Unit III - Software Design

Software design – Design process – Design concepts – Coupling – Cohesion – Functional independence – Design patterns – Model-view-controller – Publish -subscribe – Adapter – Command – Strategy – Observer – Proxy – Facade – Architectural styles – Layered - Client Server - Tiered - Pipe and filter- User interface design-Case Study.

Software Design Process

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.

Phases of software design process

- ✓ High Level Design / Interface Design
- ✓ Architectural Design
- ✓ Detailed Design

Elements of a System

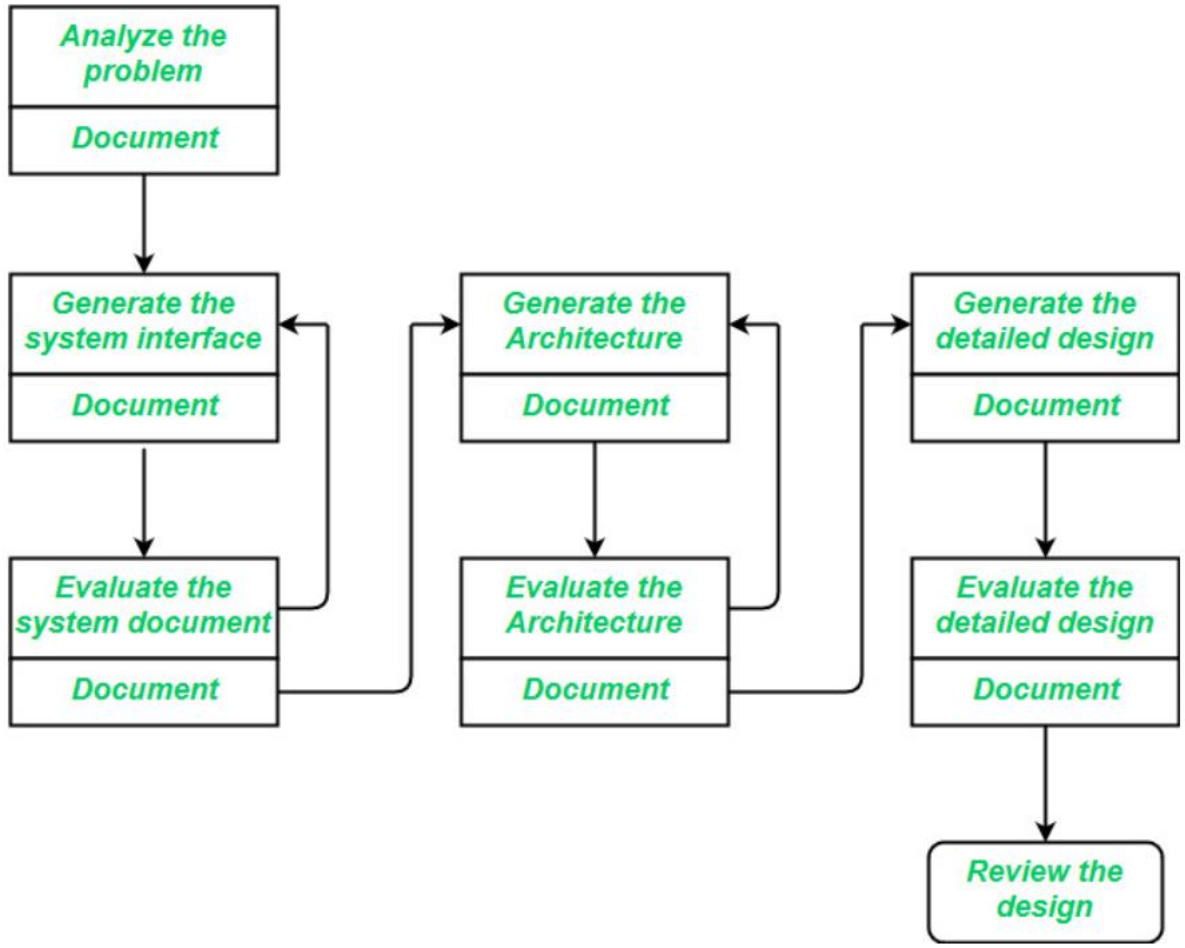
- ✓ **Architecture -** This is the conceptual model that defines the structure, behavior, and views of a system. We can use flowcharts to represent and illustrate the architecture.
- ✓ **Modules -** These are components that handle one specific task in a system. A combination of the modules makes up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

Objectives of Software Design

- ✓ **Correctness:** A good design should be correct, which means that it should correctly implement all of the system's features.
- ✓ **Efficiency:** A good software design should consider resource, time, and cost optimization parameters.
- ✓ **Understandability:** A good design should be easy to grasp, which is why it should be modular, with all parts organized in layers.

- ✓ **Completeness:** The design should include all components, such as data structures, modules, and external interfaces, among others.
- ✓ **Maintainability:** A good software design should be flexible when the client issues a modification request.

Phases of Software Design Process



Software Design Process

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. During interface design, the internal of the systems are completely ignored, and the system is treated as a black box.

The Interface Design includes

- ✓ 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.

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.

Issues in Architectural Design Includes

- ✓ Decomposition of the systems into major components.
- ✓ Allocation of functional responsibilities to components.
- ✓ Component Interfaces.
- ✓ Component scaling and performance properties.
- ✓ Component resource consumption and reliability properties
- ✓ Communication and interaction between components.

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 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.

Software Design Concepts

The set of fundamental software design concepts,

- ✓ Abstraction

- ✓ Architecture
- ✓ Modularity
- ✓ Design Patterns
- ✓ Information/Data Hiding
- ✓ Functional independence
- ✓ Refinement
- ✓ Refactoring

Abstraction

- ✓ A solution is stated in large terms using the language of the problem environment at the highest level abstraction.
- ✓ The lower level of abstraction provides a more detail description of the solution.
- ✓ A sequence of instruction that contain a specific and limited function refers in a procedural abstraction.
- ✓ A collection of data that describes a data object is a data abstraction.

Architecture

- ✓ The complete structure of the software is known as software architecture.
- ✓ Structure provides conceptual integrity for a system in a number of ways.
- ✓ The architecture is the structure of program modules where they interact with each other in a specialized way.
- ✓ The components use the structure of data.
- ✓ The aim of the software design is to obtain an architectural framework of a system.
- ✓ The more detailed design activities are conducted from the framework.

Modularity

- ✓ **Modularity refers** to breaking a system or project into smaller sections to lessen the system's or project's complexity.
- ✓ Similarly, modularity in design refers to the division of a system into smaller elements that can be built independently and then used in multiple systems to execute different purposes.

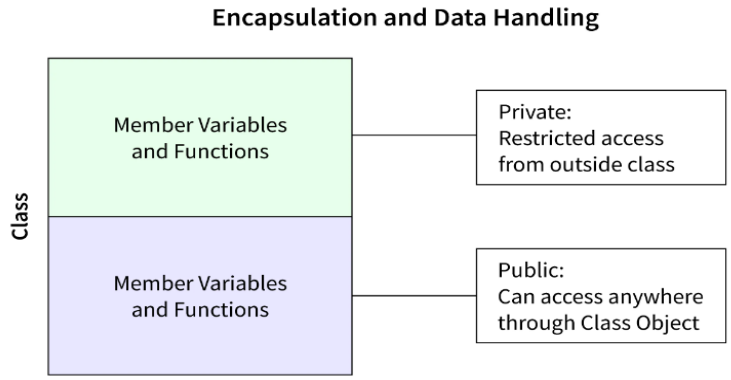
Design Patterns

- ✓ A Software Design Pattern is a general, reusable solution to a commonly occurring problem within a given context in software design.
- ✓ They are templates to solve common software engineering problems, representing some of the finest practices experienced object-oriented software engineers utilize.
- ✓ It also provides implementation guidance and examples.

