SWE 4501: Design Patterns

Mahmudul Islam Mahin (210042140)
Hasin Mahtab Alvee (210042174)

October 24, 2024

What are Design Patterns?

Design patterns are typical solutions to common problems in software design. They're like pre-made blueprints for structuring code, and they help make your code more modular, flexible, and maintainable.

Why are Design Patterns Important?

- Reusability: Instead of reinventing the wheel, you can use tried-and-tested patterns.
- Maintainability: Patterns ensure the code is easy to understand and modify.
- Scalability: They help in managing the complexities of growing software systems.

What are the categories?

- Creational Patterns: Deal with object creation mechanisms, e.g., Singleton, Factory.
- Structural Patterns: Deal with class and object composition, e.g., Adapter, Decorator, Composite.
- Behavioral Patterns: Deal with object collaboration, e.g., Observer, Command, Strategy.

Pattern	Purpose	Key Feature	Use Case Example
Strategy	Encapsulates interchangeable behaviors (algorithms)	Choose an algorithm at runtime	Different sorting algorithms
Template Method	Defines the skeleton of an algorithm and lets subclasses override steps	Fixed algorithm structure with customizable steps	Game structure with different implementations of play
Composite	Compose objects into tree structures to represent part-whole hierarchies	Treat individual objects and compositions uniformly	File system with files and directories
Decorator	Add responsibilities to objects dynamically without changing their structure	Dynamically wraps objects to add behavior	Adding features (like sugar/milk) to a base coffee object
Adapter	Convert one interface into another expected by the client	Translates one interface to another	Adapting legacy code to a new system
Singleton	Ensure a class has only one instance and provides global access	Single instance shared globally	Logger or configuration manager
Factory	Provide an interface for creating objects, but let subclasses alter the type	Factory method creates objects based on parameters	Shape creation (e.g., `Circle`, `Rectangle`)

1 Strategy Pattern

The Strategy Pattern is a behavioral design pattern that enables selecting an algorithm's behavior at runtime. It defines a family of algorithms, encapsulates each one, and makes them interchangeable. The key idea is to separate the algorithm from the object that uses it, allowing you to change the algorithm independently of the client that uses it.

Key Concepts of the Strategy Pattern

- Strategy Interface: This defines a common interface for all the supported algorithms.
- Concrete Strategies: These are the various algorithms that implement the strategy interface.
- Context: The class that uses a strategy object to delegate the algorithm. It interacts with the strategy interface rather than the concrete strategies directly.

Key benefits

- Open/Closed Principle: You can add new payment methods without modifying existing code (just implement a new strategy).
- Easier Maintenance: The payment processing logic is separated from the main system.
- Flexibility: You can switch strategies dynamically at runtime.

2 Template Pattern

The Template Method Pattern is a behavioral design pattern that defines the skeleton of an algorithm in a base class (the template), allowing subclasses to override specific steps of the algorithm without changing its overall structure. This pattern ensures that the invariant parts of the algorithm are handled in the base class, while the variable parts can be customized by subclasses.

Key Concepts of the TemplatePattern

- Abstract Class (Template): This contains the template method that defines the algorithm structure. Some of its methods are abstract or hooks, which subclasses need to implement.
- Concrete Classes: These extend the abstract class and implement the specific details of the algorithm.

When to use

- When you have an algorithm with varying steps, but a common structure.
- When you want to allow subclasses to redefine certain parts of an algorithm without changing the algorithm's structure.

Key benefits

- Code Reuse: The invariant parts of the algorithm are written once in the template method and reused by all subclasses.
- Extensibility: Subclasses can override the necessary steps to provide their own specific behavior without changing the overall algorithm.
- Consistency: The template method ensures that the steps of the algorithm are executed in a consistent order.

Template + Strategy

Consider we are building an application for a pizza store and we need a method to cook a pizza. There are three steps to cook a pizza in the pizza store:

- Prepare pizza dough;
- Add sauce and toppings;
- Bake the pizza.

The first step is a common behavior and the second and third steps are various for different pizza types. Now let's assume we have a lot of new types of pizza in store, so we need many new pizza cookers(subclasses). However, some of the cookers have the same logic in step 2 and some of them have the same logic in step 3. We want these cookers can reuse the same logic through application to reduce code duplication.

```
//src/Abstracts/Pizza

package Abstracts;

public abstract class Pizza {

    public final void cookPizza() {
        prepareDough();
        addSauceAndToppings();
        bakePizza();
    }
    // Step 1 is the same for all pizzas
    private void prepareDough() {
        System.out.println("Preparing pizza dough...");
    }
    // Steps 2 and 3 are customizable
    protected abstract void addSauceAndToppings();

    protected abstract void bakePizza();
}
```

```
//src/Behaviours/BakingStrategy

package Behaviours;

public interface BakingStrategy {
    void bake();
}
```

```
//src/Behaviours/ToppingStrategy

package Behaviours;

public interface ToppingStrategy {
    void addToppings();
}
```

```
//src/Concretes/PepperoniPizza

package Concretes;

import Abstracts.*;
import Behaviours.*;

public class PepperoniPizza extends Pizza {
    private ToppingStrategy toppingStrategy;
    private BakingStrategy bakingStrategy;
    public PepperoniPizza(ToppingStrategy toppingStrategy, BakingStrategy bakingStrategy) {
        this.toppingStrategy = toppingStrategy;
        this.bakingStrategy = bakingStrategy;
    }
    @Override
    protected void addSauceAndToppings() {
        System.out.println("Adding tomato sauce...");
        toppingStrategy.addToppings();
    }
    @Override
    protected void bakePizza() {
        bakingStrategy.bake();
    }
}
```

```java
//src/Concretes/VeggiePizza


package Concretes;

import Abstracts.*;
import Behaviours.*;

public class VeggiePizza extends Pizza {
    private ToppingStrategy toppingStrategy;
    private BakingStrategy bakingStrategy;

    public VeggiePizza(ToppingStrategy toppingStrategy, BakingStrategy bakingStrategy) {
        this.toppingStrategy = toppingStrategy;
        this.bakingStrategy = bakingStrategy;
    }
    @Override
    protected void addSauceAndToppings() {
        System.out.println("Adding pesto sauce...");
        toppingStrategy.addToppings();
    }
    @Override
    protected void bakePizza() {
        bakingStrategy.bake();
    }
}
```

```java
//src/Strategies/FastBake

package Strategies;

import Behaviours.BakingStrategy;

public class FastBake implements BakingStrategy {
    @Override
    public void bake() {
        System.out.println("Baking at 500 F  for 10 minutes...");
    }
}
```

```java
//src/Strategies/PepperoniTopping

package Strategies;

import Behaviours.ToppingStrategy;

public class PepperoniTopping implements ToppingStrategy {
    @Override
    public void addToppings() {
        System.out.println("Adding pepperoni and cheese...");
    }
}
```

```java
//src/Strategies/SlowBake

package Strategies;

import Behaviours.BakingStrategy;

public class SlowBake implements BakingStrategy {
    @Override
    public void bake() {
        System.out.println("Baking at 350 F  for 20 minutes...");
    }
}
```

```java
//src/Strategies/VeggieTopping

package Strategies;

import Behaviours.ToppingStrategy;

public class VeggieTopping implements ToppingStrategy {
    @Override
    public void addToppings() {
        System.out.println("Adding vegetables and mozzarella...");
    }
}
```

```java
//src/App

import Abstracts.*;
import Behaviours.*;
import Concretes.*;
import Strategies.*;

public class App {
        public static void main(String[] args) throws Exception {
                ToppingStrategy pepperoniTopping = new PepperoniTopping();
                ToppingStrategy veggieTopping = new VeggieTopping();
                BakingStrategy fastBake = new FastBake();
                BakingStrategy slowBake = new SlowBake();

                // Create pizzas with specific strategies
                Pizza pepperoniPizza = new PepperoniPizza(pepperoniTopping, fastBake);
                Pizza veggiePizza = new VeggiePizza(veggieTopping, slowBake);

                // Cook pizzas
                System.out.println("Cooking Pepperoni Pizza:");
                pepperoniPizza.cookPizza();

                System.out.println("\nCooking Veggie Pizza:");
                veggiePizza.cookPizza();
        }

}
```

