Heaven's light is our guide

Rajshahi University of Engineering & Technology

Department of Computer Science and Engineering

LAB REPORT

Course Name: Software Engineering Sessional

Course Code: CSE 3206

Submitted by	Submitted to
Roll: 2003178,2003179,2003180,2003181 Section: C Series: 20	Farjana Parvin Lecturer Department of CSE, RUET

Template Design Pattern: Behavioral Design Pattern

Template Design Pattern or Template Method is the behavioral design pattern that defines the skeleton of an algorithm in the superclass and allows its subclasses override specific steps of the algorithm without changing its structure.

When to use:

When you want all classes to follow specific steps to process a common task but each class implements their own logic in that specific step

Key component of template design pattern:

Abstract Class

- Define an abstract class that declares the template method. The template method typically consists of a series of method calls and control flow statements.
- The template method defines the algorithm's structure but leaves some steps to be implemented by concrete subclasses.

Concrete Classes

- Create concrete subclasses that extend the abstract class.
- Implement the abstract methods declared in the abstract class to provide specific behavior for each step of the algorithm.

Abstract (or Hook) Methods:

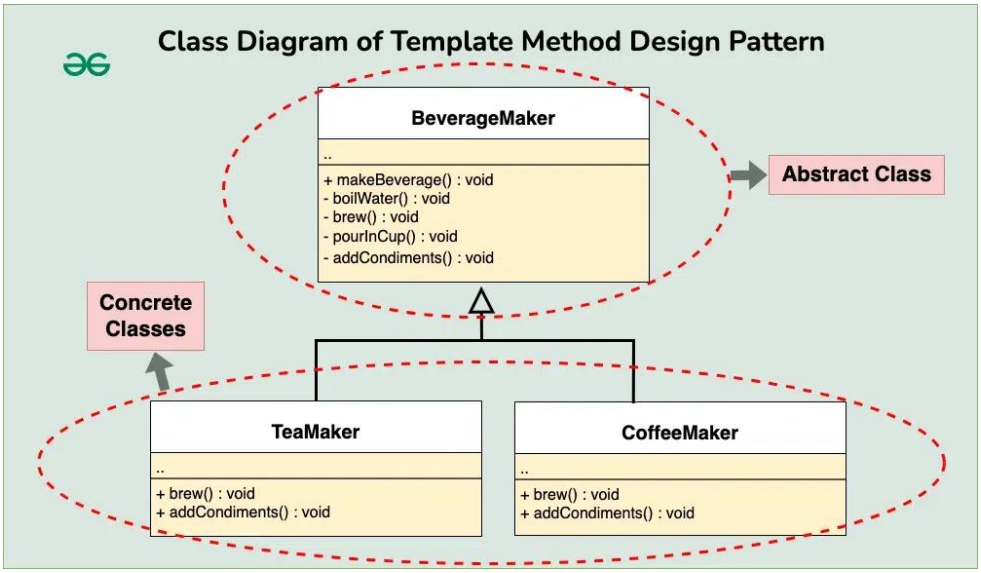
- These are methods declared within the abstract class but not implemented.

Problem Statement:

Let's consider a scenario where we have a process for making different types of beverages, such as tea and coffee. While the overall process of making beverages is similar (e.g., boiling water, adding ingredients), the specific steps and ingredients vary for each type of beverage.

In this problem the specific common steps should follow:

1. boilWater
2. addMainIngredient
3. pourInCup
4. addCondiments



Implementation in Java:

```java
// Abstract class defining the template method
abstract class BeverageMaker {
    // Template method defining the overall process
    public final void makeBeverage() {
        boilWater();
        addMainIngredient();
        pourInCup();
        addCondiments();
    }

    // Abstract methods to be implemented by subclasses
    abstract void addMainIngredient();
    abstract void addCondiments();

    // Common methods
    void boilWater() {
        System.out.println("Boiling water");
    }
```

```
    void pourInCup() {
        System.out.println("Pouring into cup");
    }
}

// Concrete subclass for making tea
class TeaMaker extends BeverageMaker {
    // Implementing abstract methods
    @Override
    void addMainIngredient() {
        System.out.println("Steeping the tea");
    }

    @Override
    void addCondiments() {
        System.out.println("Adding lemon");
    }
}

// Concrete subclass for making coffee
class CoffeeMaker extends BeverageMaker {
    // Implementing abstract methods
    @Override
    void addMainIngredient() {
        System.out.println("Dripping coffee through filter");
    }

    @Override
    void addCondiments() {
        System.out.println("Adding sugar and milk");
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        System.out.println("Making tea:");
        BeverageMaker teaMaker = new TeaMaker();
        teaMaker.makeBeverage();

        System.out.println("\nMaking coffee:");
        BeverageMaker coffeeMaker = new CoffeeMaker();
        coffeeMaker.makeBeverage();
    }
}
```