Information/Data Hiding

- ✓ Information hiding implies concealing/hiding information so that an unauthorized entity cannot access it.

✓ In software design, **information hiding** is accomplished by creating modules in such a way that information acquired or contained in one module is concealed and cannot be accessible by other modules.



Functional Independence

✓ The functional independence is the concept of separation and related to the concept of modularity, abstraction and information hiding.

✓ The functional independence is accessed using two criteria i.e Cohesion and coupling.

- ***Cohesion*** is an indication of the ***relative functional strength of a module***.
- ***Coupling*** is an indication of the ***relative interdependence among modules.***

Refinement

✓ Refinement means removing any impurities and improving the quality of something.

✓ The software design refinement idea is a process of building or presenting the software or system in a detailed manner, which implies elaborating on a system or software.

✓ Refinement is essential for identifying and correcting any possible errors.

Refactoring

✓ Refactoring is the process of **reorganizing code** without affecting its original functionality.

✓ Refactoring aims to improve internal code by making modest changes that do not affect the code's exterior behavior.

✓ Refactoring increases code readability while decreasing complications.

✓ Refactoring can also assist software engineers in locating faults or vulnerabilities in their code.

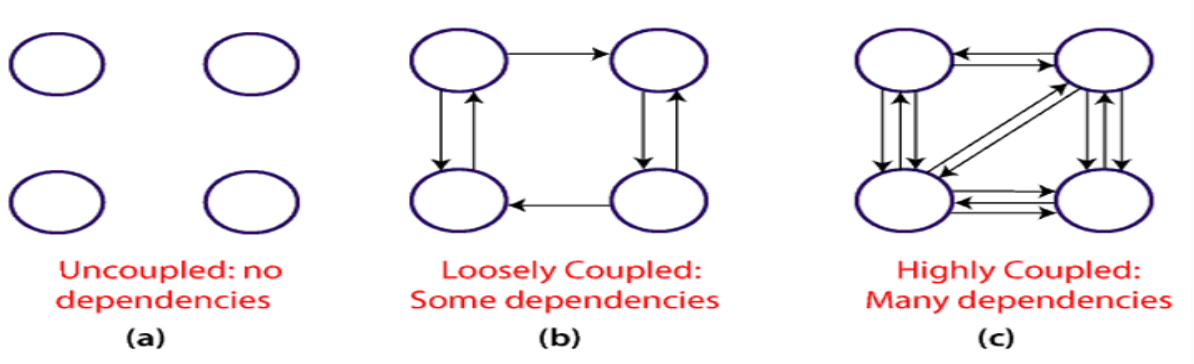
Functional Independence

Functional independence is a key to good design, Independence is measured using cohesion and coupling. A functionally independent module has minimal interaction with other modules.

- ***Coupling*** is an indication of the ***relative interdependence among modules***.
- ***Cohesion*** is an indication of the ***relative functional strength of a module*.**

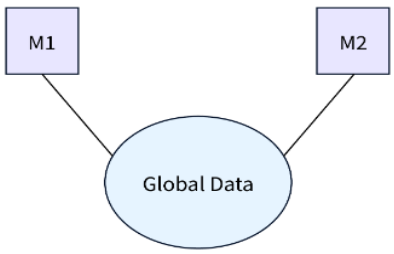
Coupling

- ✓ Coupling refers to the degree of interdependence between different modules, classes, or components of a software system.
- ✓ It shows how closely these elements relate to each other and how much one element depends on the behaviour, data or interfaces of another.
- ✓ High coupling means strong interconnections where changes in one module can cascade through others, while low coupling means greater independence and isolation between modules.



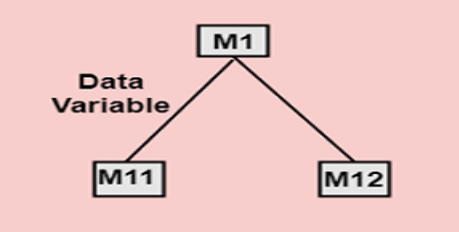
Type of Coupling

1. **Content Coupling** - Modules share data directly through global variables or parameters. This is the strongest coupling method and is not recommended because it tightly couples the modules and makes them highly dependent on each other.
2. **General Coupling -** Modules share global data or resources that are frequently used and modified by different modules. Although not as direct as pooling content, it still represents tight pooling through shared resources.

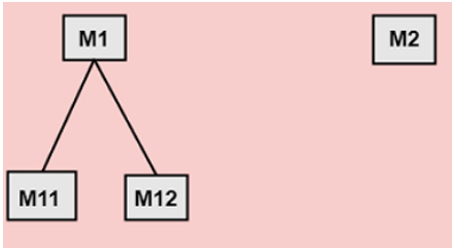


3. **External Coupling** - Modules communicate by exchanging data through external interfaces such as function parameters or method calls. Although external binding is more flexible than content and general binding, it can still cause dependencies.
4. **Control Coupling** - One module affects the behaviour of another by passing control information, often through parameters. This type of connection may be less direct than a content connection but still requires close communication.

5. **Stamp Coupling -** Modules share a composite data structure such as a record or object without sharing. Changes to the structure can affect several modules, but the connection is weaker than in the content connection.
6. **Data Coupling** - Modules share data through parameters, but there is no direct relationship between functions. Compared to the previous types, it is a relatively loose form of connection.

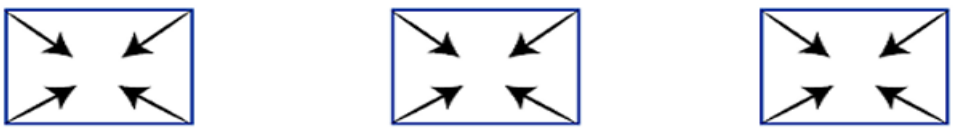


7. **No Coupling** - Modules work independently without direct communication. This is the ideal type of connection to aim for as it encourages modular design and minimizes the impact of changes. No direct connection between M1 and M2.



Cohesion

- ✓ Cohesion in software engineering refers to the degree of interrelatedness and focus among the elements within a module, class, or component.
- ✓ It measures how well the internal components of a module work together to achieve a single, well-defined purpose.
- ✓ High cohesion indicates that the elements within a module are closely related and contribute collectively to a specific functionality. Low cohesion suggests that the elements are less focused and may serve multiple unrelated purposes.



Cohesion= Strength of relations within Modules

Types of Cohesion

1. **Functional Cohesion -** Elements within a module are grouped based on a single, specific functionality or task. This is the strongest form of cohesion, where all elements contribute to the same goal.

2. **Sequential Cohesion -** Elements are organized in a linear sequence, where the output of one element becomes the input of the next. This type of cohesion is often seen in processes with step-by-step execution.
3. **Communicational Cohesion -** Elements within a module work together to manipulate a shared data structure. They might not perform the same function, but their actions are closely related to a common piece of data.
4. **Procedural Cohesion -** Elements are grouped based on their involvement in a specific sequence of actions or steps. They might share some data, but their primary focus is on the sequence of operations.
5. **Temporal Cohesion -** Elements are grouped because they need to be executed at the same time or during the same phase. They might not share functional or data-related aspects.
6. **Coincidental Cohesion -** Elements are grouped arbitrarily without a clear, meaningful relationship. This type of cohesion is typically indicative of poor module design.

