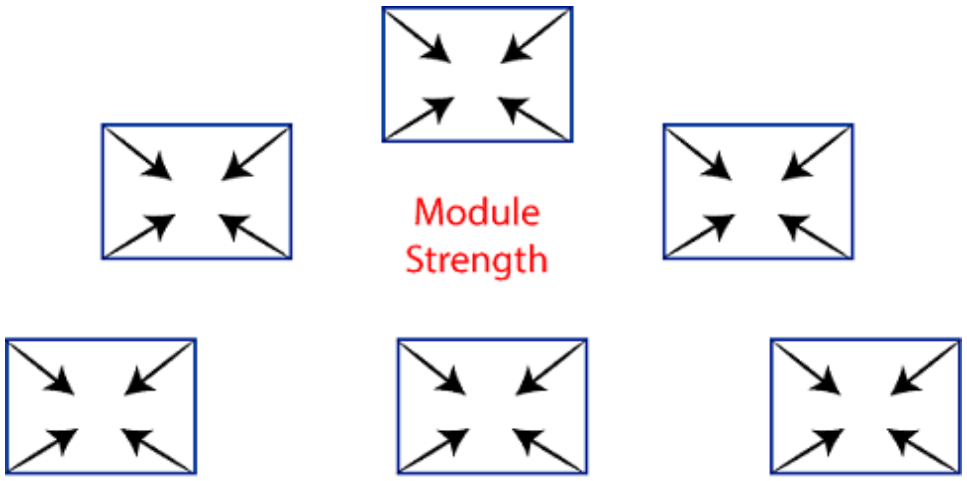


7. **Content Coupling:** Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

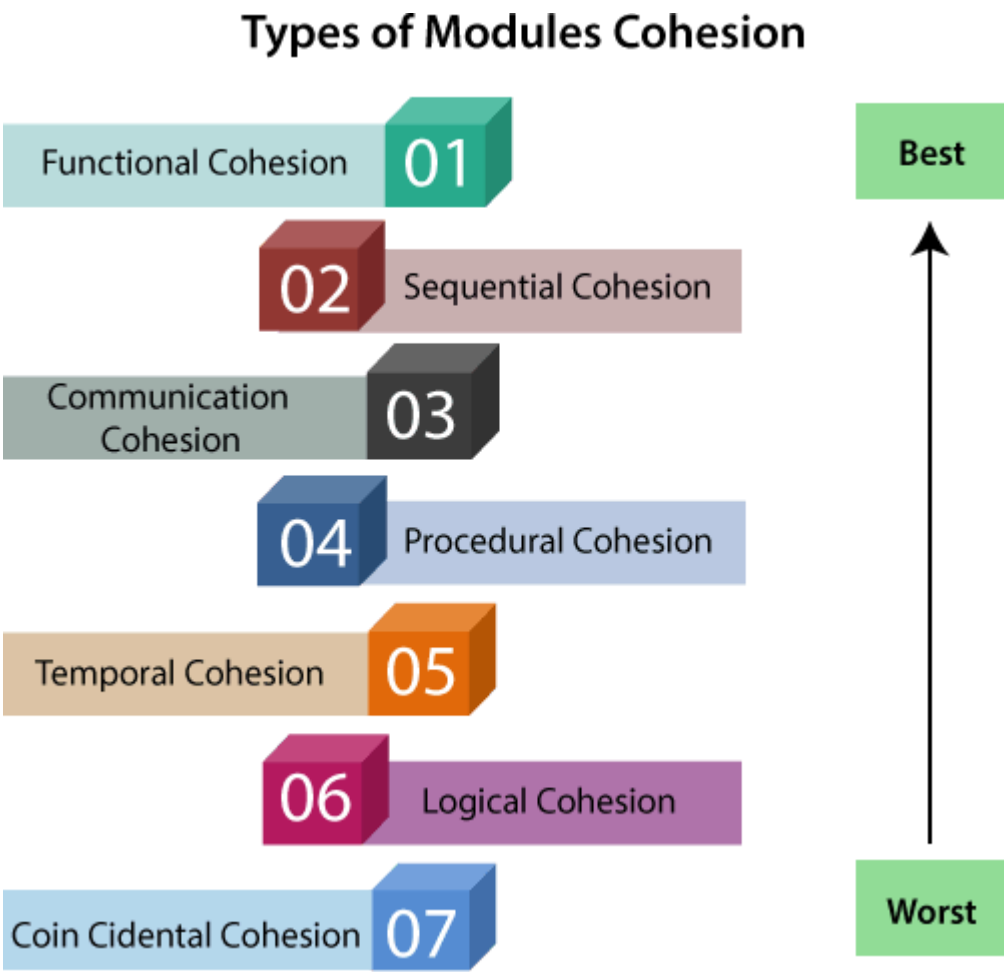
Module Cohesion

In computer programming, cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.

