# Design Patterns in Java

**Table of Contents**


- Introduction
- What Are Design Patterns?
- Why Use Design Patterns?
- Categories of Design Patterns
    - Creational Patterns
    - Structural Patterns
    - Behavioral Patterns
- Detailed Overview of Key Design Patterns
    - Creational Patterns Detailed
        - Factory Method Pattern
        - Abstract Factory Pattern
        - Singleton Pattern
        - Builder Pattern
        - Prototype Pattern
    - Structural Patterns Detailed
        - Adapter Pattern
        - Bridge Pattern
        - Composite Pattern
        - Decorator Pattern
        - Facade Pattern
        - Flyweight Pattern
        - Proxy Pattern
    - Behavioral Patterns Detailed
        - Chain of Responsibility Pattern
        - Command Pattern
        - Iterator Pattern
        - Mediator Pattern
        - Memento Pattern
        - Observer Pattern
        - State Pattern
        - Strategy Pattern
        - Template Method Pattern
        - Visitor Pattern
- Conclusion
- References


## Introduction

In software engineering, **design patterns** are standardized solutions to common design problems. They capture best practices evolved by experienced developers over many years and can be reused in various

contexts. In Java, design patterns help create robust, scalable, and maintainable code by promoting principles such as loose coupling, encapsulation, and reusability.

---

## What Are Design Patterns?

**Design patterns** are general, reusable solutions to common problems in software design. They are not finished code that can be directly inserted into a program; rather, they provide a template or blueprint for solving issues that arise frequently during development.

- **Reusable Solutions:** Patterns encapsulate proven solutions to recurring design problems.
- **Language Independence:** Although many examples are given in Java, the concepts are applicable across various programming languages.
- **Communication:** They provide a common vocabulary for developers, making it easier to discuss design ideas.
- **Best Practices:** Using design patterns encourages best practices such as separation of concerns, maintainability, and scalability.

ref:- TutorialsPoint Design Patterns Tutorial and Refactoring.Guru's Design Patterns blog.

---

## Why Use Design Patterns?

Design patterns are used to:

- **Solve Common Problems:** They provide time-tested solutions to recurring design challenges. Also, they help in avoiding common pitfalls and mistakes.
- **Improve Code Quality:** By promoting principles like loose coupling and high cohesion, patterns help in writing cleaner, more maintainable code.
- **Facilitate Communication:** A shared vocabulary (e.g., "Singleton" or "Factory Method") makes it easier for developers to understand and discuss design decisions.
- **Enhance Flexibility and Reusability:** Patterns help in creating modular designs that can be extended or modified with minimal impact on existing code.
- **Reduce Complexity:** They simplify complex designs by breaking them down into smaller, more manageable components.

ref:- GeeksforGeeks Java Design Patterns Tutorial.

---

## Categories of Design Patterns

Design patterns in Java are commonly grouped into three main categories:

### Creational Patterns

These patterns deal with object creation mechanisms. They provide ways to create objects while hiding the creation logic, rather than instantiating objects directly using constructors. Common creational patterns:

- **Factory Method Pattern**
- **Abstract Factory Pattern**

- **Singleton Pattern**
- **Builder Pattern**
- **Prototype Pattern**

### Structural Patterns

Structural patterns focus on how classes and objects are composed to form larger structures. They help ensure that if one part of a system changes, the entire system doesn't need to do the same. Key structural patterns:

- **Adapter Pattern**
- **Bridge Pattern**
- **Composite Pattern**
- **Decorator Pattern**
- **Facade Pattern**
- **Flyweight Pattern**
- **Proxy Pattern**

### Behavioral Patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. They help manage complex control flows and interactions between objects. Important behavioral patterns:

- **Chain of Responsibility Pattern**
- **Command Pattern**
- **Iterator Pattern**
- **Mediator Pattern**
- **Memento Pattern**
- **Observer Pattern**
- **State Pattern**
- **Strategy Pattern**
- **Template Method Pattern**
- **Visitor Pattern**

---

Below is the detailed overview of some of the key design patterns

## Creational Patterns Detailed

#### Factory Method Pattern

Here, we define an interface for creating an object, but let subclasses decide which class to instantiate. This pattern lets a class defer instantiation to subclasses. It means that, instead of directly creating an object, we call a factory method that creates the object. This pattern is useful when a class can't anticipate the class of objects it must create. loose coupling means that the classes are independent of each other. This pattern promotes loose coupling by encapsulating object creation logic in a separate class.