Difference Between Coupling and Cohesion

Coupling	Cohesion
Degree of interdependence between modules or components within a system.	Degree of relatedness and focus within a module or component.
Interaction between modules. Coupling represents module dependency.	Composition of elements within a module. Cohesion represents module unity and purpose.
Changes in one module can impact others.	Changes within a module are contained.
High coupling increases maintenance complexity, as changes are widespread.	High coupling increases maintenance complexity, as changes are widespread.
Coupled modules are harder to test in isolation.	Cohesive modules are easier to test, as functionality is well-contained.
Coupled modules are less reusable due to dependencies.	Cohesive modules are more reusable due to clear and focused functionality.
Aim for low coupling to minimize interdependencies.	Aim for high cohesion to ensure focused and understandable modules.

Design Patterns

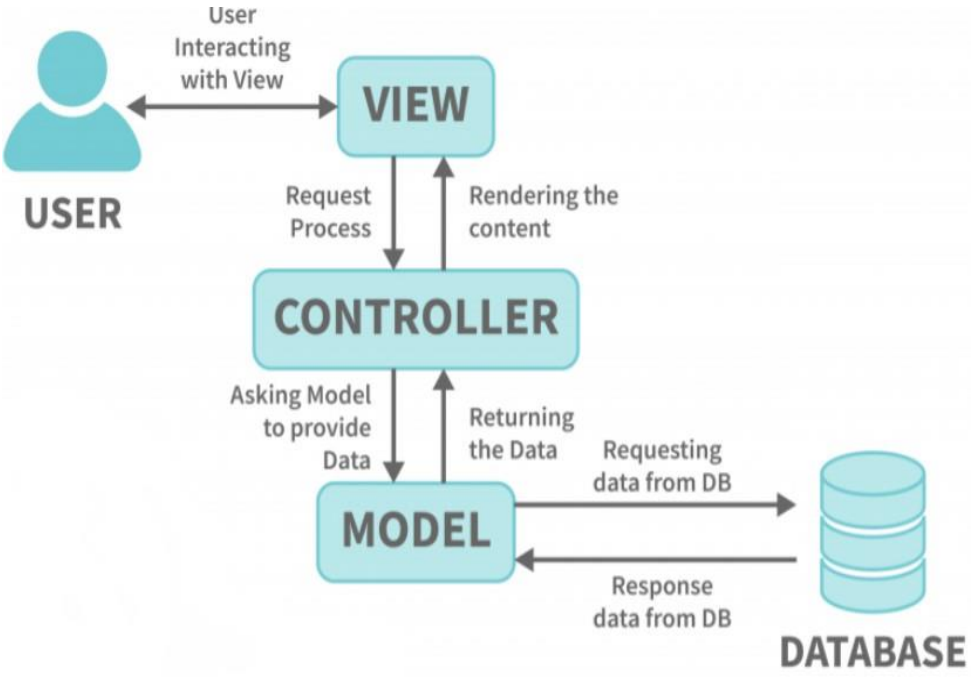
Model View Controller Architecture

The **Model View Controller** (MVC) design pattern specifies that an application consists of a data model, presentation information, and control information. MVC Architecture is the design pattern that is used to build the applications.

This architectural pattern was mostly used for Web Applications. It is used to break up a large application into smaller sections.

Components of MVC

- The MVC architecture pattern turns complex application development into a much more manageable process. It allows several developers to simultaneously work on the application.
- The complexity of modern web applications can make it difficult to make changes. When the frontend and backend are separated, the application is scalable, maintainable, and easy to expand.
  - ✓ **Model** - *Middleware that handles the database operations* (i.e.) The backend that contains all the data logic
  - ✓ **View** - *Where the user interacts* (i.e.) The frontend or graphical user interface (GUI).
  - ✓ **Controller** - *Process the request and send the response to view* (i.e.) The brains of the application that controls how data is displayed.



View

- ✓ View represents the way that data is presented in the application.
- ✓ The views are created based on the data collected from the model.
- ✓ By requesting information from the model, the user is presented with the output presentation.
- ✓ Besides representing the data from charts, diagrams, and tables, the view also displays data from other sources.

- ✓ All user interface components, such as text boxes, drop-down menus, etc. will appear in any customer view.

Controller

- ✓ Controllers are those components of an application that handle user interaction.
- ✓ User input is interpreted by the controller, causing the model and view to change based on the information it receives.
- ✓ By communicating with a controller's associated view, a user can change the view's appearance and update the state of its associated model.

Model

- ✓ A model component stores data and logic.
- ✓ A Controller object will retrieve customer information from a database. Data is transferred between the controller components or between business logic elements.
- ✓ It manipulates data and sends it back to the database, or it is used to render the same information.
- ✓ Additionally, it responds to views' requests and has instructions from the controller that allow it to update itself. It is also the lowest level of the pattern responsible for maintaining the data.

Advantages

- It has a Maintainable code that can be extended and grown easily.
- Its components can be tested separately from the user components.
- It supports new types of clients more easily.
- It is possible to parallelize the development of various components.
- By breaking an application up into three parts, it can avoid complexity.
- It supports test-driven development to the fullest extent.
- A controller's actions can be logically grouped using MVC design patterns.

Disadvantages

- Reading, changing, and unit testing this model is difficult because there is no separate component available to handle UI.
- The MVC architecture offers no formal validation support. So the validations explicitly need to be done.
- It may result in an inefficient data processing process because it makes the implementation logic a bit complex.
- Programmers should be familiar with multiple technologies.

GRASP (General Responsibility Assignment Software Patterns) Principles: Object-Oriented Design Patterns

Object-oriented design is a crucial aspect of software development, and creating robust and maintainable systems requires a systematic approach. GRASP is a set of guidelines that helps designers make informed decisions during the object-oriented design process.

By adhering to these principles, developers can create software that is modular, flexible, and maintainable. Applying *GRASP principles ensures a clear distribution of responsibilities, promotes low coupling, and high cohesion among classes.*

The key principles of GRASP

- ✓ Creator
- ✓ Information Expert
- ✓ Low Coupling
- ✓ High Cohesion
- ✓ Controller
- ✓ Polymorphism
- ✓ Pure Fabrication

Creator

- The Creator principle guides the allocation of responsibility for creating objects. According to this principle, a class should be responsible for creating objects of other classes if the first-class aggregates, contains, or has a composition relationship with the second class.
- This principle promotes low coupling between classes, enabling better maintainability and reusability.

Information Expert

- The Information Expert principle focuses on assigning responsibilities to classes that possess the most information required to fulfil them.
- A class should be responsible for a particular task if it has the necessary information to perform that task effectively. By adhering to this principle, can able to design systems where responsibilities are distributed efficiently among classes, improving cohesion and reducing dependencies.

Low Coupling

- The Low Coupling principle emphasizes designing classes with minimal dependencies on other classes. By reducing the interconnections between classes, create a more flexible and modular system.

- Achieving low coupling improves maintainability, as changes to one class are less likely to affect other classes. The use of interfaces and abstractions can help minimize coupling and promote code reusability.

High Cohesion

- High Cohesion is a principle that advocates designing classes with a clear and focused purpose.
- A class should have a single responsibility and encapsulate related behaviors and data. Cohesive classes are easier to understand, test, and maintain.
- They promote code reuse and minimize the impact of changes within the system.

