## Copilot

### Creational Patterns

#### a. Singleton Pattern

Scenario: Database Connection Pool

In a web application, you might want to manage a single instance of a database connection pool to ensure that all database requests are handled efficiently without creating multiple connections.

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

    private DatabaseConnectionPool() {
        // Initialize the connection pool
    }

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

    // Methods to get and release connections
}
```

#### b. Factory Method Pattern

Scenario: Payment Processing System

In an e-commerce application, you might have different payment methods (Credit Card, PayPal, etc.). A factory method can create the appropriate payment processor based on user selection.

```
interface PaymentProcessor {
    void processPayment(double amount);
}

class CreditCardProcessor implements PaymentProcessor {
    public void processPayment(double amount) {
        System.out.println("Processing credit card payment of " + amount);
    }
}

class PayPalProcessor implements PaymentProcessor {
    public void processPayment(double amount) {
        System.out.println("Processing PayPal payment of " + amount);
    }
}

abstract class PaymentFactory {
    public abstract PaymentProcessor createPaymentProcessor();
}

class CreditCardFactory extends PaymentFactory {
    public PaymentProcessor createPaymentProcessor() {
        return new CreditCardProcessor();
    }
}

class PayPalFactory extends PaymentFactory {
    public PaymentProcessor createPaymentProcessor() {
```

```java
        return new PayPalProcessor();
    }
}
```

### c. Abstract Factory Pattern

Scenario: UI Component Creation

In a cross-platform application, you might want to create UI components that look different on Windows and Mac. An abstract factory can help create the appropriate components based on the operating system.

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

class WinButton implements Button {
    public void paint() {
        System.out.println("Rendering a Windows button.");
    }
}

class MacButton implements Button {
    public void paint() {
        System.out.println("Rendering a Mac button.");
    }
}

interface GUIFactory {
    Button createButton();
}

class WinFactory implements GUIFactory {
    public Button createButton() {
        return new WinButton();
    }
}

class MacFactory implements GUIFactory {
    public Button createButton() {
        return new MacButton();
    }
}
```

## Structural Patterns

### a. Adapter Pattern

Scenario: Legacy System Integration

If you have a legacy system that provides data in a different format, you can use an adapter to convert the data into a format that your new system can understand.

```java
class LegacySystem {
    public String getData() {
        return "Data from legacy system";
    }
}

class Adapter implements Target {
    private LegacySystem legacySystem;

    public Adapter(LegacySystem legacySystem) {
        this.legacySystem = legacySystem;
    }

    public void request() {
        String data = legacySystem.getData();
        System.out.println("Adapting: " + data);
    }
}
```

**b. Decorator Pattern**

Scenario: Coffee Shop Order System

In a coffee shop, customers can customize their drinks by adding extras like milk, sugar, or whipped cream. The decorator pattern allows you to add these extras dynamically.

```
Coffee coffee = new SimpleCoffee();
coffee = new MilkDecorator(coffee);
coffee = new SugarDecorator(coffee);
System.out.println(coffee.getDescription() + " costs " + coffee.cost());
```

**c. Composite Pattern**

Scenario: File System Management

In a file system, you can have files and directories. The composite pattern allows you to treat both files and directories uniformly.

```
interface FileSystemComponent {
    void showDetails();
}

class File implements FileSystemComponent {
    private String name;

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

    public void showDetails() {
        System.out.println("File: " + name);
    }
}

class Directory implements FileSystemComponent {
    private String name;
    private List<FileSystemComponent> components = new ArrayList<>();

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

    public void add(FileSystemComponent component) {
        components.add(component);
    }

    public void showDetails() {
        System.out.println("Directory: " + name);
        for (FileSystemComponent component : components) {
            component.showDetails();
        }
    }
}
```

**Behavioral Patterns**

**a. Strategy Pattern**

Scenario: Sorting Algorithms

In a data processing application, you might want to allow users to choose different sorting algorithms (e.g., QuickSort, MergeSort) at runtime.

```
class SortContext {
    private SortStrategy strategy;

    public void setSortStrategy(SortStrategy strategy) {
        this.strategy = strategy;
    }

    public void sort(int[] array) {
        strategy.sort(array);
    }
}
```

**b. Observer Pattern**

Scenario: Weather Monitoring System

In a weather application, you might want to notify multiple displays (e.g., mobile app, web app) whenever the weather data changes.

```
class WeatherData extends Subject {
    private float temperature;

    public void setTemperature(float temperature) {
        this.temperature = temperature;
        notifyObservers("Temperature updated: " + temperature);
    }
}
```

**c. Command Pattern**

Scenario: Remote Control for Home Automation

In a home automation system, you can use the command pattern to encapsulate actions like turning on/off lights, adjusting the thermostat, etc.

```
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();
    }
}

class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
        }
    }
```