Output:

```
Making tea:
Boiling water
Steeping the tea
Pouring into cup
Adding lemon

Making coffee:
Boiling water
Dripping coffee through filter
Pouring into cup
Adding sugar and milk
```

Advantage:

1. **Reusability:** The BeverageTemplate class provides common steps like boiling water and pouring the beverage into a cup, avoiding code duplication.
2. **Extensibility:** Adding a new beverage (e.g., hot chocolate) requires creating a new subclass and implementing the abstract methods addMainIngredient() and addCondiments().
3. **Consistency:** The prepareBeverage method ensures the steps are followed in the correct order.

4. **Flexibility for Subclasses:** Subclasses can provide their own implementations for specific steps of the algorithm without changing the overall structure

Visitor Pattern: Behavioral Design Pattern

The **Visitor Pattern** is a design pattern that allows us to add new operations to a group of objects (or classes) without modifying their code. It achieves this by separating the operation logic from the object structure.

Problem:
Let's Imagine an **e-commerce system** where we have two types of items:

1. **Book:** Represents books for sale.
2. **Electronic:** Represents electronic gadgets for sale.

We want to perform two operations:

1. **Calculate shipping cost.**
2. **Generate item details for display.**

Using the **Visitor Pattern**, we'll encapsulate these operations in separate visitor classes, keeping the logic for operations independent of the item classes.

Explanation:

1. Define the Element Interface

The Element interface declares the accept() method. This method allows a visitor to "visit" the element.

```
// Element Interface
```

```
interface Item {
    void accept(Visitor visitor); // Accepts a visitor
}
```

2. Create ConcreteElements (Book and Electronic)

These classes implement the Item interface and define their unique properties and behaviors.

```
// ConcreteElement: Book
class Book implements Item {
    private String title;
    private double price;
    private double weight;

    public Book(String title, double price, double weight) {
        this.title = title;
        this.price = price;
        this.weight = weight;
    }

    public String getTitle() {
        return title;
    }

    public double getPrice() {
        return price;
    }

    public double getWeight() {
        return weight;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this); // Calls the visitor's visit(Book) method
    }
}

// ConcreteElement: Electronic
class Electronic implements Item {
    private String name;
    private double price;
    private double shippingWeight;
```

```
    public Electronic(String name, double price, double shippingWeight) {
        this.name = name;
        this.price = price;
        this.shippingWeight = shippingWeight;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }

    public double getShippingWeight() {
        return shippingWeight;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this); // Calls the visitor's visit(Electronic) method
    }
}
```

3. Define the Visitor Interface

The Visitor interface declares methods for visiting each type of element

```
// Visitor Interface
interface Visitor {
    void visit(Book book);          // Operation for Book
    void visit(Electronic electronic); // Operation for Electronic
}
```

4. Create ConcreteVisitors

Concrete visitors define the specific operations to perform on each element type.

```
// ConcreteVisitor: ShippingCostVisitor
class ShippingCostVisitor implements Visitor {
    @Override
    public void visit(Book book) {
        double cost = book.getWeight() * 2.0; // Example: $2 per unit weight
```

```
        System.out.println("Shipping cost for book \"" + book.getTitle() + "\":
$" + cost);
    }

    @Override
    public void visit(Electronic electronic) {
        double cost = electronic.getShippingWeight() * 3.0; // Example: $3 per
unit weight
        System.out.println("Shipping cost for electronic \"" +
electronic.getName() + "\": $" + cost);
    }
}

// ConcreteVisitor: DisplayDetailsVisitor
class DisplayDetailsVisitor implements Visitor {
    @Override
    public void visit(Book book) {
        System.out.println("Book Details: " + book.getTitle() + ", Price: $" +
book.getPrice());
    }

    @Override
    public void visit(Electronic electronic) {
        System.out.println("Electronic Details: " + electronic.getName() + ",
Price: $" + electronic.getPrice());
    }
}
```