Controller

- The Controller principle suggests that an entity responsible for handling a system operation should be a separate class.
- Controllers act as intermediaries between the user interface and the business logic, orchestrating the flow of information and managing the interaction between objects.
- By encapsulating the control logic in dedicated controller classes, it achieve better separation of concerns and improve the overall flexibility of the system

Polymorphism

- Polymorphism is a fundamental principle in object-oriented design that enables objects of different classes to be treated uniformly through a common interface.
- By leveraging polymorphism, can able to design systems that are extensible and adaptable to new requirements.
- Polymorphic behavior allows for code reuse and promotes loosely coupled systems.

Pure Fabrication

- The Pure Fabrication principle suggests the creation of artificial classes to improve system design. These classes are not directly tied to the domain or problem being solved but serve as helpers or connectors between other classes.
- Pure Fabrication aids in achieving low coupling and high cohesion by encapsulating complex operations or providing an interface to external systems.

Gang of Four (GoF) Design Patten

A design pattern designed to provide a flexible solution to various object creation problems in object-oriented programming. The intent of the GoF design pattern is to separate the construction of a complex object from its representation. It is one of the Gang of Four design patterns.

GoF Design Pattern Types

GoF Design Patterns are divided into three categories

- ✓ **Creational Pattern** (for the creation of objects) - The design patterns that *deal with the creation of an object.*
- ✓ **Structural Pattern** (to provide relationship between objects) - The design patterns in this category *deals with the class structure such as Inheritance and Composition.*
- ✓ **Behavioral Pattern** (to help define how objects interact) - This type of design patterns *provide solution for the better interaction between objects, how to provide lose coupling, and flexibility* to extend easily in future.

Creational Design Patterns Types

1. ***Abstract Factory*** **-** Allows the creation of objects without specifying their concrete type.
2. ***Builder*** - Uses to create complex objects.
3. ***Factory Method*** - Creates objects without specifying the exact class to create.
4. ***Prototype*** - Creates a new object from an existing object.
5. ***Singleton*** - Ensures only one instance of an object is created.

Structural Design Patterns Types

1. ***Adapter*** **-** Allows for two incompatible classes to work together by wrapping an interface around one of the existing classes.
2. ***Bridge*** **-** Decouples an abstraction so two classes can vary independently.
3. ***Composite*** **-** Takes a group of objects into a single object.
4. ***Decorator*** **-** Allows for an object's behavior to be extended dynamically at run time.
5. ***Facade*** **-** Provides a simple interface to a more complex underlying object.
6. ***Flyweight*** - Reduces the cost of complex object models.
7. ***Proxy*** **-** Provides a placeholder interface to an underlying object to control access, reduce cost, or reduce complexity.

Behavior Design Patterns Types

1. ***Chain of Responsibility*** - Delegates commands to a chain of processing objects.
2. ***Command*** - Creates objects which encapsulate actions and parameters.
3. ***Interpreter*** **-** Implements a specialized language.
4. ***Iterator*** **-** Accesses the elements of an object sequentially without exposing its underlying representation.
5. ***Mediator*** **-** Allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
6. ***Memento*** **-** Provides the ability to restore an object to its previous state.

7. ***Observer*** **-** Is a publish/subscribe pattern which allows a number of observer objects to see an event.
8. ***State*** - Allows an object to alter its behavior when its internal state changes.
9. ***Strategy*** **-** Allows one of a family of algorithms to be selected on-the-fly at run-time.
10. ***Template Method*** **-** Defines the skeleton of an algorithm as an abstract class, allowing its sub-classes to provide concrete behavior.
11. ***Visitor*** - Separates an algorithm from an object structure by moving the hierarchy of methods into one object.

Publisher-Subscriber (Pub/Sub) Pattern

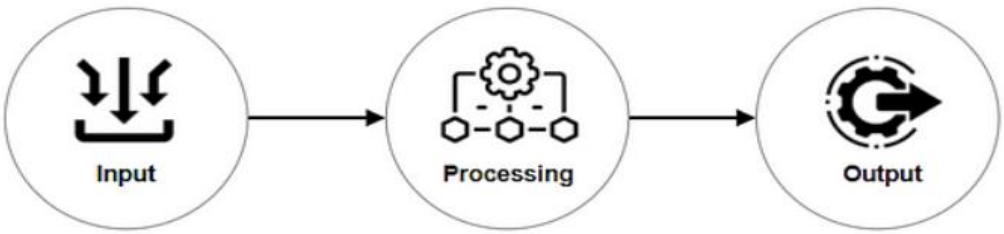
The Publish/Subscribe pattern, sometimes known as pub/sub, is an architectural design pattern that enables publishers and subscribers to communicate with one another.

- ✓ In this arrangement, the publisher and subscriber rely on a message broker to send messages from the publisher to the subscribers. Messages (events) are sent out by the host (publisher) to a channel, which subscribers can join.
- ✓ Pub/Sub is more versatile and scalable.
- ✓ The key to this is that Pub/Sub allows messages to flow between different system components without the components knowing one other's identities.

Publish-Subscribe Pattern Work

- ✓ Building reusable arrangements of modules and their interconnections is the foundation of software design patterns. In a UML design diagram, these modules are often classes or objects.
- ✓ Modern architectural patterns, it *sees modules as larger, self-executing processes scattered over distributed systems.*
- ✓ An information system is often made up of a generic ***set of software modules that are organized in this simple sequential structure.***

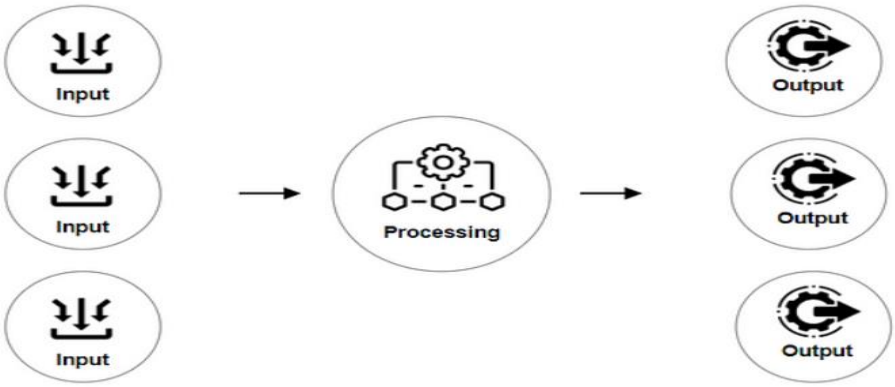
1. A simple software made up of three parts.



- ✓ ***The input module*** receives user input and converts it into a message that is sent to the processing module.

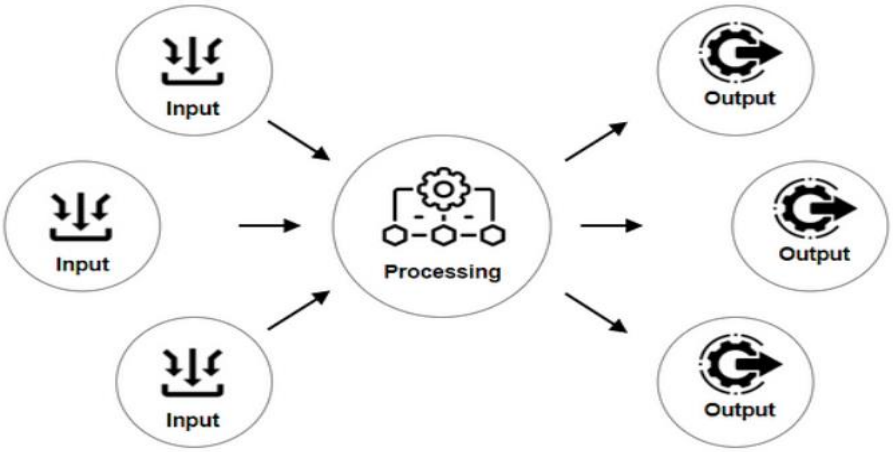
- ✓ ***The data is processed*** by the processing module and sent to the output module as a new message.
- ✓ The data is shown on the user's screen using ***the output module***.