Cohesion is an **ordinal** type of measurement and is generally described as "high cohesion" or "low cohesion."



Cohesion= Strength of relations within Modules



1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.
2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.
3. **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.
4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.
5. **Temporal Cohesion:** When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.

6. **Logical Cohesion:** A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.
7. **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.

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.

6.DESIGN HEURSITIC

Design heuristics in software engineering are essentially guidelines that help engineers make decisions during the design process. They are not strict rules, but rather, they provide a framework that engineers can use to navigate the complex landscape of software design.

These heuristics are derived from years of experience and research in the field of software engineering. They encapsulate best practices and lessons learned from past projects, providing a roadmap for future software design endeavors.

Examples of design heuristics in software engineering

a.Modularity:

This heuristic encourages the division of a system into distinct, independent modules. This approach enhances readability, facilitates testing, and simplifies maintenance.

b.Abstraction:

This principle involves hiding the complexities of a system, revealing only what is necessary. It helps in managing complexity and promoting understandability.

c.Information Hiding:

This heuristic involves concealing the details of a module from other modules. It promotes independence among modules, making the system more robust and easier to modify.

Why are design heuristics in software engineering important?

Software developers discussing the importance of design heuristics in software engineering. Design heuristics in software engineering play a crucial role in the successful development of software systems. They guide engineers through the design process, helping them make informed decisions that result in efficient, reliable, and user-friendly software.

Without these heuristics, the design process can become chaotic and unmanageable. Engineers may overlook important aspects of the design, leading to software that is difficult to maintain, inefficient, or fails to meet user needs.

7.ARCHITECTURAL DESIGN

Architectural design :It is the process of defining a structured solution that meets all technical and operational requirements while optimizing performance, security, and maintainability. It defines the overall structure of the software system, including how components interact with each other and how data flows through the system.

1.Purpose of Architectural Design:

To establish a clear structure of the system.

To improve scalability and performance.

To enhance system flexibility and maintainability.

To ensure that the system meets functional and non-functional requirements.

2. Elements of Architectural Design

Components – Modular units that perform specific tasks.

Connectors – Communication pathways between components.

Data – Information processed and exchanged between components.

Interfaces – Define how components interact with each other.

Configuration – The overall structure of components and connectors.

8.ARCHIYECTURAL STYLE

1. Monolithic Architecture:
One of the earliest and most basic architectural forms is monolithic architecture. The system is intended to function as a single, self-contained unit in a monolithic application. Each component, including the data access layer, business logic, and user interface, is closely integrated into a single codebase.
Characteristics:
Tight Coupling: When parts are closely connected, it is challenging to scale or modify individual elements without influencing the system as a whole.
Simplicity: Small to medium-sized applications might benefit from monolithic architectures since they are easy to build and implement.
Performance: Monolithic programs can be quite performant because there are no inter-process communication overheads.
2. Layered Architecture:
Layered architecture, sometimes called n-tier architecture, divides the software system into several levels, each in charge of a certain task. Better system organization and maintainability are made possible by this division.
haracteristics:
Separation of Concerns: Distinct concerns, like data access, business logic, and display, are handled by different levels.
Scalability: The ability to scale individual layers allows for improved resource and performance usage.
Reusability: Reusing components from one layer in other applications or even in other sections of the system is frequently possible.

3. Architecture of Client-Server:
The system is divided into two primary parts by client-server architecture: the client, which is responsible for the user interface, and the server, which is in charge of data management and business logic. A network is used to facilitate communication between the client and server.
Characteristics:
Scalability: This design works well for large-scale applications since servers may be scaled independently to accommodate growing loads.
Centralized Data Management: Since data is kept on the server, security and management can be done centrally.
Thin Clients: Since most work occurs on the server, clients can be quite light.
4. Microservices Foundation:
A more modern architectural style called microservices architecture encourages the creation of autonomous, little services that speak to one another via APIs. Every microservice concentrates on a certain business function.
Characteristics:
Decomposition: The system is broken down into smaller, more manageable services to improve flexibility and adaptability.
Independent Deployment: Continuous delivery is made possible by microservices' ability to be deployed and upgraded separately.
Scalability: Individual services can be scaled to maximize resource utilization.
5. Event-Driven Architecture:

The foundation of event-driven architecture is the asynchronous event-driven communication between components. An event sets off particular responses or actions inside the system.
Characteristics:
Asynchronous Communication: Independently published, subscribed to, and processed events allow for component-to-component communication.
Loose coupling: Because of their loose coupling, event-driven systems have more flexibility regarding component interactions.
Scalability: Event-driven systems scale effectively and can withstand heavy loads.
6. Service-Oriented Architecture:
A type of architecture known as service-oriented architecture, or SOA, emphasizes providing services as the fundamental units of larger systems. Services may be coordinated to build large systems since they are meant to be autonomous, reusable, and flexible.
Characteristics:
Reusability: To minimize effort duplication, services are made to be used again in many situations.
Interoperability: SOA strongly emphasizes using open standards to ensure that services from various suppliers can cooperate.
Flexibility: Adaptability is made possible by orchestrating services to develop various applications.
7. Architecture Based on Components:
The use of reusable and interchangeable components in software system development is encouraged by component-based architecture. Each component is self-contained and contains a certain function.
Characteristics:
Reusability: Components can save time and effort during development by being utilized again in various situations.
Component isolation lessens the effect of modifications made to one component on the system.
Scalability: By adding additional instances of components, systems can be made larger.
8. Peer-to-Peer Architecture:
Peer-to-peer (P2P) architecture enables communication and resource sharing between networked devices or nodes without depending on a centralized server. Every network node can serve as both a client and a server.
Characteristics:
Decentralization: The lack of a single point of failure in P2P systems results from their decentralization.
Resource Sharing: Nodes can share resources such as files, processing power, and network bandwidth.
Autonomy: Every node inside the network possesses a certain level of autonomy, enabling it to make decisions on its own.
9. Architecture in N-Tiers:
An expansion of layered architecture, which divides the system into several tiers or layers, each with a distinct function, is known as N-tier architecture. Presentation, application, business logic, and data storage layers are examples of these tiers.
Characteristics:
Modularity: N-Tier designs divide intricate systems into more manageable, smaller parts.

Scalability: Performance can be optimized by scaling each layer independently.
Security: Data security can be improved by physically or logically separating data storage levels.
1 Cloud-Based Architecture:
Software systems are developed and delivered using cloud-based architecture, which uses cloud computing services. Outsourcing infrastructure to cloud service providers makes scalability, adaptability, and cost-effectiveness possible.
Characteristics:
Scalability: Cloud services are easily expandable or contracted to accommodate fluctuating needs.
Cost-effectiveness: Cloud-based architecture lowers initial hardware purchase requirements and ongoing maintenance expenses.

Software Architecture Patterns

1. Layered Architecture Pattern

n this pattern are separated into layers of subtasks and they are arranged one above another. Each layer has unique tasks to do and all the layers are independent of one another. Since each layer is independent, one can modify the code inside a layer without affecting others. It is the most commonly used pattern for designing the majority of software. This layer is also known as 'N-tier architecture'. Basically, this pattern has 4 layers.
Presentation layer: The user interface layer where we see and enter data into an application.)
Business layer: This layer is responsible for executing business logic as per the request.)
Application layer: This layer acts as a medium for communication between the 'presentation layer' and 'data layer'.
Data layer: This layer has a database for managing data.)

2 Client-Server Architecture Pattern

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.

3. Event-Driven Architecture Pattern

Event-Driven Architecture is an agile approach in which services (operations) of the software are triggered by events. When a user takes action in the application built using the EDA approach, a state change happens and a reaction is generated that is called an event.

4. Microkernel Architecture Pattern

Microkernel pattern has two major components. They are a core system and plug-in modules.
The core system handles the fundamental and minimal operations of the application.
The plug-in modules handle the extended functionalities (like extra features) and customized processing.

5. Microservices Architecture Pattern

The collection of small services that are combined to form the actual application is the concept of microservices pattern. Instead of building a bigger application, small programs are built for every service (function) of an application independently. And those small programs are bundled together to be a full-fledged application. So adding new features and modifying existing microservices without affecting other microservices are no longer a challenge when an application is built in a microservices pattern. Modules in the application of microservices patterns are loosely coupled. So they are easily understandable, modifiable and scalable.

6. Space-Based Architecture Pattern

Space-Based Architecture Pattern is also known as Cloud-Based or Grid-Based Architecture Pattern. It is designed to address the scalability issues associated with large-scale and high-traffic applications. This pattern is built around the concept of shared memory space that is accessed by multiple nodes.