**Description:**
Instead of calling a constructor directly to create an object, a factory method is used. This method

encapsulates the object creation process and allows for more flexible and extensible code.

**Example Scenario:**
Imagine you need to create different types of notification objects (e.g., Email, SMS). Instead of using direct instantiation (`new EmailNotification()`), you can use a factory method that returns the appropriate type based on input.

**Advantages:**

- Encapsulates object creation logic.
- Promotes loose coupling.
- Enhances flexibility and scalability.

**Drawbacks:**

- Can introduce extra layers of abstraction.
- May complicate the code if overused.

**Sample Code:**

```
public interface Shape {
   void draw();
}

public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Rectangle::draw() method.");
   }
}

public class Square implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Square::draw() method.");
   }
}

public class Circle implements Shape {

   @Override
   public void draw() {
      System.out.println("Inside Circle::draw() method.");
   }
}

public class ShapeFactory {

   //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
```

```
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }

      return null;
   }
}

public class FactoryPatternDemo {

   public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");

      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");

      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");

      //call draw method of square
      shape3.draw();
   }
}
```

**output:**

```
Inside Circle::draw() method.
Inside Rectangle::draw() method.
Inside Square::draw() method.
```

**When to Use:**

- When a class cannot anticipate the class of objects it must create.
- To centralize object creation logic for easy maintenance.

ref:- TutorialsPoint's Factory Pattern explanation.

---

**Abstract Factory Pattern**

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

This pattern is a step beyond the Factory Method. It involves multiple factory methods grouped under a single interface. It's used when the system needs to be independent of how its objects are created, composed, and represented.

**Advantages:**

- Ensures consistency among products.
- Supports the addition of new product families without changing existing code.

**Drawbacks:**

- Can become complex as the number of products increases.

**Example:**
i tried following example to understand the concept of abstract factory pattern basically, we have two types of factories, one for windows and one for linux. Each factory creates a set of products (e.g., buttons, text fields) that are designed to work together. this isolates the creation of product objects from the client code, allowing the same client code to work with different types of products.

**Sample Code:**

```
public interface Button {
    void paint();
}

public interface TextField {
    void render();
}

public class WindowsButton implements Button {
    @Override
    public void paint() {
        System.out.println("Rendering a button in Windows style");
    }
}

public class WindowsTextField implements TextField {
    @Override
    public void render() {
        System.out.println("Rendering a text field in Windows style");
    }
```

```
}

public class LinuxButton implements Button {
    @Override
    public void paint() {
        System.out.println("Rendering a button in Linux style");
    }
}

public class LinuxTextField implements TextField {
    @Override
    public void render() {
        System.out.println("Rendering a text field in Linux style");
    }
}

public interface GUIFactory {
    Button createButton();
    TextField createTextField();
}

public class WindowsFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new WindowsButton();
    }
    @Override
    public TextField createTextField() {
        return new WindowsTextField();
    }
}

public class LinuxFactory implements GUIFactory {
    @Override
    public Button createButton() {
        return new LinuxButton();
    }
    @Override
    public TextField createTextField() {
        return new LinuxTextField();
    }
}
```

**When to Use:**

- When a system must be configured with one of multiple families of products.
- When product objects are designed to work together and you need to enforce this constraint.

ref:- Refactoring.Guru's Abstract Factory Pattern.

---

**Singleton Pattern**

Ensure a class has only one instance and provide a global point of access to it.

The Singleton pattern restricts the instantiation of a class to one "single" instance. This is useful when exactly one object is needed to coordinate actions across the system (e.g., a configuration manager or connection pool).

