Java Design Patterns Guide

Design patterns are categorized into three main types: Creational, Structural, and Behavioral patterns.

Creational Patterns

These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

1. Singleton Pattern

Purpose: Ensures a class has only one instance and provides global access to it.

When to use: When you need exactly one instance of a class (e.g., database connection, logger, configuration settings).

2. Factory Pattern

Purpose: Creates objects without specifying their exact classes.

When to use: When you need to create objects based on certain conditions or parameters.

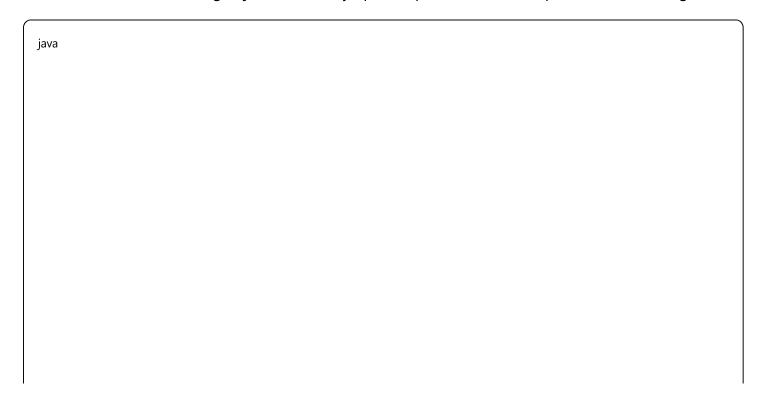
```
java
```

```
abstract class Animal {
  abstract void makeSound();
}
class Dog extends Animal {
  void makeSound() { System.out.println("Woof!"); }
}
class Cat extends Animal {
  void makeSound() { System.out.println("Meow!"); }
}
class AnimalFactory {
  public static Animal createAnimal(String type) {
     switch (type.toLowerCase()) {
       case "dog": return new Dog();
       case "cat": return new Cat();
       default: throw new IllegalArgumentException("Unknown animal type");
    }
```

3. Builder Pattern

Purpose: Constructs complex objects step by step.

When to use: When creating objects with many optional parameters or complex construction logic.



```
public class Car {
  private String engine;
  private String wheels;
  private String color;
  private Car(CarBuilder builder) {
     this.engine = builder.engine;
     this.wheels = builder.wheels;
     this.color = builder.color;
  }
  public static class CarBuilder {
     private String engine;
     private String wheels;
     private String color;
     public CarBuilder setEngine(String engine) {
       this.engine = engine;
       return this;
     }
     public CarBuilder setWheels(String wheels) {
       this.wheels = wheels;
       return this;
     }
     public CarBuilder setColor(String color) {
       this.color = color;
       return this;
     }
     public Car build() {
       return new Car(this);
     }
// Usage
Car car = new Car.CarBuilder()
  .setEngine("V8")
  .setWheels("Sports")
```

.setColor("Red")			
.build();			

Structural Patterns

These patterns deal with object composition and relationships between objects.

4. Adapter Pattern

Purpose: Allows incompatible interfaces to work together.

When to use: When you need to use an existing class with an incompatible interface.

java		

```
// Target interface
interface MediaPlayer {
  void play(String audioType, String fileName);
}
// Adaptee (existing class with different interface)
class AdvancedMediaPlayer {
  void playVIc(String fileName) {
    System.out.println("Playing vlc file: " + fileName);
  }
  void playMp4(String fileName) {
     System.out.println("Playing mp4 file: " + fileName);
  }
}
// Adapter
class MediaAdapter implements MediaPlayer {
  AdvancedMediaPlayer advancedPlayer;
  public MediaAdapter(String audioType) {
     advancedPlayer = new AdvancedMediaPlayer();
  }
  public void play(String audioType, String fileName) {
    if (audioType.equalsIgnoreCase("vlc")) {
       advancedPlayer.playVlc(fileName);
    } else if (audioType.equalsIgnoreCase("mp4")) {
       advancedPlayer.playMp4(fileName);
```

5. Decorator Pattern

Purpose: Adds new functionality to objects without altering their structure.