3 Factory Pattern

The Factory Pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created. This pattern promotes loose coupling by eliminating the need for the client code to instantiate the objects directly. Instead, it relies on a factory to create the objects.

Key Concepts of the Factory Pattern

- Factory Method: A method that is responsible for creating objects.
- Product Interface: An interface or abstract class defining the type of objects the factory method creates.
- Concrete Products: Concrete classes implementing the product interface.
- Creator: The class that contains the factory method and is responsible for creating objects.

When to use

- When a class cannot anticipate the type of objects it needs to create.
- When you want to delegate the responsibility of instantiating objects to subclasses.
- When you want to encapsulate the instantiation logic in a single location.

Key benefits

- Encapsulation: The factory encapsulates the object creation process, making it easier to manage and modify.
- Loose Coupling: Clients are decoupled from the concrete classes they instantiate, promoting flexibility and easier maintenance.
- Code Reusability: The same factory can be reused to create different products.

Strategy + Factory

Imagine a banking application that calculates interest for different types of loans like personal loans, home loans, and car loans. Each loan type has a different way of calculating the interest; a unique strategy.

```java
//src/Behaviours/InterestStrategy

package Behaviours;

public interface InterestStrategy {
    double calculateInterest(double loanAmount);
}
```

```java
//src/Concretes/Loan

package Concretes;

import Behaviours.InterestStrategy;
import Factories.*;

public class Loan {
    private final double loanAmount;
    private final InterestStrategy interestStrategy;
    private StrategyFactory strategyFactory = new StrategyFactory();

    public Loan(double loanAmount, String loanType) {
        this.loanAmount = loanAmount;
        this.interestStrategy = strategyFactory.getStrategy(loanType);
    }
    public double calculateInterest() {
        return interestStrategy.calculateInterest(loanAmount);
    }
}
```

```java
//src/Factories/StrategyFactory

package Factories;

import java.util.HashMap;
import java.util.Map;

import Behaviours.InterestStrategy;
import Strategies.*;

public class StrategyFactory {
    // Instance variable for storing strategies
    private final Map<String, InterestStrategy> strategies = new HashMap<>();
    // Constructor to initialize the strategies map
    public StrategyFactory() {
        strategies.put("personal", new PersonalLoanStrategy());
        strategies.put("home", new HomeLoanStrategy());
        strategies.put("car", new CarLoanStrategy());
    }
    // Instance method to retrieve the appropriate strategy
    public InterestStrategy getStrategy(String loanType) {
        return strategies.get(loanType);
    }
}
```

```java
//src/Strategies/CarLoanStrategy

package Strategies;

import Behaviours.InterestStrategy;

public class CarLoanStrategy implements InterestStrategy {
    @Override
    public double calculateInterest(double loanAmount) {
        return loanAmount * 0.4;
    }

}
```

```java
//src/Strategies/HomeLoanStrategy

package Strategies;

import Behaviours.InterestStrategy;

public class HomeLoanStrategy implements InterestStrategy {
    @Override
    public double calculateInterest(double loanAmount) {
        return loanAmount * 0.3;
    }

}
```

```java
//src/Strategies/PersonalLoanStrategy

package Strategies;

import Behaviours.InterestStrategy;

public class PersonalLoanStrategy implements InterestStrategy {
    @Override
    public double calculateInterest(double loanAmount) {
        return loanAmount * 0.1;
    }

}
```

```java
//src/App

import Concretes.Loan;

public class App {
        public static void main(String[] args) throws Exception {
                Loan loan = new Loan(10000, "personal");
                System.out.println(loan.calculateInterest());

                Loan loan2 = new Loan(10000, "home");
                System.out.println(loan2.calculateInterest());

                Loan loan3 = new Loan(10000, "car");
                System.out.println(loan3.calculateInterest());
        }

}
```

Template + Strategy + Factory

Consider we are building an application for a pizza store and we need a method to cook a pizza. There are three steps to cook a pizza in the pizza store:

- Prepare pizza dough;
- Add sauce and toppings;
- Bake the pizza.