**Advantages:**

- Controlled access to a single instance.
- Reduces namespace clutter.
- Can be lazily initialized to optimize performance.

**Drawbacks:**

- Can be overused; often considered an anti-pattern if it hides dependencies.
- Difficult to unit test because of its global state.

**Example Code:** code reference:- GFG

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    public void showMessage() {
        System.out.println("Singleton instance says hello!");
    }
}
```

**When to Use:**

- When exactly one instance of a class is required to coordinate actions.
- When system-wide configurations or resources must be centralized.

ref:- Refactoring.Guru Singleton Pattern.

---

**Builder Pattern**

Separate the construction of a complex object from its representation, so that the same construction process can create different representations.

The Builder pattern is used when an object requires multiple steps for its creation. It allows you to construct objects step by step, offering better control over the construction process. This is especially

useful when dealing with objects that have many optional parameters.

**Advantages:**

- Improves readability and maintainability for complex objects.
- Provides a clear separation between object construction and representation.
- Facilitates the creation of immutable objects.

**Drawbacks:**

- Can add extra complexity to the code.
- Overhead in creating builder classes.

**Example Code:** code reference:- Digital Ocean

```
public class Computer {
    private String CPU;
    private String RAM;

    private String GPU;
    private String storage;

    private Computer(Builder builder) {
        this.CPU = builder.CPU;
        this.RAM = builder.RAM;
        this.GPU = builder.GPU;
        this.storage = builder.storage;
    }

    public static class Builder {
        private String CPU;
        private String RAM;

        private String GPU = "Integrated";
        private String storage = "256GB";

        public Builder(String CPU, String RAM) {
            this.CPU = CPU;
            this.RAM = RAM;
        }

        public Builder setGPU(String GPU) {
            this.GPU = GPU;
            return this;
        }

        public Builder setStorage(String storage) {
            this.storage = storage;
            return this;
        }

        public Computer build() {
            return new Computer(this);
```

```
        }
    }

    @Override
    public String toString() {
        return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", GPU=" + GPU +
", storage=" + storage + "]";
    }
}

// following code should ideally be in main function, but i am writing it
here for showing usage
Computer myComputer = new Computer.Builder("Intel i7", "16GB")
                        .setGPU("NVIDIA GTX 1660")
                        .setStorage("512GB")
                        .build();
System.out.println(myComputer);
```

**When to Use:**

- When the construction process of an object is complex.
- When you need to create different representations of an object using the same construction process.

For a detailed guide, refer to Refactoring.Guru's Builder Pattern.

---

**Prototype Pattern**

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

The Prototype pattern is used when the process of creating a new object is resource-intensive or complex. Instead of creating a new object from scratch, you clone an existing object. This pattern is particularly useful when the cost of creating a new object is high.

**Advantages:**

- Can reduce the cost of object creation.
- Useful for objects that have a complex initialization process.
- Allows dynamic configuration of new objects at runtime.

**Drawbacks:**

- Cloning can be tricky if objects contain circular references.
- Requires careful handling of deep versus shallow copies.

**Example Code:** following code is a simple example of prototype pattern, where we have a shape interface and a concrete prototype class Circle which implements the shape interface. We have a client class ShapeClient which takes a prototype object and creates a new shape using the prototype.

code reference:- GFG

```java
// Prototype interface
interface Shape {
    Shape clone();  // Make a copy of itself
    void draw();    // Draw the shape
}

// Concrete prototype
class Circle implements Shape {
    private String color;

    // When you create a circle, you give it a color.
    public Circle(String color) {
        this.color = color;
    }

    // This creates a copy of the circle.
    @Override
    public Shape clone() {
        return new Circle(this.color);
    }

    // This is how a circle draws itself.
    @Override
    public void draw() {
        System.out.println("Drawing a " + color + " circle.");
    }
}

// Client code
class ShapeClient {
    private Shape shapePrototype;

    // When you create a client, you give it a prototype (a shape).
    public ShapeClient(Shape shapePrototype) {
        this.shapePrototype = shapePrototype;
    }

