Behavioral Design Patterns

1. Observer Pattern Use Case
   Use Case: *Stock Market Alert System*
   Imagine a system where investors are subscribed to different stocks. Whenever a stock's price changes, all subscribed investors need to be notified. The Observer Pattern is ideal for this use case, as it decouples the stock price change (subject) from the investors (observers).
   Key Points:
   - Subject: Stock
   - Observer: Investor
   - When the stock price changes, all subscribed investors get notified.

```
interface Stock {
   void addObserver(Investor investor);
   void removeObserver(Investor investor);
   void notifyObservers();
}


// Concrete Subject
class AppleStock implements Stock {
   private List<Investor> investors = new ArrayList<>();
   private double price;

   public void setPrice(double newPrice) {
       this.price = newPrice;
       notifyObservers();
   }

   @Override
   public void addObserver(Investor investor) {
       investors.add(investor);
   }

   @Override
   public void removeObserver(Investor investor) {
       investors.remove(investor);
   }

   @Override
   public void notifyObservers() {
       for (Investor investor : investors) {
           investor.update(price);
       }
   }
}

// Observer Interface
interface Investor {
   void update(double price);
}
```

```
// Concrete Observer
class ConcreteInvestor implements Investor {
   private String name;

   public ConcreteInvestor(String name) {
       this.name = name;
   }

   @Override
   public void update(double price) {
       System.out.println("Investor " + name + " notified of price change: "
+ price);
   }
}
```

2. Command Pattern Use Case
   Use Case: *Smart Home Automation System*

   A smart home automation system can be modeled using the Command pattern, where actions like turning on/off lights, air conditioners, or TVs are encapsulated as commands. Commands are queued and executed by the system.

   Key Points:

   - Command Interface: Represents a command that can be executed (turn light on, turn AC off).
   - Receiver: The actual devices like `Light`, `AC`.
   - Invoker: The Smart Home Controller that queues and executes the commands.

```
// Command Interface
interface Command {
   void execute();
}

// Concrete Commands
class LightOnCommand implements Command {
   private Light light;

   public LightOnCommand(Light light) {
       this.light = light;
   }

   @Override
   public void execute() {
       light.turnOn();
```

```java
    }
}

class ACOffCommand implements Command {
    private AC ac;

    public ACOffCommand(AC ac) {
        this.ac = ac;
    }

    @Override
    public void execute() {
        ac.turnOff();
    }
}

// Receiver Classes
class Light {
    public void turnOn() {
        System.out.println("Light turned on.");
    }
}

class AC {
    public void turnOff() {
        System.out.println("AC turned off.");
    }
}

// Invoker Class
class SmartHomeController {
    private List<Command> commandQueue = new ArrayList<>();

    public void addCommand(Command command) {
        commandQueue.add(command);
    }

    public void executeCommands() {
        for (Command command : commandQueue) {
            command.execute();
        }
        commandQueue.clear();
    }
}
```

Creational Design Patterns

1. Factory Pattern Use Case
   Use Case: *Notification Service*
   A notification system needs to create different types of notifications (e.g., email, SMS, push notifications). The Factory pattern will be used to create these notification objects based on the type required.
   Key Points:
   - Factory: NotificationFactory
   - Products: EmailNotification, SMSNotification, PushNotification

```java
// Product Interface
interface Notification {
   void send(String message);
}

// Concrete Products
class EmailNotification implements Notification {
   @Override
   public void send(String message) {
       System.out.println("Sending Email: " + message);
   }
}

class SMSNotification implements Notification {
   @Override
   public void send(String message) {
       System.out.println("Sending SMS: " + message);
   }
}

class PushNotification implements Notification {
   @Override
   public void send(String message) {
       System.out.println("Sending Push Notification: " + message);
   }
}

// Factory Class
class NotificationFactory {
   public static Notification createNotification(String type) {
       switch (type) {
           case "email":
               return new EmailNotification();
           case "sms":
               return new SMSNotification();
           case "push":
               return new PushNotification();
           default:
               throw new IllegalArgumentException("Unknown notification type.");
       }
   }
}
```

2. Builder Pattern Use Case
   Use Case: *Creating a Complex User Profile*

   Suppose we have a `UserProfile` object that has many optional parameters (name, age, address, phone, etc.). The Builder pattern allows for constructing such complex objects in a flexible way.

