Adapter Design Pattern

The Adapter pattern is a structural design pattern that allows objects with incompatible interfaces to work together. It acts as a bridge between two interfaces, converting one interface into another that a client expects. (A real life example could be a case of card reader which acts as an adapter between memory card and a laptop.)

Use of Adapter Patter

- When an existing class has an incompatible interface but needs to work with another class.
- When integrating third-party libraries or legacy systems that have different interfaces.
- When reusing classes without modifying their source code.

Benefits of Adapter Pattern

- **Reusability**: Allows the use of existing classes without modification.
- **Flexibility**: Enables integration of third-party libraries or legacy code without altering them.
- **Loose Coupling**: Reduces dependency between different components, improving maintainability.

Demerits of Not Using Adapter

- If an adapter is not used, systems with incompatible interfaces cannot communicate.
- Without an adapter, we may need to rewrite the existing code, which increases development effort, risks introducing bugs, and reduces reusability

---

Bridge Design Pattern

The Bridge pattern is a structural design pattern that separates an abstraction from its implementation, allowing them to evolve independently. It helps in reducing tight coupling between components.

Use of Bridge Pattern

- When we need to avoid a permanent binding between an abstraction and its implementation.
- When both abstraction and implementation should be extendable independently.
- When dealing with complex class hierarchies, and you want to simplify them.

Benefits of Bridge Pattern

- **Reduces Coupling**: Abstraction (Remote) and Implementation (Device) evolve independently.
- **Increases Flexibility**: Easy to extend devices or remotes without modifying existing code.
- **Avoids Class Explosion**: Prevents creating separate classes for each combination.

Demerits of Not Using Bridge

- If we do not use the Bridge pattern, the system becomes rigid and difficult to maintain.
- Without the Bridge pattern, extending functionality becomes harder, requiring modifications across multiple classes instead of independent changes

---

Decorator Design Pattern

The Decorator pattern is a structural design pattern that allows behavior to be dynamically added to an individual object without modifying its original code. It provides an alternative to subclassing for extending functionality.

Use of Decorator Pattern

- When we need to add responsibilities to objects at runtime without modifying their code.
- When subclassing is impractical because it leads to an explosion of subclasses.

- When we need flexible and reusable code that follows the Open-Closed Principle (open for extension, closed for modification).

Benefits of Decorator Pattern

- **Flexible & Scalable**: Allows adding new features without modifying existing classes.
- **Prevents Class Explosion**: Avoids unnecessary subclasses by dynamically wrapping objects.
- **Follows Open-Closed Principle**: Enhances objects without altering their code.

Demerits of Not Using Decorator

- If the decorator pattern is not used, code becomes rigid and hard to maintain.
- Without the decorator pattern, adding new functionality requires modifying multiple classes, increasing maintenance effort

---

Facade Design Pattern

The Facade pattern is a structural design pattern that provides a simplified interface to a complex subsystem or a set of interfaces. It hides the complexity of the system and allows the client to interact with the system through a single, unified interface.

Use of Facade Pattern

- When we want to provide a simple interface to a complex subsystem or set of interfaces.
- When we need to decouple a client from complex subsystems.
- When we want to reduce the number of objects that a client needs to interact with.

Benefits of Facade Pattern

- **Simplified Interface**: Hides the complexity of subsystems behind a unified interface, making it easier for clients to interact with.
- **Loose Coupling**: Reduces the dependencies between the client and subsystems, which makes the code easier to maintain.
- **Maintainability**: Changes in subsystems require fewer modifications to the client code.
- **Reduced Complexity**: Avoids the need for clients to understand the internal workings of subsystems.

Demerits of Not Using Façade

- If the Facade pattern is not used, the system becomes complex, and the client has to interact with every subsystem directly. This makes the system harder to maintain and extend.
- Without the Facade pattern, the client is exposed to the complexity of subsystems, and the system becomes more difficult to scale and maintain

---

Proxy Design Pattern

The Proxy pattern is a structural design pattern that provides an object representing another object. It acts as an intermediary, controlling access to the original object. The proxy can be used for various purposes, such as lazy initialization, access control, logging, and monitoring.

Use of Proxy Pattern

- When we need to control access to an object and decide when and how to create or interact with it.
- When we want to add additional functionality to an existing object.
- When an object is expensive to create or load, and you want to defer its creation until needed (Lazy Proxy).
- When you need to control or monitor the interaction with an object.