2. For handling concurrent queries at a suitable size, the system will require several inputs and output modules.



- ✓ The system faces the difficulty of routing messages from input modules to their appropriate output modules at this scale.
- ✓ The input and output modules will require an addressing mechanism to handle this challenge.
- ✓ The messages will be processed and routed to the relevant recipient based on an address by the processing module. In order to solve the routing problem, all three modules work together.

3. In order to solve the routing problem, all modules working together.



- ✓ The system will be able to manage thousands of concurrent connections at an Internet-scale.
- ✓ Users from all over the world will send and receive messages using the system. It must also be able to handle a large number of users from all over the world.

- ✓ The system modules, on the other hand, will not work as planned at such a vast scale.
    - The load is too much for the processing module to handle. The load must be divided across numerous processing modules due to the high volume and geographical spread.
    - The dynamics of input and output change at this scale. The use of pre-defined addressing between modules adds a significant amount of overhead.

- ✓ The first issue can be solved by using numerous processing units. This *has the effect of horizontally separating the system. This*, however, adds to the routing complexity. The messages must now be routed to the appropriate processing module by the input modules.

Advantages of Pub/Sub model

- ✓ ***Loose Coupling*** - between components, making your system more modular and flexible.
- ✓ ***High scalability*** - Pub/Sub allows any number of publishers to communicate with any number of subscribers.
- ✓ ***Language agnostic or protocol agnostic*** - which makes it straightforward and fast to integrate Pub/Sub into technical stack.
- ✓ ***Asynchronous and event driven communication*** - that's ideal for real-time, low-latency apps.
- ✓ ***Separation of concerns***
- ✓ ***Improved testability and scalability***

Disadvantages

- ✓ The inflexibility od data send by the publisher
- ✓ Instability of deliver

Adapter Design Pattern

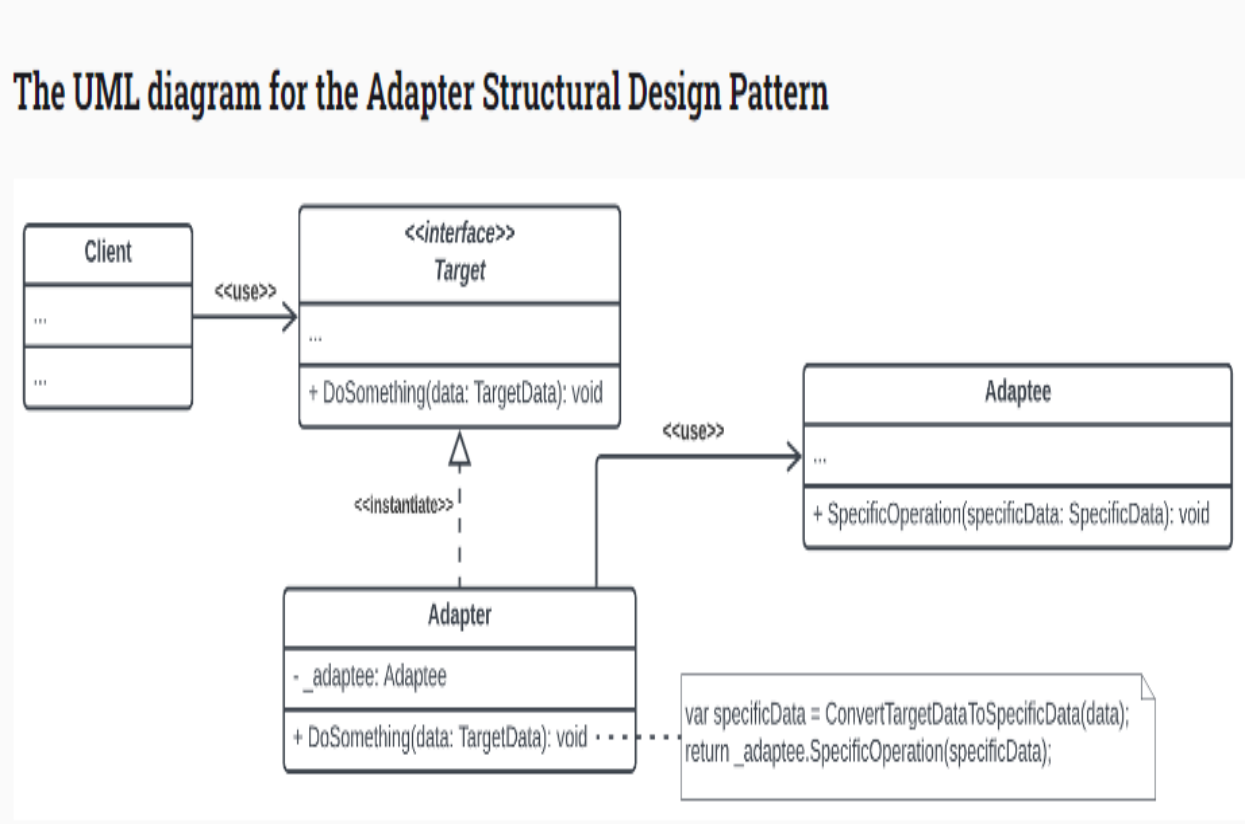
Adapter Pattern - Don't replace an old item, use an adapter instead

- ✓ **Adapter** is a Structural Design Pattern that allows incompatible interfaces between classes to work together without modifying their source code. It acts as a bridge between two interfaces, making them compatible so that they can collaborate and interact seamlessly.
- ✓ This pattern is useful when integrating existing or third-party code into an application without modifying the existing codebase.

- ✓ Additionally, it promotes code reusability by allowing objects to work together even if they were not designed to do so initially. Of course, the interfaces should refer to similar concepts; otherwise, it is almost impossible to "adapt" them.

Adapter Design Pattern Work

The implementation of an **Adapter** class uses the ***object composition principle***: the adapter implements the interface of one object and wraps the other one.



- ✓ The **Client** is a class that contains some existing business logic for the application.
- ✓ The **Target** interface describes a protocol that the **Client** code expects to work with.
- ✓ The **Adaptee** is some useful class (usually 3rd-party or legacy) that has the functionality the **Client** wants to use but is incompatible with the **Client**'s interface. In other words, the **Adaptee** is the class that needs to be adapted.
- ✓ The **Adapter** class implements the **Target** interface but internally uses an instance of the **Adaptee** to make its functionality available to the **Client**.

Advantages

- ✓ **Separation of Concern –** It can separate the interface or data conversion code from the main business logic part of the code.
- ✓ **Independence of Code -** It can implement and use the various adapters without breaking the existing client or main code.

Disadvantages

- ✓ To write a lot of code and it can decrease the efficiency. Sometimes it will be simpler to just change the code of a particular interface.

