Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

1. Model-View-Controller (MVC)

Model-View-Controller is an architectural pattern that separates an application into three components: Model for data and logic, View for the user interface, and Controller for handling input and coordinating updates.

Components:

- **Model**: Represents the core data and business logic. It directly manages the data, logic, and rules of the application.
- **View**: The presentation layer that displays data to the user and sends user inputs to the controller.
- **Controller**: Handles user input, interacts with the model, and updates the view accordingly.

Example: In a web application:

- The **Model** interacts with the database to fetch user data.
- The **View** renders the data into HTML.
- The **Controller** processes user requests (e.g., form submissions) and coordinates updates between the Model and View.

Advantages:

- Clear separation of concerns.
- Easier to test individual components.
- Supports multiple views for the same data.

Use Cases:

- Web frameworks like Ruby on Rails, Django, and ASP.NET MVC use this pattern extensively.

2. Singleton

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

How It Works:

- The class itself controls the instantiation process.
- A private constructor restricts direct object creation.
- A static method provides the instance, creating it only if it doesn’t already exist.

Example: A **Database Connection Manager** that ensures only one connection pool is used throughout the application.

Advantages:

- Reduces memory usage by preventing redundant object creation.
- Ensures consistent state across the application.

Disadvantages:

- Can introduce global state, making unit testing harder.

Use Cases:

- Logging systems.
- Configuration settings.

3. Observer

Observer is a behavioral design pattern that lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

How It Works:

- A **Subject** maintains a list of observers and notifies them of state changes.
- **Observers** register themselves with the Subject to receive updates.

Example: A **news app**:

- The Subject is the news publisher.
- Observers are users subscribed to the news feed. Whenever a new article is published, subscribers are notified.

Advantages:

- Promotes loose coupling between components.
- Dynamic communication between objects.

Disadvantages:

- Can lead to performance overhead if there are too many observers.

Use Cases:

- Event systems like JavaScript's event listeners.
- Real-time systems like stock price tickers.

4. Factory Method

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

How It Works:

- The base class defines a method for object creation.
- Subclasses override this method to specify the type of object being instantiated.

Example: A **Vehicle Factory**:

- The base class provides a method `createVehicle`.
- Subclasses like `CarFactory` or `BikeFactory` override the method to return specific vehicle types.

Advantages:

- Promotes code reuse and flexibility.
- Eliminates tight coupling between code that uses objects and their concrete classes.

Disadvantages:

- Adding new object types requires creating new subclasses.

Use Cases:

- Dependency injection frameworks.
- GUI frameworks like Swing or Qt.

5. Strategy

Strategy is a behavioral design pattern that enables selecting an algorithm's behavior at runtime by encapsulating related algorithms into separate, interchangeable classes.

How It Works:

- A **Context** object delegates the execution of a behavior to a **Strategy** object.
- Different strategies implement the same interface and can be swapped at runtime.

Example: Payment processing system:

- Strategies include `CreditCardPayment`, `PayPalPayment`, and `BankTransferPayment`.
- The Context (e.g., an e-commerce checkout) dynamically selects the appropriate strategy based on user preference.

Advantages:

- Simplifies code by isolating algorithms.
- Makes the system more flexible and extensible.

Disadvantages:

- Increases the number of objects in the system.

Use Cases:

- Sorting algorithms (e.g., quicksort, mergesort).
- Game AI behavior (e.g., aggressive or defensive strategies).