   Key Points:

   - Builder: UserProfileBuilder
   - Product: UserProfile

```
// Product Class
class UserProfile {
    private String name;
    private int age;
    private String address;
    private String phone;

    private UserProfile(UserProfileBuilder builder) {
        this.name = builder.name;
        this.age = builder.age;
        this.address = builder.address;
        this.phone = builder.phone;
    }

    @Override
    public String toString() {
        return "UserProfile{" +
                "name='" + name + '\'' +
```

```java
                ", age=" + age +
                ", address='" + address + '\'' +
                ", phone='" + phone + '\'' +
                '}';
    }

    // Builder Class
    public static class UserProfileBuilder {
        private String name;
        private int age;
        private String address;
        private String phone;

        public UserProfileBuilder setName(String name) {
            this.name = name;
            return this;
        }

        public UserProfileBuilder setAge(int age) {
            this.age = age;
            return this;
        }

        public UserProfileBuilder setAddress(String address) {
            this.address = address;
            return this;
        }

        public UserProfileBuilder setPhone(String phone) {
            this.phone = phone;
            return this;
        }

        public UserProfile build() {
            return new UserProfile(this);
        }
    }
}
```

Structural Design Patterns

1. Decorator Pattern Use Case
   Use Case: *Coffee Shop Order System*
   In a coffee shop, customers can order basic coffee and add different customizations like milk, sugar, or cream. The Decorator pattern allows for dynamically adding behaviors (customizations) to the coffee object without altering its structure.
   Key Points:
   - Component: Coffee
   - Concrete Component: BasicCoffee
   - Decorator: CoffeeDecorator (adds milk, sugar, etc.)

```java
// Component Interface
interface Coffee {
   String getDescription();
   double cost();
}

// Concrete Component
class BasicCoffee implements Coffee {
   @Override
   public String getDescription() {
       return "Basic Coffee";
   }

   @Override
   public double cost() {
       return 5.0;
   }
}

// Decorator
abstract class CoffeeDecorator implements Coffee {
   protected Coffee coffee;

   public CoffeeDecorator(Coffee coffee) {
       this.coffee = coffee;
   }

   public String getDescription() {
       return coffee.getDescription();
   }
   public double cost() {
       return coffee.cost();
   }
}

// Concrete Decorators
class MilkDecorator extends CoffeeDecorator {
   public MilkDecorator(Coffee coffee) {
       super(coffee);
   }

   @Override
   public String getDescription() {
       return coffee.getDescription() + ", Milk";
   }

   @Override
   public double cost() {
       return coffee.cost() + 1.0;
   }
}

class SugarDecorator extends CoffeeDecorator {
   public SugarDecorator(Coffee coffee) {
       super(coffee);
   }

   @Override
   public String getDescription() {
       return coffee.getDescription() + ", Sugar";
   }

   @Override
   public double cost() {
       return coffee.cost() + 0.5;
   }
}
```

2. Adapter Pattern Use Case
   Use Case: *Payment Gateway Integration*

   Suppose you have an e-commerce platform, and you want to integrate multiple payment gateways (like PayPal and Stripe). The Adapter pattern allows you to wrap these different payment services in a unified interface.

   Key Points:

   - Target Interface: PaymentGateway
   - Adapter: PayPalAdapter, StripeAdapter

```java
// Target Interface
interface PaymentGateway {
   void processPayment(double amount);
}

// Adaptee (Existing Class)
class PayPal {
   public void sendPayment(double amount) {
       System.out.println("Processing payment of " + amount + " via PayPal.");
   }
}

class Stripe {
   public void makePayment(double amount) {
       System.out.println("Processing payment of " + amount + " via Stripe.");
   }
}

// Adapter for PayPal
class PayPalAdapter implements PaymentGateway {
   private PayPal payPal;

   public PayPalAdapter(PayPal payPal) {
       this.payPal = payPal;
   }

   @Override
   public void processPayment(double amount) {
       payPal.sendPayment(amount);
   }
}

// Adapter for Stripe
class StripeAdapter implements PaymentGateway {
   private Stripe stripe;

   public StripeAdapter(Stripe stripe) {
       this.stripe = stripe;
   }

   @Override
   public void processPayment(double amount) {
       stripe.makePayment(amount);
   }
}
```