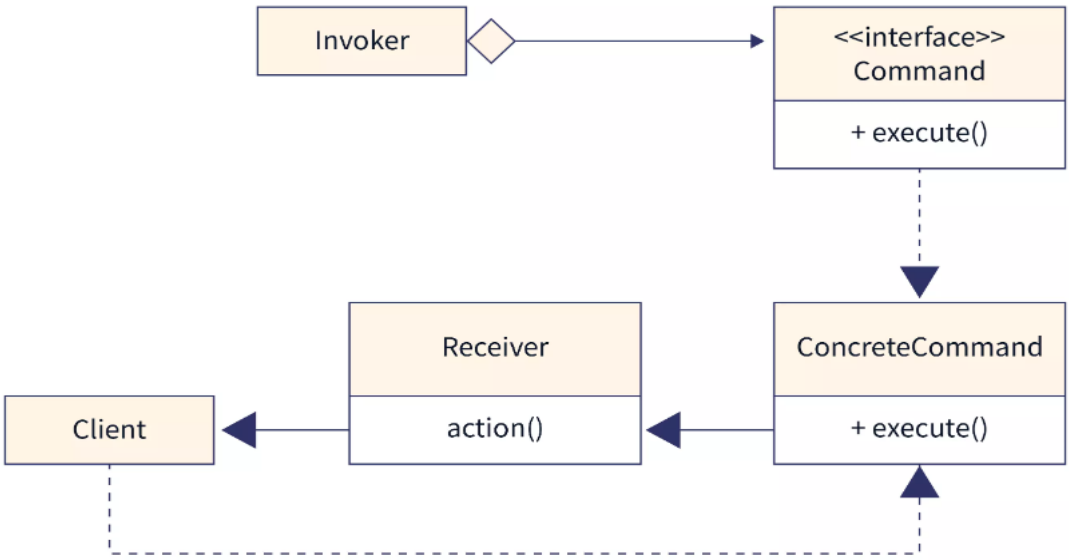
Command Design Pattern

The Command design pattern is a *behavioral design* pattern that ***provides a way to encapsulate a request as an object,*** thereby allowing clients to parameterize clients with different requests, queue or log requests, and support undoable operations.

Ideas of Command Design Pattern

- ✓ The main idea behind the Command pattern is to decouple the sender of a request from the receiver of that request.
- ✓ It allows you to encapsulate a request as an object, which can be passed to invokers and executed at a later time.
- ✓ The invoker is responsible for executing the command, and the receiver is the object that performs the actual work when the command is executed.
- ✓ *Command* establishes unidirectional connections between senders and receivers.

Main components of Command Design Pattern



- ✓ ***Command*** - This is an interface or an abstract class that defines the common methods for executing a command, typically including an execute() method.
- ✓ ***ConcreteCommand*** - This class implements the Command interface and represents a specific command along with its parameters. It encapsulates the receiver object and binds the receiver with the action to be performed.
- ✓ ***Receiver*** - This class defines the operations that need to be performed when a command is executed. It knows how to carry out the request.

- ✓ ***Invoker*** - This class requests the command to carry out the action. It contains a reference to the command and can invoke the command when needed.
- ✓ ***Client*** - This is the class or component that creates the command objects, sets the receiver for the command, and assigns the command to the invoker.

Advantages

- ✓ **Loose Coupling** - Command design pattern loosely couples the object that invokes an operation and the object that performs the operation.
- ✓ **Undo/Redo Operations** - Command design pattern provides the ability to undo and redo a command. A separate stack can be maintained to keep track of the executed command so that it can be used to **undo/redo an operation**.
- ✓ **Extensibility** Adding a new command is easy. A concrete command class must be written and passed to the invoker. Command design pattern leverages the **Open/Closed principle**.

Disadvantages

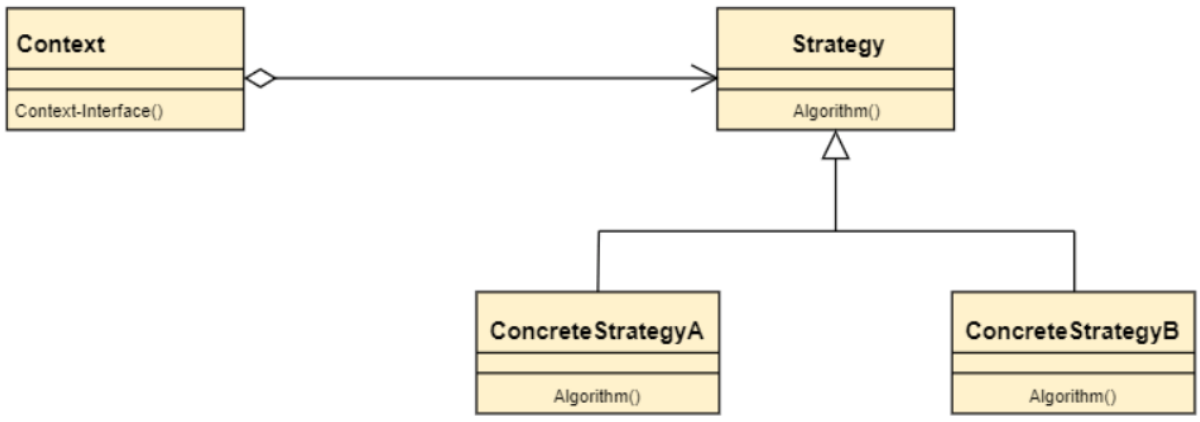
- ✓ **Growing Classes** Command design pattern insists on creating a separate class for each command is too much and may cause **difficulty maintaining** the codebase.

Strategy Design Pattern

The Strategy Design Pattern is a behavioral design pattern. It allows to dynamically change the behavior of an object by encapsulating it into different strategies.

- ✓ Strategy patterns can describe how to assemble classes, arrange a group of classes, and create objects.
- ✓ Strategy design patterns is that a variable **program and object behavior** can also be realized during software runtime.

Components of the Strategy Design Pattern



- ✓ **Context -** The object that will delegate its behavior to one of the contained strategies. The context maintains a reference to a strategy object and interacts with it through a common interface.
- ✓ **Strategy Interface -** The interface that defines the behavior for all strategies. The strategies implement this interface to provide their unique implementation of the behavior.
- ✓ **Concrete Strategies -** The classes that implement the Strategy Interface. Each strategy encapsulates a specific behavior that the context can switch to at runtime.

Strategy Design Pattern Works

- ✓ The Strategy Design Pattern works by separating the behavior of an object from the object itself.
- ✓ The behavior is encapsulated into different strategies, each with its own implementation of the behavior.
- ✓ The context maintains a reference to a strategy object and interacts with it through a common interface.
- ✓ At runtime, the context can swap the current strategy with another one, effectively changing the object's behavior.

Advantages

- ✓ Algorithms can be swapped effectively at runtime.
- ✓ The implementation of the algorithm is decoupled from the code that uses the algorithm.
- ✓ It can introduce new strategies without having to modify the original code that calls the algorithm.

Disadvantages

- ✓ Over-engineering when there are only a few algorithms.
- ✓ The clients need to consider the differences between the strategies when calling the algorithm.

