

# Template Design Pattern

Roll-2003146

## Objective

To demonstrate the implementation of the Template Design Pattern, which is part of the behavioral patterns in software design. This pattern allows the definition of a skeleton of an algorithm in an abstract class and lets subclasses override certain steps without altering the overall structure.

## Introduction

The Template Pattern is used to define a set of steps for an operation, where the specific details of some steps can vary. In this implementation, we demonstrate the pattern by creating an abstract class Game with defined methods that subclasses Cricket and Football extend to provide their specific implementations.

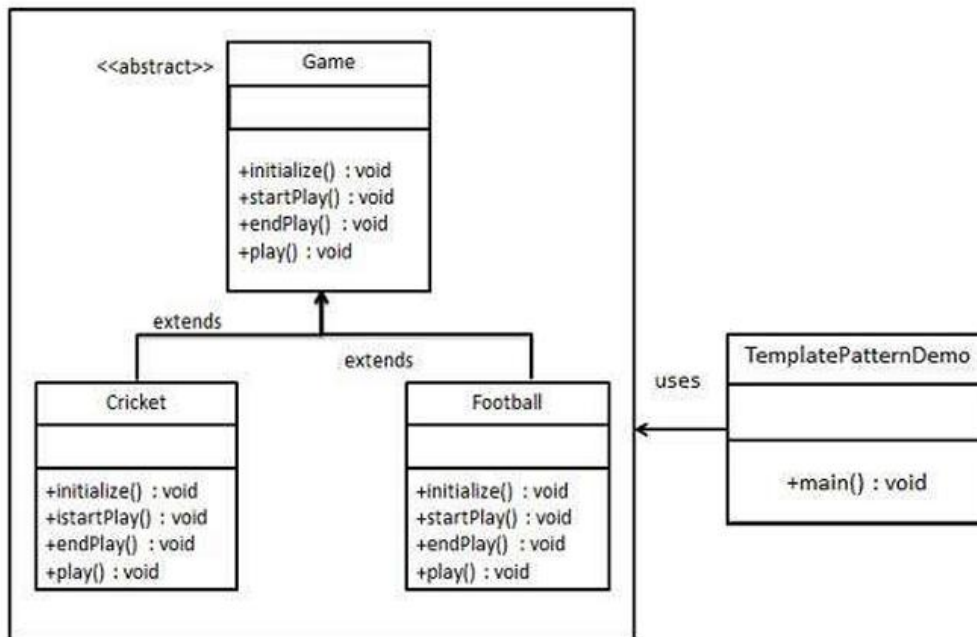
## Problem Statement

Develop an abstract class Game with a final template method to define the flow of operations. Concrete classes (Cricket, Football) will extend Game and implement specific behaviors. The TemplatePatternDemo class will demonstrate the use of the Template Pattern.

## Methodology

- Explain the approach used to solve the problem (step-by-step process of implementation).
- **Step 1:** Create an abstract class Game with a final template method (play()) and abstract methods for subclasses.
- **Step 2:** Implement concrete subclasses (Cricket and Football) to override the abstract methods.
- **Step 3:** Develop a TemplatePatternDemo class to test the implementation.

**Implementation:** Class Diagram Include a short description like: *"The following class diagram illustrates the structure of the Template Design Pattern, where the abstract Game class defines a template method and abstract methods. The concrete classes Cricket and Football extend Game and implement the abstract methods. The TemplatePatternDemo class demonstrates the use of this pattern."*