    // This method creates a new shape using the prototype.
    public Shape createShape() {
        return shapePrototype.clone();
    }
}

// Main class
public class PrototypeExample {
    public static void main(String[] args) {
        // Create a concrete prototype (a red circle).
        Shape circlePrototype = new Circle("red");

        // Create a client and give it the prototype.
        ShapeClient client = new ShapeClient(circlePrototype);
```

```
        // Use the prototype to create a new shape (a red circle).
        Shape redCircle = client.createShape();

        // Draw the newly created red circle.
        redCircle.draw();
    }
}
```

**When to Use:**

- When object creation is expensive or complex.
- When you need to create an object that is a copy of an existing object with slight modifications.

ref:- Refactoring.Guru's Prototype Pattern.

---

## Structural Patterns Detailed

### Adapter Pattern

Convert the interface of a class into another interface clients expect, allowing incompatible interfaces to work together.

The Adapter pattern wraps an existing class with a new interface so that it can be used in a context where a different interface is expected. It's often used to integrate classes that otherwise couldn't work together because of mismatched interfaces.

**Advantages:**

- Promotes reusability of existing classes.
- Facilitates integration between incompatible interfaces.
- Adheres to the Open/Closed Principle by not modifying existing code.

**Drawbacks:**

- Can add additional layers of abstraction.
- Overuse may lead to complicated code architecture.

**Example Code:** code referece:- Medium Article

```
// Target interface expected by the client
public interface MediaPlayer {
    void play(String audioType, String fileName);
}

// Existing class with a different interface
public class AdvancedMediaPlayer {
    public void playVlc(String fileName) {
        System.out.println("Playing VLC file: " + fileName);
    }
    public void playMp4(String fileName) {
```

```
            System.out.println("Playing MP4 file: " + fileName);
        }
    }

    // Adapter class that implements the target interface
    public class MediaAdapter implements MediaPlayer {
        private AdvancedMediaPlayer advancedMediaPlayer = new
    AdvancedMediaPlayer();

        @Override
        public void play(String audioType, String fileName) {
            if ("vlc".equalsIgnoreCase(audioType)) {
                advancedMediaPlayer.playVlc(fileName);
            } else if ("mp4".equalsIgnoreCase(audioType)) {
                advancedMediaPlayer.playMp4(fileName);
            }
        }
    }

    // Client class
    public class AudioPlayer implements MediaPlayer {
        private MediaAdapter mediaAdapter;

        @Override
        public void play(String audioType, String fileName) {
            if ("mp3".equalsIgnoreCase(audioType)) {
                System.out.println("Playing MP3 file: " + fileName);
            } else if ("vlc".equalsIgnoreCase(audioType) ||
    "mp4".equalsIgnoreCase(audioType)) {
                mediaAdapter = new MediaAdapter();
                mediaAdapter.play(audioType, fileName);
            } else {
                System.out.println("Invalid media. " + audioType + " format not
    supported");
            }
        }
    }
```

**When to Use:**

- When you need to integrate an existing class with a new interface.
- When you want to reuse legacy code without modifying it.

ref:- Refactoring.Guru's Adapter Pattern.

---

**Bridge Pattern**

Decouple an abstraction from its implementation so that the two can vary independently.

The Bridge pattern separates the abstraction (what the object does) from its implementation (how it does it). This is achieved by creating two separate hierarchies—one for the abstraction and one for the

implementation—that can evolve independently.

**Advantages:**

- Reduces coupling between abstraction and implementation.
- Enhances flexibility and extensibility.
- Simplifies maintenance and future development.

**Drawbacks:**

- Can result in an increased number of classes.
- Requires careful design to properly separate concerns.

**Code Example:**
A remote control (abstraction) that works with different types of devices (implementations) such as TVs, radios, etc.

code idea reference:- refactoring guru

**Sample Code Outline:**

```
// Abstraction
public abstract class RemoteControl {
    protected Device device;
    public RemoteControl(Device device) {
        this.device = device;
    }
    public abstract void togglePower();
}

