

Heaven's light is our guide

Rajshahi University of Engineering & Technology



**Computer Science and Engineering
LAB REPORT**

Course Name: Software Engineering Sessional

Course Code: CSE 3206

Submitted by	Submitted to
Roll: 2003148,2003149,2003150 Section: C Series: 20	Farjana Parvin Lecturer Department of CSE, RUET

Template Design Pattern: Behavioral Design Pattern

The Template Design Pattern defines the skeleton of an algorithm in a base class and allows its subclasses to override specific steps without altering the algorithm's structure.

Problem:

Imagine we are designing a system to process different types of reports:

1. SalesReport: Provides details about sales performance.
2. InventoryReport: Summarizes inventory status.

For all reports, there are common steps:

1. Gather data.
2. Format the data.
3. Generate the final report.

The way data is gathered and formatted may vary for each type of report, but the overall process remains the same. Using the Template Design Pattern, we encapsulate the process in a base class and let subclasses define the specific steps.

Explanation:

1. Define the Abstract Class

The abstract base class contains the template method, which defines the skeleton of the algorithm. It also includes abstract or virtual methods for the steps that need customization.

2. Create Concrete Subclasses

Concrete subclasses implement the abstract methods, providing specific behavior for the customizable steps.

Implementation in Java

// Abstract class defining the template method

```
abstract class ReportTemplate {
```

```
    // Template method defining the skeleton of the algorithm
```

```
    public final void generateReport() {  
        gatherData();  
        formatData();  
        printReport();  
    }
```

```
    // Abstract methods to be implemented by subclasses
```

```
    protected abstract void gatherData();
```

```
    protected abstract void formatData();
```

```
    // Common method with default implementation
```

```
    private void printReport() {  
        System.out.println("Printing the report...");  
    }  
}
```

```
// Concrete class for Sales Report
```

```
class SalesReport extends ReportTemplate {
```

```
    @Override
```

```
    protected void gatherData() {
```

```
        System.out.println("Gathering sales data from
```

```
database...");  
}
```

```
@Override  
protected void formatData() {  
    System.out.println("Formatting sales data into  
tables...");  
}  
}
```

```
// Concrete class for Inventory Report  
class InventoryReport extends ReportTemplate {
```

```
@Override  
protected void gatherData() {  
    System.out.println("Gathering inventory data  
from warehouse system...");  
}
```

```
@Override  
protected void formatData() {  
    System.out.println("Formatting inventory data  
into charts...");  
}  
}
```

```
// Main class to demonstrate the Template Pattern  
public class TemplatePatternExample {
```

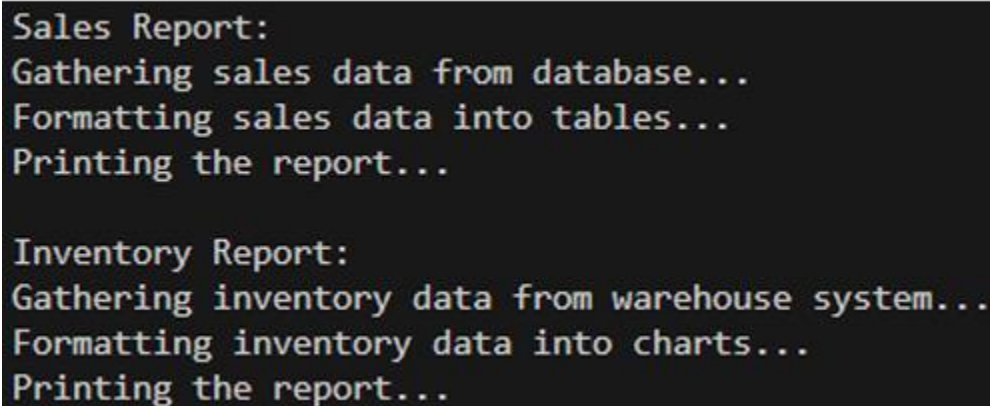
```
    public static void main(String[] args) {  
        // Generate a sales report  
        System.out.println("Sales Report:");  
        ReportTemplate salesReport = new SalesReport();
```

```
salesReport.generateReport();

System.out.println();

// Generate an inventory report
System.out.println("Inventory Report:");
ReportTemplate inventoryReport = new
InventoryReport();
inventoryReport.generateReport();
}
}
```

Output:



```
Sales Report:
Gathering sales data from database...
Formatting sales data into tables...
Printing the report...

Inventory Report:
Gathering inventory data from warehouse system...
Formatting inventory data into charts...
Printing the report...
```

Conclusion:

1. The template method (generateReport) defines the fixed steps of the process.
 2. Abstract methods (gatherData, formatData) allow customization of specific steps.
 3. Common logic (printReport) is implemented in the base class to ensure consistency.
- This pattern is useful when you want to enforce a specific workflow while still allowing flexibility in parts of the process.

Software Design Pattern:

A standard solution to a common programming problem.

A design or implementation structure that achieves a particular purpose. It is a description or template for how to solve a problem that can be used in many different situations.

A technique for making code more flexible or efficient.

What Are Behavioral Design Patterns?

Behavioral design patterns focus on communication between objects. They define how objects should cooperate and assign responsibilities to improve flexibility and maintainability. These patterns help in managing algorithms, relationships, and responsibilities among objects.

an **object** refers to an instance of a class that encapsulates both **data** (attributes or properties) and **behavior** (methods or functions).

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.

Advantages

Open/Closed Principle: You can add new operations without modifying existing object structures.

Separation of Concerns: Business logic is separated from object structure.

Disadvantages

Adding new element types requires updating all visitors.

Can become complex with too many element and visitor types.

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 %

```