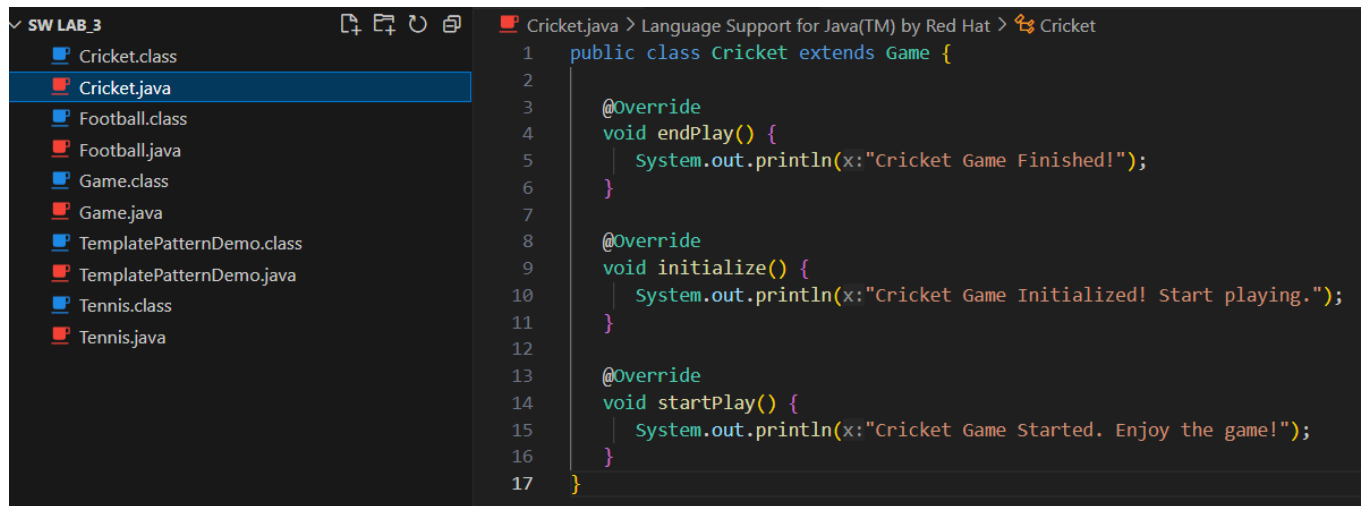


**Step 1:** Create an abstract class with a template method being final.

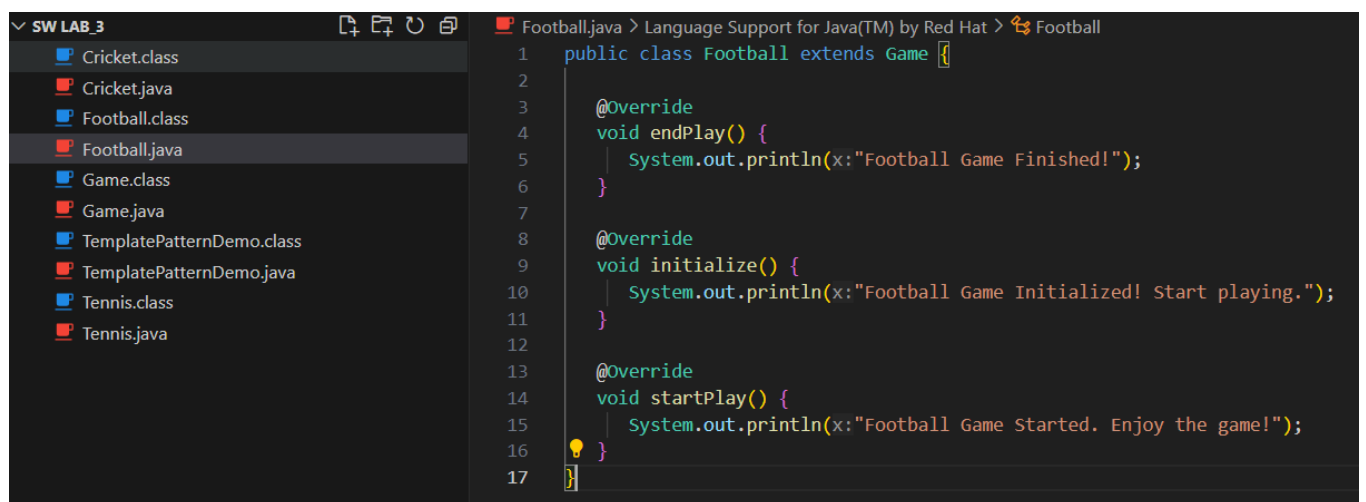
```
SW LAB_3
  Cricket.class
  Cricket.java
  Football.class
  Football.java
  Game.class
  Game.java
  TemplatePatternDemo.class
  TemplatePatternDemo.java
  Tennis.class
  Tennis.java

1 // Source code is decompiled from a .class file using FernFlower decompiler.
2 public abstract class Game {
3     public Game() {
4     }
5
6     abstract void initialize();
7
8     abstract void startPlay();
9
10    abstract void endPlay();
11
12    public final void play() {
13        this.initialize();
14        this.startPlay();
15        this.endPlay();
16    }
17 }
18
```