When to use: When you want to add responsibilities to objects dynamically.

java

```
interface Coffee {
  double getCost();
  String getDescription();
}
class SimpleCoffee implements Coffee {
  public double getCost() { return 2.0; }
  public String getDescription() { return "Simple coffee"; }
}
abstract class CoffeeDecorator implements Coffee {
  protected Coffee decoratedCoffee;
  public CoffeeDecorator(Coffee coffee) {
     this.decoratedCoffee = coffee:
  }
}
class MilkDecorator extends CoffeeDecorator {
  public MilkDecorator(Coffee coffee) { super(coffee); }
  public double getCost() { return decoratedCoffee.getCost() + 0.5; }
  public String getDescription() { return decoratedCoffee.getDescription() + ", milk"; }
}
class SugarDecorator extends CoffeeDecorator {
  public SugarDecorator(Coffee coffee) { super(coffee); }
  public double getCost() { return decoratedCoffee.getCost() + 0.2; }
  public String getDescription() { return decoratedCoffee.getDescription() + ", sugar"; }
}
```

6. Facade Pattern

Purpose: Provides a simplified interface to a complex subsystem.

When to use: When you want to hide the complexity of a system and provide a simple interface.

java

```
class CPU {
  public void freeze() { System.out.println("CPU freezing..."); }
  public void jump(long position) { System.out.println("Jumping to " + position); }
  public void execute() { System.out.println("Executing..."); }
}
class Memory {
  public void load(long position, byte[] data) {
     System.out.println("Loading data to memory at " + position);
  }
}
class HardDrive {
  public byte[] read(long lba, int size) {
     System.out.println("Reading from hard drive");
     return new byte[size];
}
// Facade
class ComputerFacade {
  private CPU cpu;
  private Memory memory;
  private HardDrive hardDrive;
  public ComputerFacade() {
     this.cpu = new CPU();
     this.memory = new Memory();
     this.hardDrive = new HardDrive();
  }
  public void start() {
     cpu.freeze();
     memory.load(0, hardDrive.read(0, 1024));
     cpu.jump(0);
     cpu.execute();
  }
```

Behavioral Patterns

These patterns focus on communication between objects and the assignment of responsibilities.

7. Observer Pattern

Purpose : Defines a one-to-many dependency between objects so that when one object changes state, all dependents are notified.					
When to use : When changes to one object require changing many other objects, and you don't know how many objects need to be changed.					
java					

```
import java.util.*;
interface Observer {
  void update(String message);
}
interface Subject {
  void registerObserver(Observer observer);
  void removeObserver(Observer observer);
  void notifyObservers();
}
class NewsAgency implements Subject {
  private List < Observer > observers;
  private String news;
  public NewsAgency() {
     this.observers = new ArrayList<>();
  }
  public void setNews(String news) {
     this.news = news;
     notifyObservers();
  }
  public void registerObserver(Observer observer) {
     observers.add(observer);
  }
  public void removeObserver(Observer observer) {
     observers.remove(observer);
  }
  public void notifyObservers() {
     for (Observer observer : observers) {
       observer.update(news);
    }
}
class NewsChannel implements Observer {
  private String name;
```