// Implementor
public interface Device {
    void turnOn();
    void turnOff();
}

// Concrete implementation
public class TV implements Device {
    @Override
    public void turnOn() {
        System.out.println("TV is now ON");
    }
    @Override
    public void turnOff() {
        System.out.println("TV is now OFF");
    }
}

// Refined Abstraction
public class AdvancedRemoteControl extends RemoteControl {
    public AdvancedRemoteControl(Device device) {
        super(device);
```

```
    }
    @Override
    public void togglePower() {
        // Example logic: Check current status then switch state
        System.out.println("Toggling power");
        device.turnOn(); // or turnOff() based on state
    }
}
```

**When to Use:**

- When you want to allow the abstraction and its implementation to vary independently.
- When changing the implementation should not affect the abstraction's interface.

ref:- Refactoring.Guru's Bridge Pattern.

---

**Composite Pattern**

Compose objects into tree structures to represent part-whole hierarchies, allowing clients to treat individual objects and compositions uniformly.

The Composite pattern allows you to build complex objects from simpler ones. It lets clients work in a uniform manner with both individual objects and compositions (groups) of objects.

**Advantages:**

- Simplifies client code by treating composite structures uniformly.
- Makes it easier to add new types of components.
- Enhances flexibility in handling hierarchical data.

**Drawbacks:**

- Can make the design overly general.
- Harder to restrict types in the composite structure.

**Example:**
File systems where both files and directories (which contain files) are treated as "components" of the system. this is just the idea, the actual code will be more complex.

**When to Use:**

- When you need to represent part-whole hierarchies.
- When clients should be able to ignore the difference between compositions of objects and individual objects.

ref:- Refactoring.Guru's Composite Pattern.

---

**Decorator Pattern**

Dynamically add responsibilities to an object without modifying its code. This pattern is a flexible alternative to subclassing for extending functionality.

The Decorator pattern involves wrapping an object in a decorator object that adds new behavior before or after delegating to the original object. It is particularly useful when you want to add responsibilities to objects at runtime without affecting other instances of the same class.

**Advantages:**

- Enhances flexibility by adding responsibilities dynamically.
- Supports the Single Responsibility Principle by dividing functionality among classes.
- Can be combined to add multiple behaviors.

**Drawbacks:**

- Can lead to a system with many small classes.
- Increased complexity if overused.

**Example Code:**
ref:- Wikipedia Decorator Pattern example

---

**Facade Pattern**

**Intent:**
Provide a simplified interface to a complex subsystem.

**Description:**
The Facade pattern defines a higher-level interface that makes a subsystem easier to use. It hides the complexity of the subsystem by exposing a simple interface.

**Advantages:**

- Simplifies the usage of a complex system.
- Reduces dependencies between subsystems.
- Improves code readability and maintainability.

**Drawbacks:**

- Can limit the flexibility of the subsystem.
- May hide performance bottlenecks.

**When to Use:**

- When a system is very complex or difficult to understand.
- When you want to provide a simple interface to a complicated subsystem.

ref:- Refactoring.Guru's Facade Pattern.

---

**Flyweight Pattern**

**Intent:**
Reduce the memory footprint by sharing common parts of the object state between multiple objects.

**Description:**
The Flyweight pattern minimizes memory use by sharing as much data as possible with similar objects. It's especially useful when working with a large number of objects that have similar characteristics.

**Advantages:**

- Greatly reduces memory usage.
- Improves performance when managing many objects.

**Drawbacks:**

- Can complicate the design.
- May lead to difficulties in managing shared states.

**When to Use:**

- When you need to handle a large number of similar objects.
- When shared state can be separated from unique object state.

ref:- Refactoring.Guru's Flyweight Pattern.

---

**Proxy Pattern**

**Intent:**
Provide a surrogate or placeholder for another object to control access to it.

**Description:**
The Proxy pattern involves creating a proxy object that acts as an intermediary for requests to the real object. This allows you to add additional behavior such as lazy loading, access control, or logging without changing the original object.

**Advantages:**

- Controls access to the real object.