**Step 2:**Create concrete classes extending the above class.



```
1 public class Cricket extends Game {
2
3     @Override
4     void endPlay() {
5         System.out.println(x:"Cricket Game Finished!");
6     }
7
8     @Override
9     void initialize() {
10        System.out.println(x:"Cricket Game Initialized! Start playing.");
11    }
12
13    @Override
14    void startPlay() {
15        System.out.println(x:"Cricket Game Started. Enjoy the game!");
16    }
17 }
```



```
1 public class Football extends Game {
2
3     @Override
4     void endPlay() {
5         System.out.println(x:"Football Game Finished!");
6     }
7
8     @Override
9     void initialize() {
10        System.out.println(x:"Football Game Initialized! Start playing.");
11    }
12
13    @Override
14    void startPlay() {
15        System.out.println(x:"Football Game Started. Enjoy the game!");
16    }
17 }
```

**Step 3:** Use the *Game*'s template method `play()` to demonstrate a defined way of playing game

**Results:** Upon executing the TemplatePatternDemo class, the following output is produced:

For Cricket:

```
Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!
```

For Football:

```
Football Game Initialized! Start playing.
Football Game Started. Enjoy the game!
Football Game Finished!
```

If add more game then output shows

```
Tennis Game Initialized! Start practicing.
Tennis Game Started. Enjoy the rally!
Tennis Game Finished!
```

## Discussion

The Template Design Pattern is highly effective when designing systems that require a common structure but allow variation in specific steps. Key benefits demonstrated in this lab include:

- **Code Reusability:** Common logic is encapsulated in the Game abstract class.

- Flexibility: Subclasses can customize the steps (initialize(), startPlay(), and endPlay()).
- Consistency: Ensures that all games follow the same algorithmic flow.

## Conclusion

- Summarize the key points.
- Example: *"The Template Design Pattern was successfully implemented using Java to simulate two different games (Cricket and Football). The pattern enforced a fixed sequence of operations while allowing subclass-specific implementations, demonstrating its utility in standardizing algorithm design."*

**Roll:2003147**

**Title: Design Patterns - Strategy Pattern**

## Introduction

In software design, the Strategy Pattern is a behavioral design pattern that allows a class's behavior or algorithm to be selected at runtime. This pattern is particularly useful for scenarios where multiple algorithms can be applied to solve a problem, and the choice of algorithm is determined dynamically.

The Strategy Pattern involves defining a family of algorithms, encapsulating each one in a separate class, and making them interchangeable. This report outlines the implementation of the Strategy Pattern with a practical demonstration.

---

## 2. Objective

The objective of this lab is to:

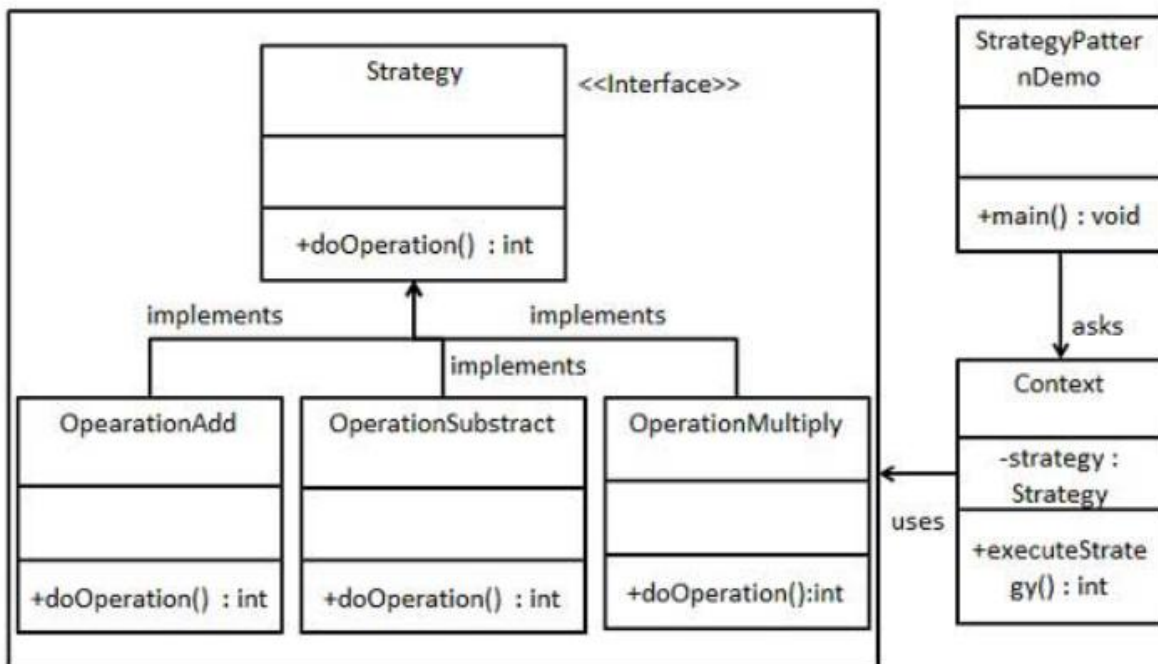
1. Understand the Strategy Pattern in software design.
2. Implement the Strategy Pattern in Java.

