Design Patterns

Creational Design Patterns

➢ Creational design patterns provide various object creations, which increase flexibility and reusability of existing code.

➢ Some of the creational design patterns are :

1. Factory method
2. Abstract factory
3. Builder
4. Singleton
5. Prototype

1. Factory method

- Factory method pattern provides an interface for creating objects in a superclass and allows subclasses to alter the type of objects that will be created.
- The interface encapsulates the object creation and lets subclasses choose what objects to create.
- First, We will define an interface that represents common behavior and create classes implementing the interface.
- Next, We create a factory class which is responsible for creating instances based on the input.
- Finally, In main we can create different items(Vehicles, Flowers, etc) without instantiation.
- We can add new items in future without modifying the existing code as we encapsulated the object creation process.
- *Real world examples :*
    - Payment Gateway Integration : Payment gateway factory can be used to create instances of various payment gateways classes. Integration of gateways is easy and can be switched as needed.
    - Logging : Used to create different types of loggers (console, file, database, etc). Can switch between these without affecting the rest of the app.

2. Abstract Factory

- Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- First, We will define abstract classes (items) and interfaces(factories).
- Next, We will create types of items (Laptops) and their corresponding items (Accessories) extending abstract classes.
- Next, We will create concrete Factory classes implementing interfaces.
- Finally, In main we can create different items and their corresponding items by using abstract factories.
- *Real world examples :*
    - GUI : Abstract factory pattern used to create families of related UI components for different platforms (Windows, Mac, Linux) or themes by creating concrete factories.
    - Database : Creating abstract data access objects that are not tied to vendors. Concrete factory implementations can provide classes to interact with (MYSQL, Oracle, etc).

3. **Builder**
    - Builder pattern separates the construction of a complex object from its representation and allows the same construction process to create different representations.
    - The Builder can be enhanced by using Director class as it encapsulates the building steps and guides the construction process. It is responsible for actual construction of complex objects.
    - First, we create a class (complex object) making variables and constructor private and declaring getters. It has a nested builder class with setter methods for each property.
    - Next, we create a build method in builder class which creates complex object instances with specified properties.
    - Finally, In main by using builder we can create objects step by step by specifying only the properties we want while leaving the others with default values if not explicitly defined.
    - *Real world examples :*
        - Meal Ordering : Create complex Meal objects with different options for starters, main courses, etc. The builder allows clients to customize their orders.
        - HTML, XML Doc Generation : While dynamically creating, builder can be used to add elements and attributes in a structured way.

4. **Singleton**
    - Singleton ensures that the class has only one instance and provides a global point of access to that instance.
    - To implement this, Firstly we create a class that has a private static final instance variable and private constructor for preventing external instantiation.
    - We create a public static method to provide the access of instance.
    - "Private static final instance variable" holds the single instance. It is made private to restrict direct access, static to be shared among all instances, and final to maintain the same instance throughout the application's lifecycle.
    - *Real world examples :*
        - Configuration setting : It is used to hold configuration parameters such as database con, API keys, etc.
        - Device Drivers and Hardware access : To represent a single access point to hardware, preventing multiple conflicting access attempts.

5. **Prototype**
    - Prototype pattern allows to create new objects by copying existing objects, known as prototypes. It avoids the cost and time of complex object creation from scratch.
    - It has two clone methods :
        - Shallow Clone - creates new objects and copies the *reference* of original object fields.
        - Deep Clone - creates new objects and copies the *contents* of original object fields.
    - Firstly, we create a class implementing a *cloneable* interface to enable cloning.
    - The clone() method uses the default implementation of Object's clone() method, which performs a shallow copy by creating a new object and copying the references of the original object's fields. The deepCopy() method creates a new object and a new copy of the data.
    - *Real world examples :*
        - Game development : Create game objects/ prototypes, developers can clone and adjust their characteristics with different behavior and attributes.
        - Product Configs in E-commerce : Offer customized products. Users can start with a base product prototype and modify it by adding or removing features, colors, or accessories.

Structural Design Patterns

- ➢ Structural design patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.
- ➢ Some of the structural design patterns are :
    1. Adapter
    2. Bridge
    3. Composite
    4. Decorator

1. Adapter

- This allows incompatible interfaces to work together by creating an adapter class that acts as a bridge between them.
- To implement this, we consider a class with legacy code with an incompatible interface.
- Next, We consider an interface that represents new use cases.
- Next, We create an Adapter class that implements the interface and contains an instance of the legacy code.
- Finally, We create instances of legacy codes and then use adapters to make work with new methods.
- *Real world examples :*
    - Printer drivers : drivers act as adapters between different OS and printer models. They convert printer commands and data suitable to OS and printer for easy printing.
    - Third-party API integration : Adapter used to wrap APIs and provide a consistent interface that matches the application architecture.

2. Bridge

- This decouples the two hierarchies and an abstraction from its implementation, allowing both to vary independently. Provides interchangeability and flexibility.
- First, we create interfaces for each hierarchy. By implementing these interfaces we create concrete classes.
- We create instances of different items(remotes) and link them with different items (devices).
- *Real world examples :*
    - Payment gateways : The gateway interface serves as abstraction, while different processors (e.g., UPI, Netbanking) act as implementations. Applications can integrate with different processors while maintaining a unified payment gateway interface.
    - Remote Controls and Devices : Various remotes as abstraction, different devices as implementations. This separation allows the remote controls to operate various devices without altering their behavior or the devices' internals.

3. **Composite**

- This allows us to treat individual objects and compositions of objects uniformly.
- First, we will create component interfaces(File comp), then leaf classes(file) and composite classes(folder) implementing component interfaces.
- We will add leaf items / sub composite items to the component.
- *Real world examples :*
    - File systems : It consists of files (leaves) and directories (composites) organized in a hierarchical manner. We can treat files and directories uniformly, enabling easy traversal and manipulation of the file system.
    - Computer networks : Networks can be organized into subnets, which can further contain subnets or devices. This can be used to represent these hierarchical network structures, enabling network administrators to manage and configure networks effectively.

4. **Decorator**

- This allows behavior to be added to individual objects without affecting the behavior of other objects from the same class.
- First, we will create a component interface and concrete components implementing the component interface.
- Next, we will create the decorator abstract class implementing interface for adding behavior and concrete decorators for different behaviors.
- Real world examples :
    - Java Swing GUI Library : Swing components, such as buttons, labels, panels, etc., can be decorated with additional functionalities like borders, backgrounds, and tooltips.
    - Image Processing Filters: Apps can utilize decorators to apply various filters and transformations to images, such as blurring, sharpening, and cropping, while maintaining the original image data.