Observer Design Pattern

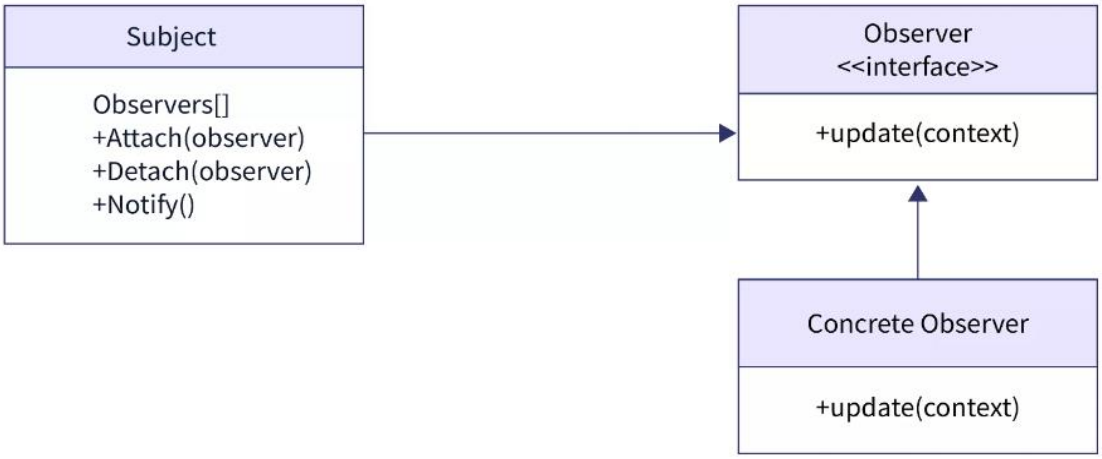
The Observer Pattern maintains a one-to-many relationship among objects, ensuring that when the state of one object is changed, all of its dependent objects are simultaneously informed and updated. This design pattern is also referred to as Dependents.

- ✓ Observer design pattern falls under the category of behavioral design patterns.
- ✓ A subject and observer(many) exist in a one-to-many relationship. The observers do not have any access to data, so they are dependent on the subject to feed them with information.

Components of Observer Design Pattern

- ✓ **observers -** an array of observers that will get notified whenever a specific event occurs
- ✓ **subscribe() -** a method in order to add observers to the observers list
- ✓ **unsubscribe() -** a method in order to remove observers from the observers list
- ✓ **notify() -** a method to notify all observers whenever a specific event occurs

Observer Design Patterns work



- ✓ The ***subject delivers*** events that are intriguing to the observers. These events occur due to changes in the state of the subject or the execution of certain behaviors.
- ✓ Subjects have a ***registration architecture*** that enables new observers to join and existing observers to withdraw from the list.
- ✓ Whenever a new event occurs, the subject iterates through the list of observers, calling the notify method provided in the 'observer' interface.
- ✓ The ***notification interface*** is declared by the Observer interface. It usually includes an updating method.
- ✓ ***Concrete Observers*** do certain activities in response to alerts sent by the Subject.

Advantages

- ✓ This design pattern allows information or data transfer to multiple objects without any change in the observer or subject classes.
- ✓ It adheres to the loose coupling concept among objects that communicate with each other.
- ✓ This design pattern follows the Open/Closed Principle, which says that entities should be open for extension but closed for modification. Here the observers can easily be added or removed anytime without any change in the code.

Disadvantages

- ✓ The Observer pattern can increase complexity and potentially cause efficiency issues if it's not executed properly.

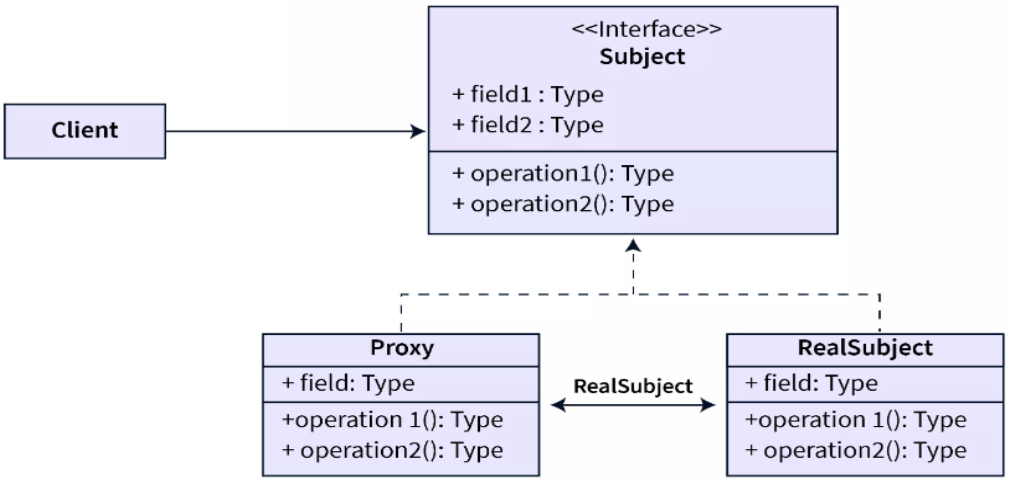
- ✓ The fundamental shortcoming of this design pattern is that the subscribers/observers are updated in a random sequence.

Proxy Design Pattern

Proxy Design Pattern is a type of structural design pattern which is used when a placeholder or representational object that can work in place of the real object.

The proxy acts as an intermediatory layer between the client and the real object and hence can control the access to the real object, add additional functionality, and even restrict client access. It is also known as **Surrogate Pattern**.

Proxy Design Pattern Work



Subject (Interface)

- ✓ It defines the interface which is used by the client.
- ✓ It is the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
- ✓ It defines the operations that are to be expected by the actual implementations.

RealSubject (Normal Class)

- ✓ It implements the Subject and provides the real implementation of the operations defined in the Subject Interface.
- ✓ It provides the actual functionality in the application by overriding the operations defined by the Subject Interface.
- ✓ It provides some useful business logic.

Proxy (Normal Class)

- ✓ It implements the Subject Interface to disguise itself as a Real Subject's object.
- ✓ It also maintains a reference to the RealSubject to provide actual functionality.

- ✓ It controls access to the RealSubject and may be responsible for its creation and deletion.

Advantages

- ✓ **Security -** Proxy provides an additional layer of protection to the original object from the outside world
- ✓ **Better Performance -** By avoiding the creation or duplication of memory-intensive objects and cache
- ✓ **Reliability –** The proxy can work in place of actual objects even if the actual service is not ready or is not available.

Disadvantages

- ✓ **Complexity -** To write repeated code as the proxy is similar to the actual object. Hence, it increases the code complexity.
- ✓ **Extra Effort -** To update the application, the developer must concurrently update the proxy object alongside the real object.
- ✓ **Ambiguity -** Proxies are meant to act as a substitute for real and heavy objects.

Architectural Styles

- ✓ ***Software architecture*** is the process of defining the high-level structure and organization of a software system. It involves identifying and selecting the right components, deciding how they should interact with each other, and determining how they should be organized to achieve specific goals.
- ✓ The goal of software architecture is to create a system that is maintainable, scalable, and secure, and that can meet the needs of users and organizations over time.

List of Architectural Styles

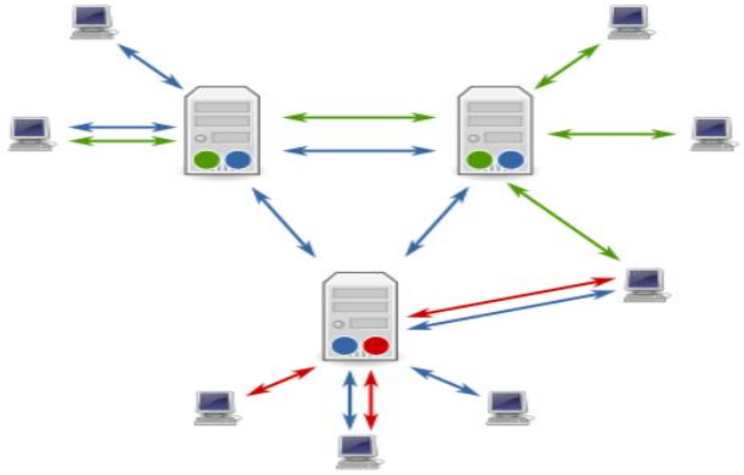
- ✓ Client-Server
- ✓ Layered
- ✓ Pipe and Filter
- ✓ Master-Slave
- ✓ Micro-kernal
- ✓ Microservice
- ✓ Event Driven
- ✓ DDD (Domain Driven Design)
- ✓ Component Based
- ✓ SOA (Service Oriented Architecture)
- ✓ Stream Based

Software architecture Styles

1. Client-server pattern

- ✓ This pattern is the most frequent type used by online programs. In this pattern, one server is connected to at least one client.
- ✓ The server is the central aspect of the database and holds virtually everything needed by the clients, including files, data, and programs.