7. Master-Slave Architecture Pattern

The Master-Slave Architecture Pattern is also known as Primary-Secondary Architecture. It involves a single master component and that controls multiple slave components. The master components assign tasks to slave components and the slave components report the results of task execution back to the master. This is often used for parallel processing and load distribution

UML (Unified Modeling Language)

It is standard visual language for describing and modelling software blueprints. The UML is more than just a graphical language. Stated formally, the UML is for: Visualizing, Specifying, Constructing, and Documenting. The artifacts of a software-intensive system (particularly systems built using the object-oriented style).

Three Aspects of UML

1. Language:

It enables us to communicate about a subject which includes the requirements and the system. It is difficult to communicate and collaborate for a team to successfully develop a system without a language.

2. Model:

It is a representation of a subject.

It captures a set of ideas (known as abstractions) about its subject.

3. Unified:

It is to bring together the information systems and technology industry's best engineering practices.

Conceptual Model of UML

The Unified Modeling Language (UML) is a standardized modeling language used in software engineering to visualize, specify, construct, and document software system designs. A Conceptual Model of UML defines the key elements and relationships involved in UML.

Elements of UML Conceptual Model

1.Thngs

2.Relationship

3.Diagrams

a. Things: it represents component that are modeled within a system. There are four types of "Things":

a) Structural Things: It represent static parts of a system.

Examples: Classes, Interfaces ,Components, Nodes

b) Behavioral Things: It represent dynamicparts of a system.

Examples: Iteractions, Sate Machines

c) Grouping Things: Organize elements into groups.

3. Relationships in UML
a) Association – Defines a structural relationship between objects.
b) Dependency – One element depends on another.
c) Generalization – Represents an inheritance relationship.
d) Realization – A class implements an interface.

4. UML Diagrams
a) Structural Diagrams
Class Diagram
Object Diagram
Component Diagram
Deployment Diagram

b) Behavioral Diagrams
Use Case Diagram
Sequence Diagram
Activity Diagram
State Diagram

CLASS DIAGRAM IN UML
Class diagrams are a type of UML (Unified Modeling Language) diagram used in software engineering to visually represent the structure and relationships of classes within a system i.e. used to construct and visualize object-oriented systems. n these diagrams, classes are depicted as boxes, each containing three compartments for the class name, attributes, and methods. Lines connecting classes illustrate associations, showing relationships such as one-to-one or one-to-many. Class diagrams provide a high-level overview of a system's design, helping to communicate and document the structure of the software. They are a fundamental tool in object-oriented design and play a crucial role in the software development lifecycle.

Class Diagram Notations

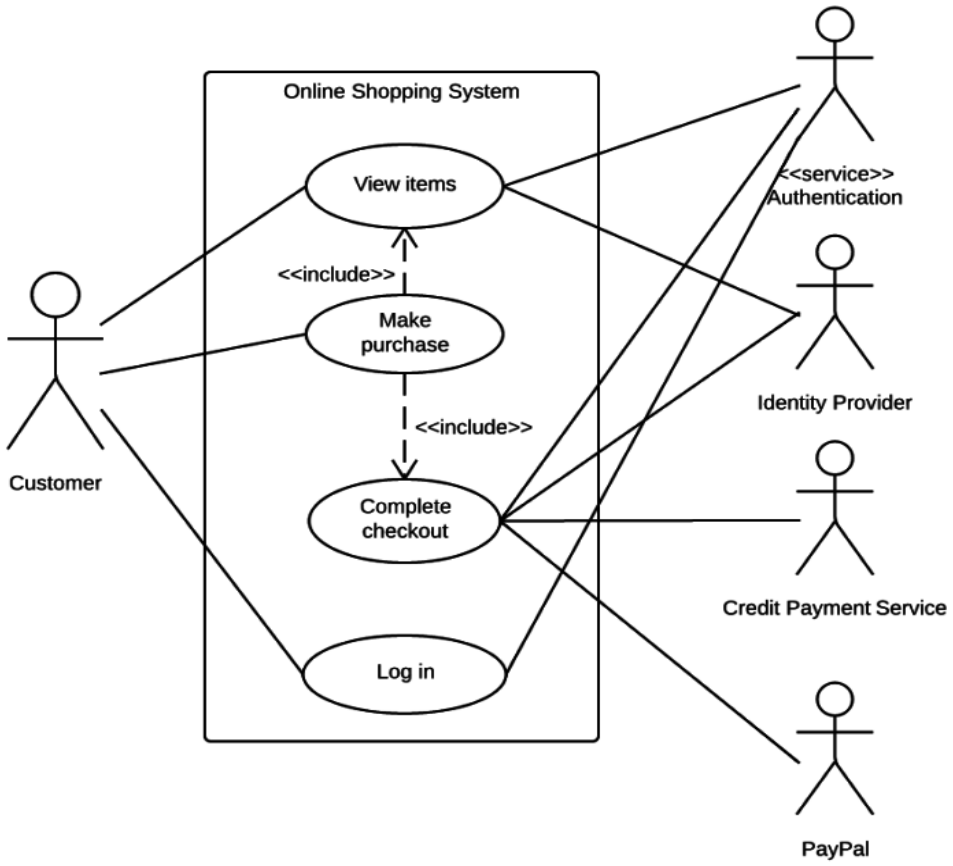
Class Name:
The name of the class is typically written in the top compartment of the class box and is centered and bold.
Attributes:
Attributes, also known as properties or fields, represent the data members of the class. They are listed in the second compartment of the class box and often include the visibility (e.g., public, private) and the data type of each attribute.
Methods:

Methods, also known as functions or operations, represent the behavior or functionality of the class. They are listed in the third compartment of the class box and include the visibility (e.g., public, private), return type, and parameters of each method.

Visibility Notation:

Visibility notations indicate the access level of attributes and methods. Common visibility notations include:

+ for public (visible to all classes)
- for private (visible only within the class)
for protected (visible to subclasses)
~ for package or default visibility (visible to classes in the same package)



CLASS Diagram for Online shopping System

SEQUENCE DIAGRAM

Sequence diagrams are a type of UML (Unified Modeling Language) diagram that visually represent the interactions between objects or components in a system over time. They focus on the order and timing of messages or events exchanged between different system elements. The diagram captures how objects communicate with each other through a series of messages, providing a clear view of the sequence of operations or processes.

Sequence Diagram Notations

1. Actors

An actor in a UML diagram represents a type of role where it interacts with the system and its objects. It is important to note here that an actor is always outside the scope of the system we aim to model using the UML diagram.

2. Lifelines

A lifeline is a named element which depicts an individual participant in a sequence diagram. So basically each instance in a sequence diagram is represented by a lifeline. Lifeline elements are located at the top in a sequence diagram

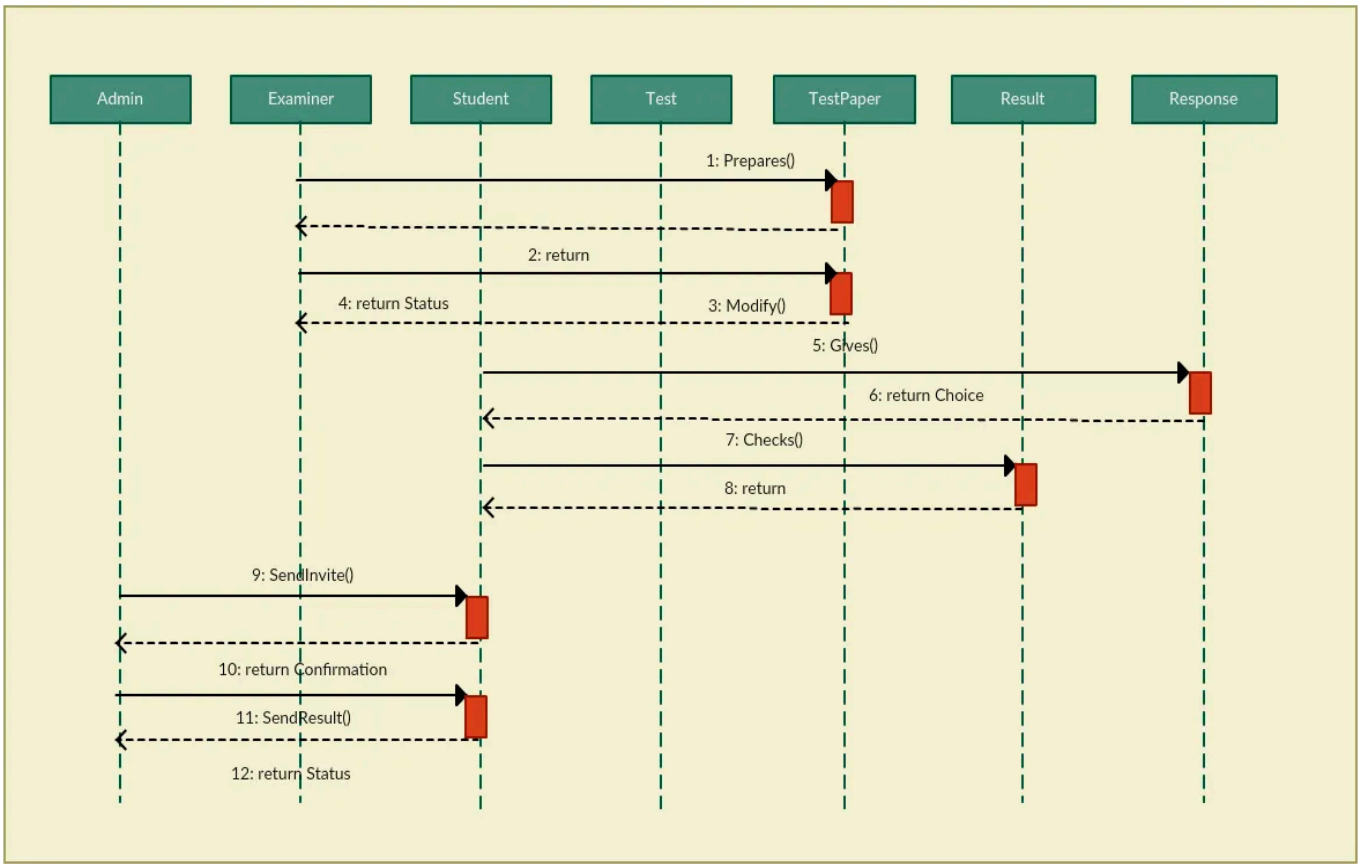
3 Messages

Communication between objects is depicted using messages. The messages appear in a sequential order on the lifeline.

We represent messages using arrows.

Lifelines and messages form the core of a sequence diagram.

Sequence Diagram of an Online Exam System

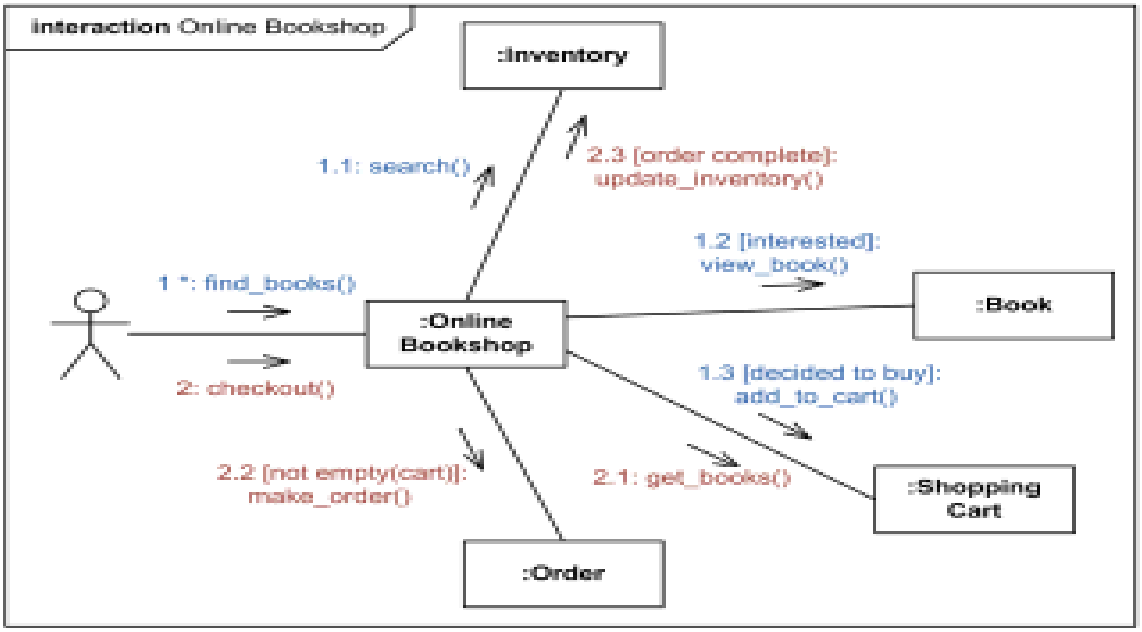


COLLABORATION DIAGRAM

A collaboration diagram is a behavioral UML diagram which is also referred to as a communication diagram. It illustrates how objects or components interact with each other to achieve specific tasks or scenarios within a system.