3. Demonstrate dynamic behavior changes in a context class using different strategies.

### Problem Statement:

The objective is to design a Strategy interface that defines an action and implement concrete strategy classes adhering to this interface. A Context class will be used to integrate and apply the chosen strategy. The StrategyPatternDemo class will serve as a demonstration, showcasing how the behavior of the Context can dynamically change depending on the strategy selected and utilized.

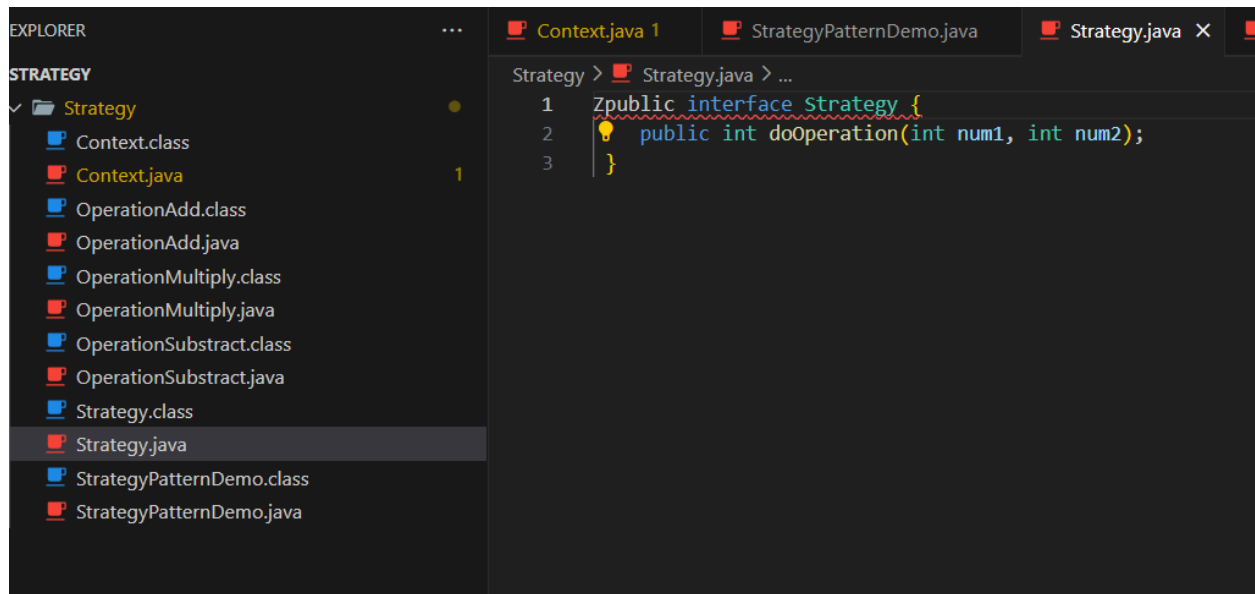
### Implementation



We are going to create a *Strategy* interface defining an action and concrete strategy classes implementing the *Strategy* interface. *Context* is a class which uses a *Strategy*.