5. Use the Visitor Pattern in the Main Application

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

public class VisitorPatternExample {
    public static void main(String[] args) {
        // Create a list of items
        List<Item> items = new ArrayList<>();
        items.add(new Book("Java Programming", 50.0, 1.5));
        items.add(new Electronic("Smartphone", 600.0, 0.8));

        // Create visitors
        Visitor shippingCostVisitor = new ShippingCostVisitor();
        Visitor displayDetailsVisitor = new DisplayDetailsVisitor();
```

```
        // Apply visitors to each item
        System.out.println("Displaying item details:");
        for (Item item : items) {
            item.accept(displayDetailsVisitor); // Display details
        }

        System.out.println("\nCalculating shipping costs:");
        for (Item item : items) {
            item.accept(shippingCostVisitor); // Calculate shipping cost
        }
    }
}
```

Complete Code:

Code:

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

interface Item {
    void accept(Visitor visitor);
}

class Book implements Item {
    private String title;
    private double price;
    private double weight;

    public Book(String title, double price, double weight) {
        this.title = title;
        this.price = price;
        this.weight = weight;
    }

    public String getTitle() {
        return title;
    }

    public double getPrice() {
        return price;
    }

    public double getWeight() {
        return weight;
```

```
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

class Electronic implements Item {
    private String name;
    private double price;
    private double shippingWeight;

    public Electronic(String name, double price, double shippingWeight) {
        this.name = name;
        this.price = price;
        this.shippingWeight = shippingWeight;
    }

    public String getName() {
        return name;
    }

    public double getPrice() {
        return price;
    }

    public double getShippingWeight() {
        return shippingWeight;
    }

    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}

interface Visitor {
    void visit(Book book);
    void visit(Electronic electronic);
}

class ShippingCostVisitor implements Visitor {
    @Override
    public void visit(Book book) {
```

```
        double cost = book.getWeight() * 2.0;
        System.out.println("Shipping cost for book \"" + book.getTitle() + "\":
$" + cost);
    }

    @Override
    public void visit(Electronic electronic) {
        double cost = electronic.getShippingWeight() * 3.0;
        System.out.println("Shipping cost for electronic \"" +
electronic.getName() + "\": $" + cost);
    }
}

class DisplayDetailsVisitor implements Visitor {
    @Override
    public void visit(Book book) {
        System.out.println("Book Details: " + book.getTitle() + ", Price: $" +
book.getPrice());
    }

    @Override
    public void visit(Electronic electronic) {
        System.out.println("Electronic Details: " + electronic.getName() + ",
Price: $" + electronic.getPrice());
    }
}

public class VisitorPatternExample {
    public static void main(String[] args) {
        List<Item> items = new ArrayList<>();
        items.add(new Book("Java Programming", 50.0, 1.5));
        items.add(new Electronic("Smartphone", 600.0, 0.8));

        Visitor shippingCostVisitor = new ShippingCostVisitor();
        Visitor displayDetailsVisitor = new DisplayDetailsVisitor();

        System.out.println("Displaying item details:");
        for (Item item : items) {
            item.accept(displayDetailsVisitor);
        }

        System.out.println("\nCalculating shipping costs:");
        for (Item item : items) {
            item.accept(shippingCostVisitor);
        }
```

```
    }
}
```

Output:

```
bangtansistu@BANGTAN SISTU farzanaa ma'am % cd "/Users/bangtansistu/3107 lab materi
al/farzanaa ma'am/" && javac VisitorPatternExample.java && java VisitorPatternExamp
le
Displaying item details:
Book Details: Java Programming, Price: $50.0
Electronic Details: Smartphone, Price: $600.0

Calculating shipping costs:
Shipping cost for book "Java Programming":$3.0
Shipping cost for electronic "Smartphone": $2.4000000000000004
bangtansistu@BANGTAN SISTU farzanaa ma'am %
```