The first step is a common behavior and the second and third steps are various for different pizza types. Now let's assume we have a lot of new types of pizza in store, so we need many new pizza cookers(subclasses). However, some of the cookers have the same logic in step 2 and some of them have the same logic in step 3. We want these cookers can reuse the same logic through application to reduce code duplication. Use Factory pattern to build the pizzas as well.

```
//src/Abstracts/Pizza

package Abstracts;

public abstract class Pizza {

    public final void cookPizza() {
        prepareDough();
        addSauceAndToppings();
        bakePizza();
    }

    // Step 1 is the same for all pizzas
    private void prepareDough() {
        System.out.println("Preparing pizza dough...");
    }

    // Steps 2 and 3 are customizable
    protected abstract void addSauceAndToppings();

    protected abstract void bakePizza();
}
```

```
//src/Bheaviours/BakingStrategy

package Behaviours;

public interface BakingStrategy {
    void bake();
}
```

```
//src/Bheaviours/ToppingStrategy

package Behaviours;

public interface ToppingStrategy {
    void addToppings();
}
```

```java
//src/Concretes/PepperoniPizza

package Concretes;

import Abstracts.*;
import Behaviours.*;

public class PepperoniPizza extends Pizza {
    private ToppingStrategy toppingStrategy;
    private BakingStrategy bakingStrategy;

    public PepperoniPizza(ToppingStrategy toppingStrategy, BakingStrategy bakingStrategy) {
        this.toppingStrategy = toppingStrategy;
        this.bakingStrategy = bakingStrategy;
    }

    @Override
    protected void addSauceAndToppings() {
        System.out.println("Adding tomato sauce...");
        toppingStrategy.addToppings();
    }

    @Override
    protected void bakePizza() {
        bakingStrategy.bake();
    }
}
```

```java
//src/Concretes/VeggiePizza


package Concretes;

import Abstracts.*;
import Behaviours.*;

public class VeggiePizza extends Pizza {
    private ToppingStrategy toppingStrategy;
    private BakingStrategy bakingStrategy;

    public VeggiePizza(ToppingStrategy toppingStrategy, BakingStrategy bakingStrategy) {
        this.toppingStrategy = toppingStrategy;
        this.bakingStrategy = bakingStrategy;
    }

    @Override
    protected void addSauceAndToppings() {
        System.out.println("Adding pesto sauce...");
        toppingStrategy.addToppings();
    }

    @Override
    protected void bakePizza() {
        bakingStrategy.bake();
    }
}
```

```java
//src/Factories/PizzaFactory

package Factories;

import Abstracts.*;
import Behaviours.*;
import Concretes.*;
import Strategies.*;

// Updated Factory class to create pizzas
public class PizzaFactory {

    public static Pizza createPizza(String type) {
        ToppingStrategy toppingStrategy;
        BakingStrategy bakingStrategy;

        switch (type.toLowerCase()) {
            case "pepperoni":
                // Use the pepperoni topping strategy and fast bake strategy
                toppingStrategy = new PepperoniTopping();
                bakingStrategy = new FastBake();
                return new PepperoniPizza(toppingStrategy, bakingStrategy);

            case "veggie":
                // Use the veggie topping strategy and slow bake strategy
                toppingStrategy = new VeggieTopping();
                bakingStrategy = new SlowBake();
                return new VeggiePizza(toppingStrategy, bakingStrategy);

            // Additional pizza types can be added here
            default:
                throw new IllegalArgumentException("Unknown pizza type: " + type);
        }
    }
}
```

```java
//src/Strategies/FastBake

package Strategies;

import Behaviours.BakingStrategy;

public class FastBake implements BakingStrategy {
    @Override
    public void bake() {
        System.out.println("Baking at 500 F  for 10 minutes...");
    }
}
```

```java
//src/Strategies/PepperoniTopping

package Strategies;

import Behaviours.ToppingStrategy;

public class PepperoniTopping implements ToppingStrategy {
    @Override
    public void addToppings() {
        System.out.println("Adding pepperoni and cheese...");
    }
}
```

```java
//src/Strategies/SlowBake

package Strategies;

import Behaviours.BakingStrategy;

public class SlowBake implements BakingStrategy {
    @Override
    public void bake() {
        System.out.println("Baking at 350 F  for 20 minutes...");
    }
}
```

```java
//src/Strategies/VeggieTopping

package Strategies;

import Behaviours.ToppingStrategy;

public class VeggieTopping implements ToppingStrategy {
    @Override
    public void addToppings() {
        System.out.println("Adding vegetables and mozzarella...");
    }
}
```

```java
//src/App

import Abstracts.*;
import Factories.*;

public class App {
        public static void main(String[] args) throws Exception {
                Pizza pepperoniPizza = PizzaFactory.createPizza("pepperoni");
                Pizza veggiePizza = PizzaFactory.createPizza("veggie");

                // Cook pizzas
                System.out.println("Cooking Pepperoni Pizza:");
                pepperoniPizza.cookPizza();

                System.out.println("\nCooking Veggie Pizza:");
                veggiePizza.cookPizza();
        }
}
```

4 Decorator Pattern

The Decorator Pattern is a structural design pattern that allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class. This pattern is particularly useful for adhering to the Single Responsibility Principle by allowing functionality to be divided into classes with specific concerns.

Key Concepts of the Decorator Pattern

- Component: An interface or abstract class that defines the operations that can be dynamically added to concrete components.
- Concrete Component: The class that implements the component interface. This is the object that will be decorated.
- Decorator: A class that also implements the component interface and has a reference to a component object. The decorator class delegates calls to the component object and can add additional functionality before or after the delegation.

When to use

- When you want to add responsibilities to objects dynamically and transparently, without affecting other objects.
- When extending functionality through subclassing would lead to an explosion of subclasses.

Key benefits

- Single Responsibility Principle: Each decorator class has a single responsibility for adding specific functionality.
- Open/Closed Principle: You can add new decorators without modifying existing code, making the system extensible.
- Flexible and Reusable: You can combine different decorators to create a variety of behaviors dynamically.