Benefits of Proxy Pattern

- **Lazy Initialization**: The object is not created until it's actually needed, which can save resources and improve performance.

- **Access Control**: The proxy can be used to restrict access to the real object, e.g., providing a security check or limiting the number of times an object is accessed.
- **Additional Functionality**: The proxy can add additional functionality such as logging, caching, or security checks without modifying the original object.
- **Separation of Concerns**: The proxy allows for separation of the object's logic from its access control, making the system easier to maintain and extend.

Demerits of Not Using Proxy

- If the Proxy pattern is not used, the client might interact directly with the real object. This can lead to inefficient resource usage, security risks, or lack of control over access to the object.
- Without the Proxy pattern, the client doesn't have the opportunity to defer object creation or add additional logic (such as access control or logging) before interacting with the real object.

Chain of Responsibility Design Pattern

The Chain of Responsibility (CoR) pattern is a behavioral design pattern that allows multiple objects (handlers) to process a request in a chain without the sender needing to know which object will handle it. Each handler can either process the request or pass it to the next handler in the chain.

Use of Chain of Responsibility Pattern

- When multiple objects can handle a request but the handler is determined dynamically at runtime.
- When we want to decouple the sender and receiver of a request.
- When we need to execute multiple processing steps in a flexible sequence.
- When handling different levels of access, such as logging, authentication, and request validation.

Benefits of Chain of Responsibility Pattern

- **Loose Coupling**: The client does not need to know which handler will process the request.
- **Flexible Processing Chain**: Easily modify or extend the chain without altering existing classes.
- **Better Maintainability**: Each handler has a single responsibility, following the Single Responsibility Principle (SRP).
- **Improves Scalability**: New handlers can be added without affecting the client or other handlers.

Demerits of Not Using Chain of Responsibility

- The logic is not reusable, since handlers are tightly coupled inside a single class.
- Difficult to maintain when handling multiple conditions.
- Without the Chain of Responsibility, the system is rigid, hard to extend, and tightly coupled, leading to poor maintainability

---

Observer Design Pattern

The Observer Pattern is a behavioral design pattern where an object (subject) maintains a list of dependent observers and notifies them of any state changes. It helps establish a one-to-many relationship between objects, ensuring that dependent objects update automatically when the subject changes.

Use of Observer Pattern

- When multiple objects need to be notified of changes in another object's state.
- When implementing event-driven programming

- When decoupling subjects (publishers) from observers (subscribers) to make the system more flexible.
- Used in stock market systems, weather tracking apps, and social media notifications.

Benefits of Observer Pattern

- **Loose Coupling**: The WeatherStation (subject) does not need to know about specific observer types.
- **Scalability**: Adding new observers (e.g., SmartWatchDisplay) does not require changes to WeatherStation.
- **Automatic Updates**: Observers receive updates automatically without polling.
- **Follows Open-Closed Principle** – New observers can be added without modifying existing code.

Demerits of Not Using Observer

- Difficult to maintain – Adding a new display requires modifying WeatherStation.
- Breaks Open-Closed Principle – Every new observer requires code changes in multiple places.
- Without the Observer Pattern, the system becomes rigid, tightly coupled, and difficult to scale

---

Visitor Design Pattern

The Visitor pattern is a behavioral design pattern that allows you to add new operations to existing object structures without modifying their classes. It separates algorithms from the objects on which they operate, making it easier to extend functionality without modifying existing code.

Use of Visitor Pattern

- When we want to add new operations to a class hierarchy without modifying the existing classes.
- When we need to perform operations on a group of related objects
- When we want to follow the Open-Closed Principle (open for extension but closed for modification).
- When different types of objects require different processing logic but belong to the same structure.

Benefits of Visitor Pattern

- **Open-Closed Principle**: New operations can be added without modifying existing objects.
- **Separation of Concerns**: Business logic is separated from object structure.
- **Extensibility**: Adding new visitors (operations) is easy without affecting the base classes.
- **Better Organization**: Encapsulates related operations in separate visitor classes.

Demerits of Not Using Visitor

- Tightly Coupled Code: The operations are bound to Employee classes.
- Difficult Maintenance: Each operation is hardcoded, making it difficult to extend.
- Without the Visitor pattern, extending functionality requires modifying existing classes, leading to a less maintainable and less scalable system