- Can add security, caching, or logging features transparently.
- Supports lazy initialization.

**Drawbacks:**

- Adds an extra level of indirection.
- Can complicate the code if not implemented properly.

**When to Use:**

- When you need to control access to an object.
- When you want to implement lazy loading or add cross-cutting concerns like logging.

ref:- Refactoring.Guru's Proxy Pattern.

---

## Behavioral Patterns Detailed

### Chain of Responsibility Pattern

**Intent:**
Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request.

**Description:**
The Chain of Responsibility pattern creates a chain of receiver objects for a request. Each object in the chain either handles the request or passes it along to the next object in the chain.

**Advantages:**

- Reduces coupling between sender and receiver.
- Enhances flexibility in assigning responsibilities.
- Simplifies object interconnections.

**Drawbacks:**

- Can lead to unhandled requests if not designed carefully.
- Difficult to debug when the chain is long.

**Example Scenario:**
Exception handling in a series of catch blocks.

**When to Use:**

- When multiple objects can handle a request.
- When you want to decouple senders and receivers.

ref:- Refactoring.Guru's Chain of Responsibility.

---

### Command Pattern

**Intent:**
Encapsulate a request as an object, thereby letting you parameterize clients with queues, requests, and operations.

**Description:**
The Command pattern turns a request into a standalone object containing all information about the request. This allows you to decouple the invoker of an operation from the object that actually performs it.

**Advantages:**

- Supports undoable operations.
- Enables queuing or logging of requests.
- Decouples sender from receiver.

**Drawbacks:**

- Can lead to many command classes.
- Overhead of creating command objects.

**When to Use:**

- When you need to parameterize objects with actions.
- When you require operation queuing, logging, or undo functionality.

ref:- Refactoring.Guru's Command Pattern.

---

**Iterator Pattern**

**Intent:**
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**Description:**
The Iterator pattern separates the traversal of a collection from the collection itself. It allows clients to iterate over complex data structures without needing to know their internal structure.

**Advantages:**

- Simplifies collection traversal.
- Provides a standard iteration interface.
- Hides internal structure.

**Drawbacks:**

- May lead to concurrent modification issues if not handled properly.

**When to Use:**

- When you need a uniform way to traverse a collection.
- When the collection's internal structure should remain hidden.

Learn more from Refactoring.Guru's Iterator Pattern.

---

**Mediator Pattern**

**Intent:**
Reduce direct communication between objects by introducing a mediator object that handles the interactions.

**Description:**
The Mediator pattern centralizes complex communications and control logic between related objects, ensuring that they do not communicate directly. This reduces the dependencies between objects.

**Advantages:**

- Reduces coupling between objects.

- Simplifies object protocols.
- Centralizes complex communication logic.

**Drawbacks:**

- The mediator can become overly complex.
- May centralize too much responsibility.

**When to Use:**

- When many objects interact in complex ways.
- When you want to simplify communication between objects.

ref:- Refactoring.Guru's Mediator Pattern.

---

**Memento Pattern**

**Intent:**
Capture and externalize an object's internal state so that the object can be restored to this state later without violating encapsulation.

**Description:**
The Memento pattern is used to save and restore the state of an object. It is useful for implementing features like undo/redo.

**Advantages:**

- Preserves encapsulation.
- Supports undo mechanisms.
- Separates state saving from object logic.

**Drawbacks:**

- Can consume significant memory if state is large.
- Increased complexity in managing mementos.

**When to Use:**

- When you need to restore an object's state.
- For implementing undo/redo functionality.

ref:- Refactoring.Guru's Memento Pattern.

---

**Observer Pattern**

**Intent:**
Define a one-to-many dependency so that when one object changes state, all its dependents are notified and updated automatically.

**Description:**
The Observer pattern allows an object (subject) to maintain a list of its dependents (observers) and notify them automatically of any state changes, usually by calling one of their methods.

**Advantages:**

- Promotes loose coupling.
- Supports dynamic relationships between objects.
- Useful for event handling systems.