Components and their Notations in Collaboration Diagrams

1. Objects/Participants: Objects are represented by rectangles with the object's name at the top. Each object participating in the interaction is shown as a separate rectangle in the diagram. Objects are connected by lines to indicate messages being passed between them.
2. Multiple Objects :Multiple objects are represented by rectangles,each with the object's name inside, and interactions between them are shown using arrows to indicate message flows.
3. Actors: They are usually shown at the top or side of the diagram. Actors indicate their involvement in the interactions with the system's objects or components. They are connected to objects through messages, showing the communication with the system.
4. Messages: Messages represent communication between objects. Messages are shown as arrows between objects, indicating the flow of communication. Each message may include a label indicating the type of message (e.g., method call, signal). Messages can be asynchronous (indicated by a dashed arrow) or synchronous (solid arrow).
5.Self Messages: it is a message that an object sends to itself. It represents an action or behavior that the object performs internally without involving any other objects. Self-messages are useful for modeling scenarios where an object triggers its own methods or processes.
6. Links: Links represent associations or relationships between objects. Links are shown as lines connecting objects, with optional labels to indicate the nature of the relationship. Links can be uni-directional or bi-directional, depending on the nature of the association.
7. Return Messages :Return messages represent the return value of a message. They are shown as dashed arrows with a label indicating the return value. Return messages are used to indicate that a message has been processed and a response is being sent back to the calling object.



COMPONENT DIAGRAM

It shows how the components of a system are arranged and relate to one another is termed a component-based diagram, or simply a component diagram. System components are modular units that offer a set of interfaces and encapsulate implementation. These diagrams illustrate how components are wired together to form larger systems, detailing their dependencies and interactions.

Components of Component-Based Diagram

1. Component

Represent modular parts of the system that encapsulate functionalities. Components can be software classes, collections of classes, or subsystems.

Symbol: Rectangles with the component stereotype («component»).

Function: Define and encapsulate functionality, ensuring modularity and reusability.

2.Interfaces

Specify a set of operations that a component offers or requires, serving as a contract between the component and its environment.

Symbol: Circles (lollipops) for provided interfaces and half-circles (sockets) for required interfaces.

Function: Define how components communicate with each other, ensuring that components can be developed and maintained independently.

3. Relationships

Depict the connections and dependencies between components and interfaces.

4. Ports

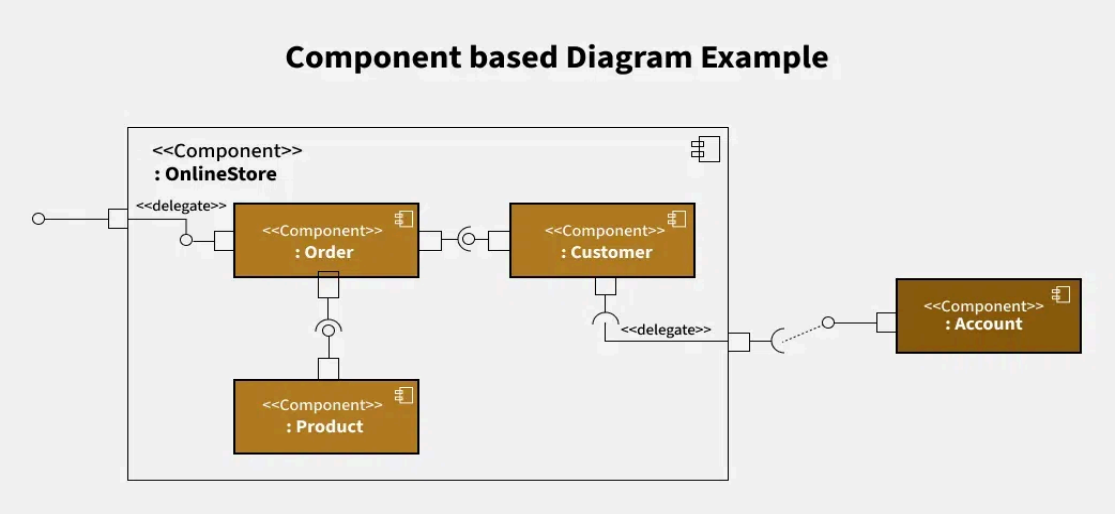
Role: Represent specific interaction points on the boundary of a component where interfaces are provided or required.

5. Artifacts

Represent physical files or data that are deployed on nodes.

6. Nodes

Represent physical or virtual execution environments where components are deployed.