Decorator (Lab Task)

Lab IUTBeans, Coffee - Condiments

```
//src/Behaviours/ICoffee

package Behaviours;

public interface ICoffee {
    public String GetDescription();

    public double GetPrice();
}
```

```
//src/Behaviours/ICondiment

package Behaviours;

public interface ICondiment {
    public String GetDescription();

    public double GetPrice();
}
```

```java
//src/Concretes.Coffee/Americano

package Concretes.Coffee;

import Behaviours.ICoffee;
import Behaviours.ICondiment;

public class Americano implements ICoffee {
    private final ICondiment condiment;

    public Americano(ICondiment condiment) {
        this.condiment = condiment;
    }

    @Override
    public String GetDescription() {
        return "Americano" + (condiment != null ? " with " + condiment.GetDescription() : ""
            );
    }

    @Override
    public double GetPrice() {
        return 1.5 + (condiment != null ? condiment.GetPrice() : 0);
    }
}
```

```java
//src/Concretes,Coffee

package Concretes.Coffee;

import Behaviours.ICoffee;
import Behaviours.ICondiment;

public class Espresso implements ICoffee {
    private final ICondiment condiment;

    public Espresso(ICondiment condiment) {
        this.condiment = condiment;
    }

    @Override
    public String GetDescription() {
        return "Espresso" + (condiment != null ? " with " + condiment.GetDescription() : "")
            ;
    }

    @Override
    public double GetPrice() {
        return 1.1 + (condiment != null ? condiment.GetPrice() : 0);
    }
}
```

```java
//src/Decorators.Condiments/Milk

package Decorators.Condiments;

import Behaviours.ICondiment;

public class Milk implements ICondiment {
    private final ICondiment condiment;

    public Milk(ICondiment condiment) {
        this.condiment = condiment;
    }
    @Override
    public String GetDescription() {
        return "Milk" + (condiment != null ? " + " + condiment.GetDescription() : "");
    }
    @Override
    public double GetPrice() {
        return 0.5 + (condiment != null ? condiment.GetPrice() : 0);
    }
}
```

```java
//src/Decorators.Condiments/WhippedCream

package Decorators.Condiments;

import Behaviours.ICondiment;

public class WhippedCream implements ICondiment {
    private final ICondiment condiment;

    public WhippedCream(ICondiment condiment) {
        this.condiment = condiment;
    }
    @Override
    public String GetDescription() {
        return "Whipped Cream" + (condiment != null ? " + " + condiment.GetDescription() : "
            ");
    }
    @Override
    public double GetPrice() {
        return 0.7 + (condiment != null ? condiment.GetPrice() : 0);
    }
}
```

```java
//src/Decorators.Condiments/WhiteSugar

package Decorators.Condiments;

import Behaviours.ICondiment;

public class WhiteSugar implements ICondiment {
    private final ICondiment condiment;

    public WhiteSugar(ICondiment condiment) {
        this.condiment = condiment;
    }
    @Override
    public String GetDescription() {
       return "White Sugar" + (condiment != null ? " + " + condiment.GetDescription() : "");
    }
    @Override
    public double GetPrice() {
        return 0.2 + (condiment != null ? condiment.GetPrice() : 0);
    }
}
```

```
//src/App

import Behaviours.ICoffee;
import Concretes.Coffee.Americano;
import Concretes.Coffee.Espresso;
import Decorators.Condiments.Milk;
import Decorators.Condiments.WhippedCream;
import Decorators.Condiments.WhiteSugar;

public class App {
        public static void main(String[] args) throws Exception {
                ICoffee americano = new Americano(new Milk(new WhippedCream(null)));
                System.out.println(americano.GetDescription());
                System.out.println(String.format("%.2f", americano.GetPrice()));

                ICoffee espresso = new Espresso(
                                new WhiteSugar(new Milk(new WhippedCream(new WhiteSugar(new
                                    WhiteSugar(null))))));
                System.out.println(espresso.GetDescription());
                System.out.println(String.format("%.2f", espresso.GetPrice()));
        }
}
```

5 Composite Pattern

The Composite Pattern is a structural design pattern that allows you to compose objects into tree-like structures to represent part-whole hierarchies. This pattern enables clients to treat individual objects and compositions of objects uniformly. In essence, it allows you to create complex structures from simpler components.

Key Concepts of the Composite Pattern

- Component: An interface or abstract class defining the common operations for both leaf nodes and composite nodes.
- Leaf: Represents the individual objects in the composition. Leaf nodes implement the component interface and define specific behavior.
- Composite: Represents the composite nodes that can contain leaves and other composites. It implements the component interface and can delegate operations to its child components.

When to use

- When you want to represent part-whole hierarchies of objects.
- When clients should be able to treat both individual objects and compositions uniformly.
- When you need to simplify the client code for handling groups of objects.

Key benefits

- Simplifies Client Code: Clients can work with single objects and compositions of objects uniformly, reducing the complexity of the client code.
- Easier to Add New Components: New leaf or composite classes can be added without changing existing code, following the Open/Closed Principle.
- Hierarchical Representation: It allows for a clear and hierarchical representation of complex structures.

Composite (Lab Task)