**Drawbacks:**

- Can lead to unexpected updates if not managed correctly.
- May cause performance issues if many observers are involved.

**When to Use:**

- When an object must notify other objects about changes in its state.
- In event-driven systems.

ref:- Refactoring.Guru's Observer Pattern.

---

**State Pattern**

**Intent:**
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

**Description:**
The State pattern encapsulates varying behavior for the same object based on its state. It lets you change an object's behavior when its internal state changes, without resorting to large conditional statements.

**Advantages:**

- Improves maintainability by localizing state-specific behavior.
- Eliminates large conditional statements.
- Promotes clear separation of state and behavior.

**Drawbacks:**

- Can increase the number of classes.
- Complexity in managing state transitions.

**When to Use:**

- When an object must change behavior at runtime based on its state.
- To replace complex conditional logic with state-specific classes.

ref:- Refactoring.Guru's State Pattern.

---

**Strategy Pattern**

**Intent:**
Define a family of algorithms, encapsulate each one, and make them interchangeable so that the algorithm can vary independently from clients that use it.

**Description:**
The Strategy pattern allows you to choose an algorithm's behavior at runtime. It encapsulates different algorithms in separate classes and makes them interchangeable, providing flexibility to change behavior without modifying the client code.

**Advantages:**

- Promotes reusability of algorithms.
- Enables easy switching of algorithms at runtime.
- Adheres to the Open/Closed Principle.

**Drawbacks:**

- Increased number of classes.
- Clients must be aware of the strategies available.

**When to Use:**

- When multiple algorithms are available for a task.
- To switch behavior dynamically.

ref:- Refactoring.Guru's Strategy Pattern.

---

**Template Method Pattern**

**Intent:**
Define the skeleton of an algorithm in a method, deferring some steps to subclasses. This pattern lets subclasses redefine certain steps of an algorithm without changing its structure.

**Description:**
The Template Method pattern provides a base class that implements the overall algorithm structure while allowing subclasses to override specific steps. This ensures a consistent algorithm structure while enabling customization.

**Advantages:**

- Ensures a consistent structure for algorithms.
- Encourages code reuse.
- Simplifies maintenance by centralizing common behavior.

**Drawbacks:**

- Can limit flexibility if subclasses are forced to follow the predefined structure.
- Subclasses may only partially change the algorithm.

**When to Use:**

- When you have a common algorithm with varying steps.
- To enforce an algorithm's structure while allowing customization.

ref:- Wikipedia Template Method Pattern (Italian).

---

**Visitor Pattern**

**Intent:**
Represent an operation to be performed on elements of an object structure. The Visitor pattern lets you define a new operation without changing the classes of the elements on which it operates.

**Description:**
The Visitor pattern separates an algorithm from the object structure it operates on by using double dispatch. A visitor class is created to perform operations on elements of an object structure, allowing new operations to be added without modifying the element classes.

**Advantages:**

- Adds new operations without modifying object structure.
- Centralizes related operations in one visitor class.
- Supports the Open/Closed Principle.

**Drawbacks:**

- Adding new element classes can be cumbersome since every visitor must be updated.
- Can break encapsulation if not designed carefully.

**When to Use:**

- When you need to perform operations across a complex object structure.
- When you want to add new behavior without changing the element classes.

ref:- Wikipedia's Visitor Pattern.

---

## Conclusion

Design patterns in Java are essential tools that help solve recurring design problems using proven, best-practice solutions. By understanding and applying these patterns, you can write code that is more modular, scalable, and maintainable. Each pattern serves a specific purpose—whether it is to create objects efficiently (creational patterns), structure complex systems (structural patterns), or manage algorithms and interactions (behavioral patterns).

---

## References


- TutorialsPoint Design Patterns Tutorial
- Refactoring.Guru – Design Patterns in Java
- GeeksforGeeks Java Design Patterns Tutorial
- DigitalOcean Java Design Patterns Example Tutorial

- Tpoint Tech – Java Design Patterns
- Wikipedia – Decorator Pattern
- Wikipedia – Visitor Pattern
- Wikipedia – Template Method Pattern (Italian)