✓ This pattern is not very complex, but it is also fairly vulnerable to breakdown. All that is necessary for the system to fall apart is for the server to be corrupted.



Each of the servers in this client-server software system has multiple clients.

2. Layered

✓ It's a common way to design complex software systems, and it involves breaking down the system into layers, where each layer is responsible for a specific set of functions. This approach helps to organize code and makes it easier to maintain and modify the system over time.

✓ It consists of 3 main layers - presentation, business logic, and data access.

- **Presentation Layer -** responsible for displaying information to the user and gathering input. This layer includes the user interface and any other components that interact directly with the user.
- **Business Logic Layer -** responsible for implementing the business rules of the application. This layer contains the code that processes and manipulates data, as well as any other application logic.
- **Data Access Layer -** responsible for interacting with the database or other external data sources. This layer contains the code that reads and writes data to and from the database.

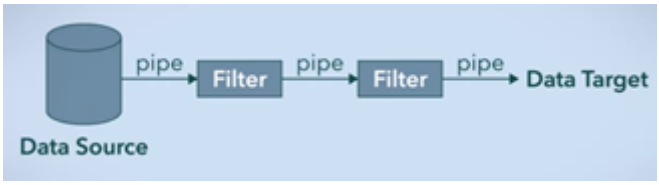
3. Pipe-filter Pattern

✓ Pipe and Filter is another architectural pattern, which has independent entities called **filters** (components) which perform transformations on data and process the input they receive, and **pipes**, which serve as connectors for the stream of data being transformed, each connected to the next component in the pipeline.

Description of the Pattern

✓ The pattern of interaction in the pipe-and-filter pattern is characterized by successive transformations of streams of data.

- ✓ The data flows in one direction. It starts at a data source, arrives at a filter's input port(s) where processing is done at the component, and then, is passed via its output port(s) through a pipe to the next filter, and then eventually ends at the data target.

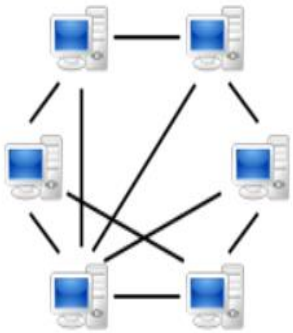


Data transformation in a pipe and filter architecture.

- ✓ A single filter can consume data from, or produce data to, one or more ports. They can also run concurrently and are not dependent. The output of one filter is the input of another, hence, the order is very important.
- ✓ A pipe has a single source for its input and a single target for its output. It preserves the sequence of data items, and it does not alter the data passing through.

4. Peer-to-peer Pattern

- ✓ This software system is comprised of multiple computers, whose users are called peers, that both provide and retrieve data.
- ✓ An oversimplified yet understandable explanation of this pattern is that each computer in the system operates as both a client and a server.



Peer to peer networks are systems of multiple computers communicating with each other rather than with, or through, a server.

User Interface Design UID

User interface (UI) design is the process designers use to build interfaces in software or computerized devices, focusing on looks or style. Designers aim to create interfaces which users find easy to use and pleasurable.

Golden Rules or Mandel's Golden Rules **of UID**

1. Place Users in Control

2. Reduce Users' Memory Load
3. Make the Interface Consistent

Types of User Interface

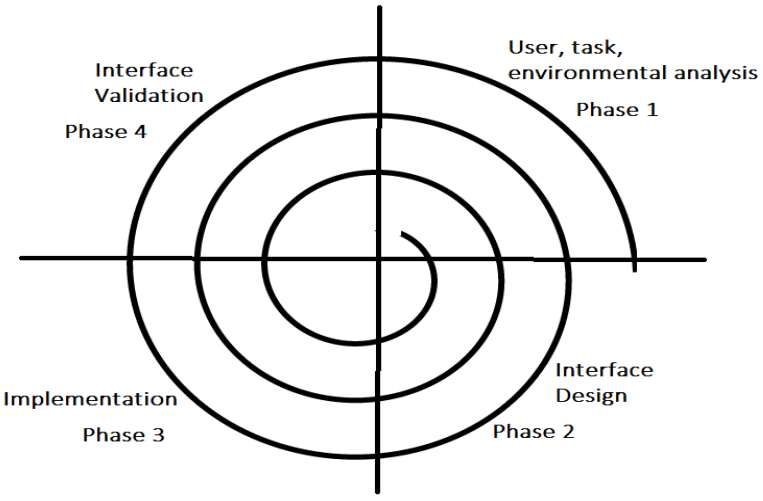
- ✓ **Command Line Interface -** The Command Line Interface provides a command prompt, where the user types the command and feeds it to the system. The user needs to remember the syntax of the command and its use.
- ✓ **Graphical User Interface -** Graphical User Interface provides a simple interactive interface to interact with the system. GUI can be a combination of both hardware and software. Using GUI, the user interprets the software.

Designing User Interface for Users

- ✓ **Graphical user interfaces (GUIs) -** Users interact with visual representations on digital control panels. A computer's desktop is a GUI.
- ✓ **Voice-controlled interfaces (VUIs) -** Users interact with these through their voices. Most smart assistants e.g., Siri on iPhone and Alexa on Amazon devices are VUIs.
- ✓ **Gesture-based interfaces** - Users engage with 3D design spaces through bodily motions: e.g., in virtual reality (VR) games.

User Interface Design Process

- ✓ The analysis and design process of a user interface is iterative and can be represented by a spiral model.
- ✓ The analysis and design process of user interface consists of four framework activities.
    1. User, Task, Environmental Analysis, and Modeling
    2. Interface Design
    3. Interface Implementation
    4. Interface Validation



1. User, Task, Environmental Analysis, and Modeling

- ✓ Initially, the focus is based on the profile of users who will interact with the system, i.e., understanding, skill and knowledge, type of user, etc., based on the user's profile users are made into categories.
- ✓ From each category requirements are gathered. Based on the requirement's developer understand how to develop the interface. Once all the requirements are gathered a detailed analysis is conducted.

2. Interface Design

- ✓ The goal of this phase is to define the set of interface objects and actions i.e., control mechanisms that enable the user to perform desired tasks. Indicate how these control mechanisms affect the system.
- ✓ Specify ***the action sequence of tasks and subtasks,*** also called a ***user scenario.*** Indicate the state of the system when the user performs a particular task. This phase serves as the foundation for the implementation phase.

3. Interface Construction and Implementation

- ✓ The implementation activity begins with the creation of a prototype (model) that enables usage scenarios to be evaluated.
- ✓ As iterative design process continues a User Interface toolkit that allows the creation of windows, menus, device interaction, error messages, commands, and many other elements of an interactive environment can be used for completing the construction of an interface.

4. Interface Validation

- ✓ This phase focuses on testing the interface. The interface should be in such a way that it should be able to perform tasks correctly, and it should be able to handle a variety of tasks.
- ✓ It should achieve all the user's requirements. It should be easy to use and easy to learn. Users should accept the interface as a useful one in their work.