*StrategyPatternDemo*, our demo class, will use *Context* and strategy objects to demonstrate change in *Context* behaviour based on strategy it deploys or us

Step 1: Create an interface:

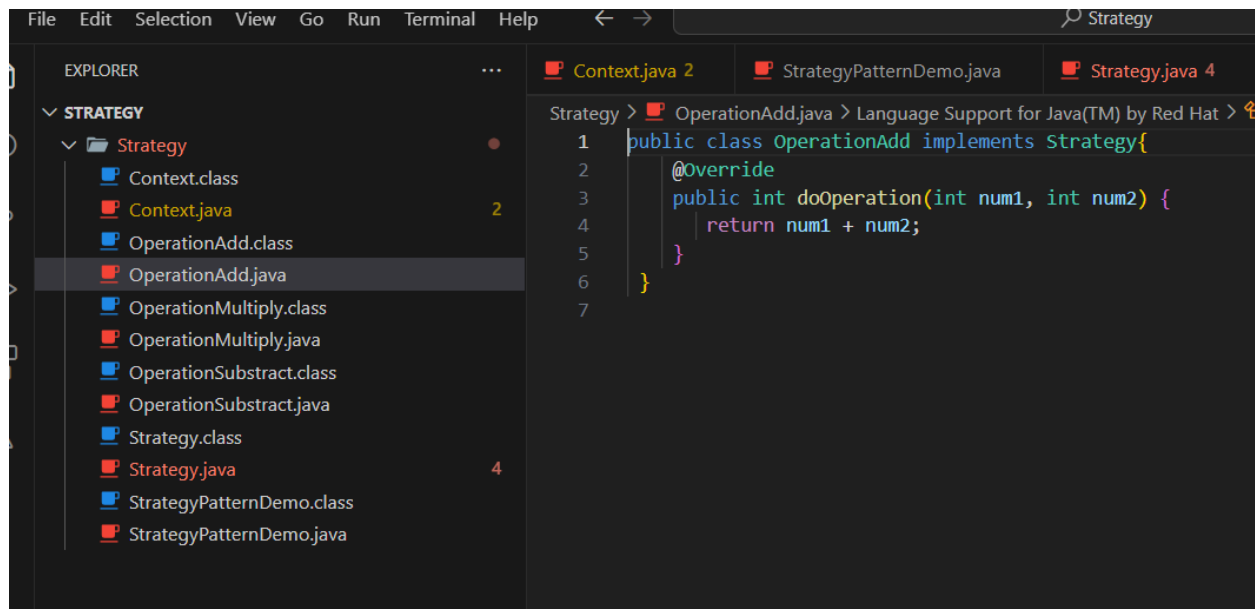


The screenshot shows an IDE with the Explorer panel on the left and the editor on the right. The Explorer panel shows a project named 'STRATEGY' with a folder 'Strategy' containing several files. The file 'Strategy.java' is selected. The editor shows the code for the 'Strategy' interface.

```
1 public interface Strategy {  
2     public int doOperation(int num1, int num2);  
3 }
```

Step 2

Create concrete classes implementing the same interface.



The screenshot shows the same IDE as before, but now the file 'OperationAdd.java' is selected in the Explorer panel. The editor shows the code for the 'OperationAdd' class, which implements the 'Strategy' interface.

```
1 public class OperationAdd implements Strategy {  
2     @Override  
3     public int doOperation(int num1, int num2) {  
4         return num1 + num2;  
5     }  
6 }  
7
```

```
1 public class OperationSubtract implements Strategy{
2     @Override
3     public int doOperation(int num1, int num2) {
4         return num1 - num2;
5     }
6 }
```

```
1 public class OperationMultiply implements Strategy{
2     @Override
3     public int doOperation(int num1, int num2) {
4         return num1 * num2;
5     }
6 }
7
```

Step 3: Create *Context* Class.

```
1 public class Context {
2     private Strategy strategy;
3
4     public Context(Strategy strategy){
5         this.strategy = strategy;
6     }
7
8     public int executeStrategy(int num1, int num2){
9         return strategy.doOperation(num1, num2);
10    }
11 }
```



Step 4: Use the *Context* to see change in behaviour when it changes its *Strategy*.

Output

```
PS E:\Strategy\Strategy> java StrategyPatternDemo
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
PS E:\Strategy\Strategy>
```

## Conclusion

This lab successfully demonstrates the implementation of the Strategy Pattern in Java. By encapsulating algorithms within interchangeable strategy classes, the design achieves flexibility and scalability. This pattern is especially useful for applications requiring dynamic behavior changes without modifying the context class.