Lab Bundlers

```java
//src/Product

import java.util.List;

public interface Product {
    String getName();
    String getDescription();
    double calculateTotalPrice();
    void displayProductInfo();
}
```

```java
//src/SimpleProduct

public class SimpleProduct implements Product {
    private String name;
    private String description;
    private double price;

    public SimpleProduct(String name, String description, double price) {
        this.name = name;
        this.description = description;
        this.price = price;
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public String getDescription() {
        return description;
    }

    @Override
    public double calculateTotalPrice() {
        return price;
    }

    @Override
    public void displayProductInfo() {
        System.out.println("Simple Product: " + name + " - " + description + " - Price: $" +
            price);
    }
}
```

```java
//src/Bundle

import java.util.ArrayList;
import java.util.List;

public abstract class Bundle implements Product {
    private String name;
    private String description;
    private double discount;
    protected List<Product> products = new ArrayList<>();

    public Bundle(String name, String description, double discount) {
        this.name = name;
        this.description = description;
        this.discount = discount;
        // Each concrete bundle will define its own products
        addPredefinedProducts();
    }

    // Abstract method that must be implemented by subclasses to add predefined products
    protected abstract void addPredefinedProducts();

    @Override
    public String getName() {
        return name;
    }

    @Override
    public String getDescription() {
        return description;
    }

    @Override
    public double calculateTotalPrice() {
        double totalPrice = 0;
        for (Product product : products) {
            totalPrice += product.calculateTotalPrice();
        }
        return totalPrice * (1 - discount);  // Apply bundle discount
    }

    @Override
    public void displayProductInfo() {
        System.out.println("Bundle: " + name + " - " + description + " - Discount: " + (
            discount * 100) + "%");
        for (Product product : products) {
            product.displayProductInfo();
        }
    }
}
```

```java
//src/StudentBundle

public class StudentBundle extends Bundle {

    public StudentBundle() {
        super("Student Bundle", "A bundle for students with necessary gadgets", 0.20);
    }

    @Override
    protected void addPredefinedProducts() {
        products.add(new SimpleProduct("Laptop", "Affordable student laptop", 600.0));
        products.add(new SimpleProduct("Headphones", "Noise-cancelling headphones", 150.0));
        products.add(new SimpleProduct("Notebook", "Set of notebooks", 20.0));
    }
}
```

```java
//src/OfficeBundle

public class OfficeBundle extends Bundle {

    public OfficeBundle() {
        super("Office Bundle", "A bundle of office essentials", 0.15);
    }

    @Override
    protected void addPredefinedProducts() {
        products.add(new SimpleProduct("Laptop", "High performance laptop", 1000.0));
        products.add(new SimpleProduct("Monitor", "24-inch LED Monitor", 200.0));
        products.add(new SimpleProduct("Keyboard", "Mechanical keyboard", 100.0));
    }
}
```

```java
//src/Main

public class Main {
    public static void main(String[] args) {
        // Creating predefined bundles
        Product officeBundle = new OfficeBundle();
        Product studentBundle = new StudentBundle();

        // Display product information and calculate total price for Office Bundle
        officeBundle.displayProductInfo();
        System.out.println("Total Price for Office Bundle: $" + officeBundle.
            calculateTotalPrice());

        System.out.println("\n");

        // Display product information and calculate total price for Student Bundle
        studentBundle.displayProductInfo();
        System.out.println("Total Price for Student Bundle: $" + studentBundle.
            calculateTotalPrice());

    }
}
```

Composite + Decorator

A publishing company needs a flexible document management system that can handle complex document structures with various formatting options. The system should be able to:

- Manage both simple text elements and composite document sections
- Apply multiple formatting styles to any text (bold, italic, highlighting, etc.)
- Calculate total character count including formatting characters
- Allow for easy addition of new formatting options in the future
- Support nested document structures of arbitrary depth

```java
//src/Behaviours/IDocumentElement

package Behaviours;

public interface IDocumentElement {
    void render();
    int getCharCount();
}
```

```java
//src/Abstracts/DocumentDecorator

package Abstracts;

import Behaviours.IDocumentElement;

public class DocumentDecorator implements IDocumentElement {
    protected IDocumentElement element;
    public DocumentDecorator(IDocumentElement element) {
        this.element = element;
    }
    @Override
    public void render() {
        element.render();
    }
    @Override
    public int getCharCount() {
        return element.getCharCount();
    }
}
```

```java
//src/Concretes/DocumentSection

package Concretes;

import java.util.ArrayList;
import java.util.List;
import Behaviours.IDocumentElement;

public class DocumentSection implements IDocumentElement {
    private List<IDocumentElement> elements = new ArrayList<>();
    private String sectionName;
    public DocumentSection(String sectionName) {
        this.sectionName = sectionName;
    }
    public void addElement(IDocumentElement element) {
        elements.add(element);
    }
    public void removeElement(IDocumentElement element) {
        elements.remove(element);
    }
    @Override
    public void render() {
        System.out.println("\n=== " + sectionName + " Start ===");
        for (IDocumentElement element : elements) {
            element.render();
        }
        System.out.println("\n=== " + sectionName + " End ===");
    }
    @Override
    public int getCharCount() {
        return elements.stream()
                .mapToInt(
                        IDocumentElement::getCharCount)
                .sum();
    }
}
```

```java
//src/Concretese/TextElement

package Concretes;

import Behaviours.IDocumentElement;

public class TextElement implements IDocumentElement {
    private String text;
    public TextElement(String text) {
        this.text = text;
    }
    @Override
    public void render() {
        System.out.print(text);
    }
    @Override
    public int getCharCount() {
        return text.length();
    }
}
```

```java
//src/Decorators/BoldDecorator

package Decorators;

import Abstracts.DocumentDecorator;
import Behaviours.IDocumentElement;

public class BoldDecorator extends DocumentDecorator {
    public BoldDecorator(IDocumentElement element) {
        super(element);
    }
    @Override
    public void render() {
        System.out.print("**");
        super.render();
        System.out.print("**");
    }
    @Override
    public int getCharCount() {
        return super.getCharCount() + 4;
    }
}
```

```java
//src/Decorators/HighlightDecorator

package Decorators;

import Abstracts.DocumentDecorator;
import Behaviours.IDocumentElement;

public class HighlightDecorator extends DocumentDecorator {
    public HighlightDecorator(IDocumentElement element) {
        super(element);
    }
     @Override
    public void render() {
        System.out.print("[[");
        super.render();
        System.out.print("]]");
    }
    @Override
    public int getCharCount() {
        return super.getCharCount() + 4;
    }
}
```

```java
//src/Decorators/ItalicDecorator

package Decorators;

import Abstracts.DocumentDecorator;
import Behaviours.IDocumentElement;

public class ItalicDecorator extends DocumentDecorator {
    public ItalicDecorator(IDocumentElement element) {
        super(element);
    }
    @Override
    public void render() {
        System.out.print("_");
        super.render();
        System.out.print("_");
    }
    @Override
    public int getCharCount() {
        return super.getCharCount() + 2;
    }
}
```

```java
//src/App

import Behaviours.IDocumentElement;
import Concretes.DocumentSection;
import Concretes.TextElement;
import Decorators.BoldDecorator;
import Decorators.HighlightDecorator;
import Decorators.ItalicDecorator;

public class App {
    public static void main(String[] args) throws Exception {
        // Create a document structure
        DocumentSection document = new DocumentSection("Main Document");
        // Create a subsection
        DocumentSection section1 = new DocumentSection("Introduction");
         // Create and decorate some text elements
        IDocumentElement title = new BoldDecorator(
                new TextElement("Document Title"));

        IDocumentElement highlight = new HighlightDecorator(
                new TextElement("Important Note"));

        IDocumentElement emphasisText = new ItalicDecorator(
                new TextElement("This is emphasized"));
        // Create a nested decorated element
        IDocumentElement complexText = new BoldDecorator(
                new ItalicDecorator(new HighlightDecorator(new TextElement("Bold, italic,
                    and highlighted text"))));
        // Build the document structure
        section1.addElement(title);
        section1.addElement(new TextElement("\n"));
        section1.addElement(highlight);
        section1.addElement(new TextElement("\n"));
        section1.addElement(emphasisText);
        section1.addElement(new TextElement("\n"));
        section1.addElement(complexText);

        document.addElement(section1);
        // Render the document
        document.render();
        // Show total character count
        System.out.println("\nTotal characters: " + document.getCharCount());
    }
}
```