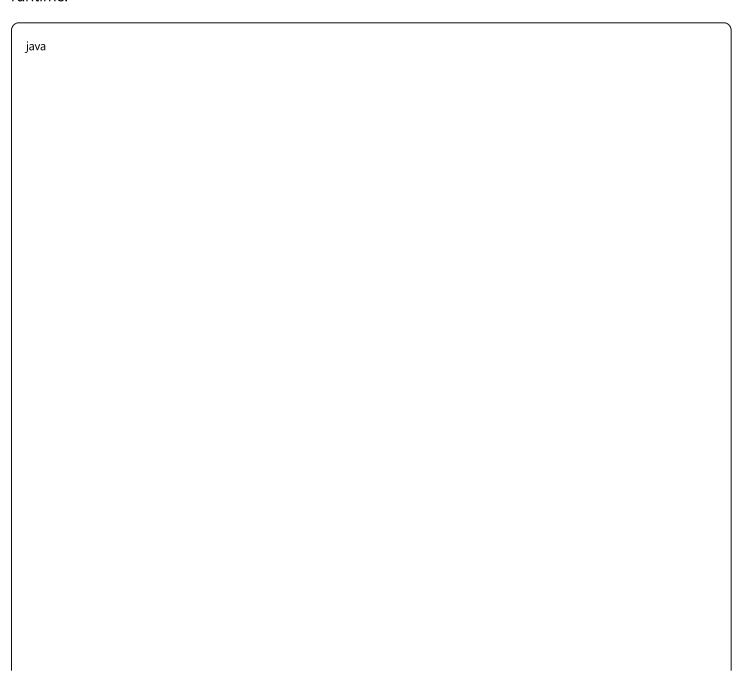
```
public NewsChannel(String name) {
    this.name = name;
}

public void update(String news) {
    System.out.println(name + " received news: " + news);
}
```

8. Strategy Pattern

Purpose: Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

When to use: When you have multiple ways of performing a task and want to choose the algorithm at runtime.



```
interface PaymentStrategy {
  void pay(double amount);
}
class CreditCardPayment implements PaymentStrategy {
  private String cardNumber;
  public CreditCardPayment(String cardNumber) {
    this.cardNumber = cardNumber;
  }
  public void pay(double amount) {
     System.out.println("Paid $" + amount + " using Credit Card: " + cardNumber);
}
class PayPalPayment implements PaymentStrategy {
  private String email;
  public PayPalPayment(String email) {
    this.email = email;
  }
  public void pay(double amount) {
    System.out.println("Paid $" + amount + " using PayPal: " + email);
}
class ShoppingCart {
  private PaymentStrategy paymentStrategy;
  public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
     this.paymentStrategy = paymentStrategy;
  }
  public void checkout(double amount) {
    paymentStrategy.pay(amount);
  }
}
```

9. Command Pattern

Purpose: Encapsulates a request as an object, allowing you to parameterize clients with different					
requests.					
When to use: When you want to queue operations, undo operations, or log operations.					
iava					
java					

```
interface Command {
  void execute();
  void undo();
}
class Light {
  public void turnOn() { System.out.println("Light is ON"); }
  public void turnOff() { System.out.println("Light is OFF"); }
}
class LightOnCommand implements Command {
  private Light light;
  public LightOnCommand(Light light) {
     this.light = light;
  }
  public void execute() { light.turnOn(); }
  public void undo() { light.turnOff(); }
}
class LightOffCommand implements Command {
  private Light light;
  public LightOffCommand(Light light) {
     this.light = light;
  }
  public void execute() { light.turnOff(); }
  public void undo() { light.turnOn(); }
}
class RemoteControl {
  private Command;
  public void setCommand(Command command) {
     this.command = command;
  }
  public void pressButton() {
     command.execute();
  }
```

```
public void pressUndo() {
    command.undo();
}
```

Benefits of Using Design Patterns

- 1. **Reusability**: Patterns provide tested, proven development paradigms
- 2. **Communication**: They provide a common vocabulary for developers
- 3. **Best Practices**: They represent solutions that have evolved over time
- 4. Flexibility: They make code more flexible and easier to modify
- 5. **Maintainability**: Well-structured code is easier to maintain and debug

When to Use Design Patterns

- **Don't force patterns**: Only use them when they solve a real problem
- Understand the problem first: Make sure you understand what problem the pattern solves
- Consider simplicity: Sometimes a simple solution is better than a complex pattern
- Think about future needs: Patterns can help make code more extensible

Common Anti-Patterns to Avoid

- 1. **God Object**: Classes that do too much
- 2. **Spaghetti Code**: Code with poor structure and flow control
- 3. **Copy-Paste Programming**: Duplicating code instead of creating reusable components
- 4. Magic Numbers: Using unexplained constants in code
- 5. **Premature Optimization**: Optimizing before identifying performance bottlenecks