6.Elements of Good Design

1. Modularity: Break down the system into smaller, independent, and reusable components.
Benefits:
a. to test and debug
b. improves maintainability and scalability
c. Promotes code reuse
2. Abstraction: Hide implementation details and expose only essential features.
Benefits:
a. Reduces complexity
b. Promotes reusability
3. Encapsulation: Bundle data and methods that operate on the data within a single unit (class). It protects the internal state of an object from unintended interference.
Benefits:
a. Improves security
b. Enhances flexibility and maintainability
4. Separation of Concerns: Divide a program into distinct sections where each section handles a specific responsibility. Example: MVC (Model-View-Controller) pattern
Benefits:
a. Reduces complexity
b. Improves testability and maintainability
5. Single Responsibility Principle (SRP): A class should have only one reason to change (one responsibility). Keeps the code clean and organized.
Benefits:
a. Easier to understand and modify
b. Reduces the risk of unintended side effects
6. Open/Closed Principle: Software entities should be open for extension but closed for modification.
Benefits:
a.Promotes flexibility
b.Reduces the need to alter existing code
7. Liskov Substitution Principle: Objects of a superclass should be replaceable with objects of a subclass without altering the behavior. Ensures that inheritance follows expected behavior.
Benefits:
a.Promotes polymorphism
b.Enhances code reliability
8. Interface Segregation Principle: Clients should not be forced to depend on interfaces they do not use. Create smaller, more specific interfaces instead of large ones.
Benefits:
a. Reduces dependency issues
b. Improves code clarity and organization
9. Dependency Inversion Principle: High-level modules should not depend on low-level modules. Both should depend on abstractions.
Benefits:
a.Promotes loose coupling
b.Enhances flexibility
10. Consistency: Use consistent naming conventions, code styles, and structure.

Benefits:
a.Easier to understand and maintain
b.Improves team collaboration
11. Flexibility and Extensibility: Design the system to adapt to future changes with minimal impact.
Benefits:
a.Reduces future development effort
b.Increases scalability
12. Performance and Efficiency: Optimize for speed, memory usage, and resource consumption. Avoid premature optimization — focus on clean design first.
Benefits:
a. Faster execution
b. Better user experience
13. Testability: Write code that is easy to test. Use dependency injection, modularization, and interfaces.
Benefits:
a. Easier to catch and fix bugs
b. Improves code quality
14. Scalability: Design to handle increasing load and data volumes. It Use patterns like:
Load balancing, Caching, Database sharding
Benefits:
a. Ensures smooth performance under load
b. Improves user experience
15. Security: Follow secure coding practices. Examples: Input validation Authentication and authorization, Data encryption
Benefits:
a. Protects user data
b. Reduces vulnerabilities

Design Issues in Modern GUI
a) Usability: The interface should be easy to understand and operate. It minimize the learning curve for new users. It provide intuitive navigation and clear instructions.
b) Consistency: It Maintain uniform color schemes, fonts, and layouts. Similar actions should produce consistent results. It use standard icons and buttons across the application.
c) Accessibility: It ensure the interface is usable by people with disabilities. It provide options for text size adjustment, color contrast, and screen readers. It ensure compatibility with assistive technologies.
d) Performance: Fast response times and smooth transitions. Minimal system resource consumption. Efficient handling of large data sets.
e) Error Handling: It dsisplay clear error messages. It provide actionable suggestions to resolve errors. It allows undo/redo options where possible.

f) Adaptability and Responsiveness: GUI should adapt to different screen sizes and resolutions. Ensure compatibility with various devices (mobile, desktop, tablets). Support different operating systems and browsers.

g) Feedback and Confirmation: It provide immediate visual or audio feedback upon user action. It use progress indicators and status bars for long-running tasks. It provide Confirmation dialogs for critical actions (e.g., delete).

Features of Modern GUI

a) Menus

Dropdown or pop-up menus for easy navigation.

Contextual menus based on the current state of the application.

b) Scroll Bars

Horizontal and vertical scroll bars for navigating large content.

Auto-hide or fixed-position scroll bars.

c) Windows

Multiple window support (modal and non-modal).

Ability to minimize, maximize, resize, and close windows.

d) Buttons

Command buttons (OK, Cancel, Apply) for executing tasks.

Toggle buttons and radio buttons for selecting options.

e) Icons

Graphical representations of actions or files.

Tooltips for additional information on hover.

f) Panels

Organized sections for grouping related information.

Collapsible and expandable panels.

g) Toolbars

Quick access to frequently used functions.

Customizable layout and content.

h) Dialog Boxes

Modal and non-modal dialog boxes for user interaction.

Input validation and error messages.

i) Status Bars

Display system information and current state.

Provide real-time updates on ongoing processes.

j) Drag and Drop

Allows moving files or components within the interface.

Supports intuitive rearrangement of elements.

k) Error Messages

Clear and descriptive error messages.

Color-coded messages for different severity levels.