6 Adapter Pattern

The Adapter Pattern is a structural design pattern that allows objects with incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces, enabling collaboration without modifying the existing code. The Adapter Pattern is particularly useful when integrating new systems with legacy systems or when you need to use a library that doesn't match your existing interfaces.

Key Concepts of the Adapter Pattern

- Target Interface: The interface that the client expects to interact with.
- Adaptee: The existing class with an interface that is incompatible with the target interface.
- Adapter: The class that implements the target interface and contains a reference to the adaptee. It translates calls from the target interface to the adaptee's interface.

When to use

- When you need to integrate with a legacy system or third-party library that does not match your expected interface.
- When you want to use an existing class and its interface does not match the one you need.

Key benefits

- Separation of Concerns: The Adapter Pattern allows for separation between the client and the adaptee. The client does not need to understand the details of how the adaptee works.
- Reusability: The adapter can be reused to adapt multiple incompatible interfaces without modifying existing code.
- Flexibility: It allows for dynamic composition of interfaces, as you can adapt various classes to the target interface without changing the original classes.

Adapter (Lab Task)

Postal Notify System

```
//src/Adapters/EPostalAdapter

package Adapters;

import Behaviours.INotify;
import pkg.EPostalMail;

public class EPostalAdapter implements INotify {
    EPostalMail ePostalMail = new EPostalMail();

    public void pushNotify(String uniqueId, String message) {
        ePostalMail.sendPostalMail(uniqueId, message, ePostalMail.getPostalCode(),
            ePostalMail.getPostalCode());
    }
}
```

```
//src/Behaviours/INotify

package Behaviours;

public interface INotify {
    void pushNotify(String uniqueId, String message);
}
```

```java
//src/Concretes/EmailNotify

package Concretes;

import Behaviours.INotify;

public class EmailNotify implements INotify {
    public void pushNotify(String uniqueId, String message) {
        System.out.println("Email sent to " + uniqueId + " with message: " + message);
    }
}
```

```java
//src/Concretese/SMSNotify

package Concretes;

import Behaviours.INotify;

public class SMSNotify implements INotify {
    public void pushNotify(String uniqueId, String message) {
        System.out.println("SMS sent to " + uniqueId + " with message: " + message);
    }
}
```

```java
//src/pkg/EPostalMail

package pkg;

public class EPostalMail {

    public void sendPostalMail(String uniqueId, String message, String PostalAddress, String
         PostalCode) {
        System.out.println("Postal mail sent to " + uniqueId + " with message: " + message +
             " to address: "
                + PostalAddress + " with postal code: " + PostalCode);
    }

    public String getPostalAddress() {
        return "Postal Address";
    }

    public String getPostalCode() {
        return "Postal Code";
    }
}
```

```java
//src/Sevices/NotificationService

package Services;

import Behaviours.INotify;

public class NotificationService {
    public void SendNotification(INotify notifier, String uniqueId, String message) {
        notifier.pushNotify(uniqueId, message);
    }
}
```

```java
//src/App

import Adapters.EPostalAdapter;
import Behaviours.INotify;
import Concretes.EmailNotify;
import Concretes.SMSNotify;
import Services.NotificationService;

public class App {
    public static void main(String[] args) throws Exception {
        NotificationService notificationService = new NotificationService();

        INotify email = new EmailNotify();
        INotify sms = new SMSNotify();
        INotify postalAdapter = new EPostalAdapter();

        String userId = "hasin023";
        String message = "Goodbye from Java";

        notificationService.SendNotification(postalAdapter, userId, message);
    }
}
```

Adapter IRL

Let's imagine that you were hired by Nintendo to meet a very specific demand, they need PS5 players to be able to use the Nintendo Switch Pro controller to play games. Why? Nintendo was acquired by Sony.

Right, we know that they are different platforms, so we have a nintendo controller that can not adapt to the playstation console because it is incompatible. After all we can not directly change the control of a Nintendo, it would generate more work. But the two controls have some features in common. The controls have directionals, triggers and buttons that perform actions. But there is no Playstation driver that accepts the inputs from a Nintendo control. The standard adapter can solve this problem. We have a class that needs to be integrated into the system, and we can't change it directly. So we can create an adapter that will do this conversion. So when the user clicks on the X button on the switch pro, the adapter will convert to the triangle button on the Playstation and so on.

- X button on the switch pro -¿ Triangle button on the Playstation
- Y button on the switch pro -¿ Square button on the Playstation
- B button on the switch pro -¿ Cross button on the Playstation
- A button on the switch pro -¿ Circle button on the Playstation

```java
//src/Behaviours/PlayStationController

package Behaviours;

public interface PlayStationController {
    void pressTriangle();
    void pressSquare();
    void pressCross();
    void pressCircle();
}
```

```java
//src/Adapters/SwitchToPlayStationAdapter

package Adapters;

import Behaviours.PlayStationController;
import Concretes.SwitchProController;

public class SwitchToPlayStationAdapter implements PlayStationController {
    private final SwitchProController switchController;

    public SwitchToPlayStationAdapter(SwitchProController switchController) {
        this.switchController = switchController;
    }
    @Override
    public void pressTriangle() {
        // Map X button to Triangle button
        switchController.pressX();
    }
    @Override
    public void pressSquare() {
        // Map Y button to Square button
        switchController.pressY();
    }
    @Override
    public void pressCross() {
        // Map B button to Cross button
        switchController.pressB();
    }
    @Override
    public void pressCircle() {
        // Map A button to Circle button
        switchController.pressA();
    }
}
```

```java
//src/Concretes/SwitchProController

package Concretes;

public class SwitchProController {
    public void pressX() {
        System.out.println("Switch Pro: X button pressed");
    }

    public void pressY() {
        System.out.println("Switch Pro: Y button pressed");
    }

    public void pressB() {
        System.out.println("Switch Pro: B button pressed");
    }

    public void pressA() {
        System.out.println("Switch Pro: A button pressed");
    }
}
```

```java
//src/App

import Adapters.SwitchToPlayStationAdapter;
import Behaviours.PlayStationController;
import Concretes.SwitchProController;

public class App {
        public static void main(String[] args) throws Exception {
                SwitchProController switchProController = new SwitchProController();

                // Use the adapter to translate Switch inputs to PlayStation inputs
                PlayStationController playStationController = new SwitchToPlayStationAdapter
                    (switchProController);

                playStationController.pressTriangle(); // Switch X button pressed
                playStationController.pressSquare(); // Switch Y button pressed
                playStationController.pressCross(); // Switch B button pressed
                playStationController.pressCircle(); // Switch A button pressed
        }
}
```

7 Singleton Pattern

The Singleton Pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to that instance. This pattern is useful when exactly one object is needed to coordinate actions across the system.

Key Concepts of the Singleton Pattern

- Single Instance: The Singleton class controls the instantiation process and ensures that only one instance of the class exists.
- Global Access Point: The instance can be accessed globally, which means it can be referenced from anywhere in the application.

When to use

- When a single instance of a class is required to control actions.
- When you want to avoid multiple instances of a class that can lead to resource conflicts or inconsistencies (like database connections, configuration settings, etc.).

Key benefits

- Controlled Access to the Instance: The Singleton Pattern provides a controlled way to access the single instance of a class, ensuring that only one instance exists.
- Lazy Initialization: The instance is created only when it is needed, which can help save resources.
- Global Access: The singleton instance can be accessed from anywhere in the application, providing a central point of access.

Singleton

Everyday Analogy
Think of a car key. Just as there's one key that starts your car, the Singleton Pattern ensures there's only one key to open a special door in your favorite store. No matter how many shoppers there are, only one person at a time can access the hidden treasure.
Domain Example
In the gaming industry, the Singleton Pattern can be used to manage a game's global leaderboard. There's a single instance that keeps track of all players' scores, ensuring fair and consistent ranking.
Best Fit Domain
Configuration Management, Database Connection Handling.

```java
//src/Behaviours/IKey

package Behaviours;

public interface IKey {
    public void unlockDoor();

    public void lockDoor();
}
```

```java
//src/StoreTreasureKey

import Behaviours.IKey;

public class StoreTreasureKey implements IKey {
    private static StoreTreasureKey instance;

    // Encapsulation: Private constructor to prevent instantiation from outside.
    private StoreTreasureKey() {
    }

    // Singleton Access: Static method to get the single instance of the class.
    public static StoreTreasureKey getInstance() {
        if (instance == null) {
            // Lazy initialization: Creating the instance only when needed.
            instance = new StoreTreasureKey();
        }

        // Return the single instance of the class.
        return instance;
    }

    // Behavior of unlocking the door
    @Override
    public void unlockDoor() {
        System.out.println("The store's treasure door is unlocked.");
    }

    // Behavior of locking the door
    @Override
    public void lockDoor() {
        System.out.println("The store's treasure door is locked.");
    }
}
```

```java
//src/App

public class App {
        public static void main(String[] args) throws Exception {
                StoreTreasureKey treasureKey1 = StoreTreasureKey.getInstance();
                treasureKey1.lockDoor();

                // Both references should point to the same instance.
                StoreTreasureKey treasureKey2 = StoreTreasureKey.getInstance();
                System.out.println(treasureKey1 == treasureKey2); // Output: true (Singleton
                    )

                // Using the other singleton instance to unlock the door
                treasureKey2.unlockDoor();
        }
}
```

Quiz 01 Solution

```java
//src/Behaviours/ITaskExecutionStrategy

package Behaviours;

public interface ITaskExecutionStrategy {
    String ExecuteTask();
}
```

```java
//src/Abstracts/Robot

package Abstracts;
import Behaviours.ITaskExecutionStrategy;

public abstract class Robot {
    public ITaskExecutionStrategy TaskExecutionStrategy;

    protected Robot(ITaskExecutionStrategy taskExecutionStrategy) {
        TaskExecutionStrategy = taskExecutionStrategy;
    }
    // Template Method defining the workflow
    public void ProcessTask() {
        InitializeRobot();
        MoveToLocation();
        PerformTask(); // Delegating task execution to the strategy
        ShutdownRobot();
    }
    protected abstract void InitializeRobot();

    protected abstract void MoveToLocation();

    protected abstract void ShutdownRobot();

    // Task execution step, delegated to strategy
    private void PerformTask() {
        String result = TaskExecutionStrategy.ExecuteTask();
        System.out.println(result);
    }
}
```

```java
//src/Concretes/CleaningRobot

package Concretes;
import Abstracts.Robot;
import Behaviours.ITaskExecutionStrategy;

public class CleaningRobot extends Robot {
    public CleaningRobot(ITaskExecutionStrategy taskExecutionStrategy) {
        super(taskExecutionStrategy);
    }
    @Override
    protected void InitializeRobot() {
        System.out.println("Cleaning robot initialized");
    }
    @Override
    protected void MoveToLocation() {
        System.out.println("Cleaning robot moving to location");
    }
    @Override
    protected void ShutdownRobot() {
        System.out.println("Cleaning robot shutting down");
    }
}
```

```java
//src/Concretes/RepairRobot

package Concretes;
import Abstracts.Robot;
import Behaviours.ITaskExecutionStrategy;

public class RepairRobot extends Robot {
    public RepairRobot(ITaskExecutionStrategy taskExecutionStrategy) {
        super(taskExecutionStrategy);
    }
    @Override
    protected void InitializeRobot() {
        System.out.println("Repair robot initialized");
    }
    @Override
    protected void MoveToLocation() {
        System.out.println("Repair robot moving to location");
    }
    @Override
    protected void ShutdownRobot() {
        System.out.println("Repair robot shutting down");
    }
}
```

```java
//src/Contexts/TaskExecution

package Contexts;
import Abstracts.Robot;
import Behaviours.ITaskExecutionStrategy;

public class TaskExecutionContext {
    private final Robot _processor;

    public TaskExecutionContext(Robot processor) {
        _processor = processor;
    }
    public void SetStrategy(ITaskExecutionStrategy strategy) {
        _processor.TaskExecutionStrategy = strategy;
    }
    public void Execute() {
        _processor.ProcessTask();
    }
}
```

```java
//src/Strategies/App

import Concretes.*;
import Contexts.*;
import Strategies.*;

public class App {
        public static void main(String[] args) throws Exception {
                CleaningRobot cleaningRobot = new CleaningRobot(new HighSpeedExecution());
            // Set the context with the processor
                TaskExecutionContext context = new TaskExecutionContext(cleaningRobot);
             // Process the task with the initial strategy
                context.Execute();
            // Change the strategy at runtime to PrecisionExecution
                context.SetStrategy(new PrecisionExecution());
                context.Execute();
            // You could also create a RepairRobotProcessor and switch strategies as needed
                RepairRobot repairRobot = new RepairRobot(new HighSpeedExecution());
                TaskExecutionContext repairContext = new TaskExecutionContext(repairRobot);
                repairContext.Execute();
        }
}
```

Quiz 02 Solution

```java
//src/Abstracts/ImageServiceDecorator

package Abstracts;

import Behaviours.IImageService;

public abstract class ImageServiceDecorator implements IImageService {

    protected IImageService wrappedImageService;

    public ImageServiceDecorator(IImageService imageService) {
        this.wrappedImageService = imageService;
    }

    @Override
    public byte[] processImage(byte[] image) {
        return wrappedImageService.processImage(image);
    }

    @Override
    public void sendImage(byte[] image, String recipient) {
        wrappedImageService.sendImage(image, recipient);
    }
}
```

```java
//src/Behaviours/IImageService

package Behaviours;

public interface IImageService {
    byte[] processImage(byte[] image);

    void sendImage(byte[] image, String recipient);
}
```

```java
//src/Concretese/BaseImageService

package Concretes;

import Behaviours.IImageService;

public class BaseImageService implements IImageService {

    @Override
    public byte[] processImage(byte[] image) {
        System.out.println("Reading image from file...");
        return image;
    }

    @Override
    public void sendImage(byte[] image, String recipient) {
        System.out.println("Sending image to " + recipient + "...");
    }
}
```

```java
//src/Decorators/CompressionDecorator

package Decorators;

import Abstracts.ImageServiceDecorator;
import Behaviours.IImageService;

public class CompressionDecorator extends ImageServiceDecorator {

    public CompressionDecorator(IImageService imageService) {
        super(imageService);
    }

    @Override
    public byte[] processImage(byte[] image) {
        image = super.processImage(image);
        System.out.println("Compressing image...");
        return image;
    }
}
```

```java
//src/Decorators/ResizeDecorator

package Decorators;

import Abstracts.ImageServiceDecorator;
import Behaviours.IImageService;

public class ResizeDecorator extends ImageServiceDecorator {

    public ResizeDecorator(IImageService imageService) {
        super(imageService);
    }

    @Override
    public byte[] processImage(byte[] image) {
        image = super.processImage(image);
        System.out.println("Resizing image to 1:1 ratio...");
        return image;
    }
}
```

```java
//src/Decorators/VirusScanDecorator

package Decorators;

import Abstracts.ImageServiceDecorator;
import Behaviours.IImageService;

public class VirusScanDecorator extends ImageServiceDecorator {
    public VirusScanDecorator(IImageService imageService) {
        super(imageService);
    }

    @Override
    public byte[] processImage(byte[] image) {
        image = super.processImage(image);
        System.out.println("Scanning image for viruses...");
        return image;
    }
}
```

```java
//src/Services/ChatImageService

package Services;

import Behaviours.IImageService;
import Concretes.BaseImageService;
import Decorators.CompressionDecorator;
import Decorators.ResizeDecorator;
import Decorators.VirusScanDecorator;

public class ChatImageService {

    public void processAndSendImage(String imagePath, String recipient) {
        IImageService service = new BaseImageService();
        service = new VirusScanDecorator(service);
        service = new ResizeDecorator(service);
        service = new CompressionDecorator(service); // Compression added here

        byte[] image = service.processImage(new byte[] {});
        service.sendImage(image, recipient);
    }

}
```

```java
//src/Services/ProfileImageServices

package Services;

import Behaviours.IImageService;
import Concretes.BaseImageService;
import Decorators.ResizeDecorator;
import Decorators.VirusScanDecorator;

public class ProfileImageService {
    public void processAndSendImage(String imagePath, String recipient) {
        IImageService service = new BaseImageService();
        service = new VirusScanDecorator(service);
        service = new ResizeDecorator(service);
        // No compression decorator here

        byte[] image = service.processImage(new byte[] {});
        service.sendImage(image, recipient);
    }
}
```

```java
//src/App

import Services.ChatImageService;
import Services.ProfileImageService;

public class App {
        public static void main(String[] args) throws Exception {
                ChatImageService chatService = new ChatImageService();
                chatService.processAndSendImage("chatImage.jpg", "recipient@example.com");

                // ProfileImageService without compression
                ProfileImageService profileService = new ProfileImageService();
                profileService.processAndSendImage("profileImage.jpg", "recipient@example.
                    com");
        }

}
```