Heaven's Light Is Our Guide

Rajshahi University of Engineering & Technology
Department of Computer Science & Engineering

Course Title: Software Engineering Sessional
Course Code: CSE 3206

Lab Report 1

Submitted By

Roll: 2003124, 2003125, 2003126
Section: C
Session: 2020-2021

Submitted To

Farjana Parvin
Lecturer
Department of CSE, RUET

Submission Date: 07/01/2025

Design Pattern Used: Prototype Design Pattern

Definition: Prototype is a creational design pattern that lets you copy existing objects without making your code dependent on their classes.

Description of the Problem:

Let's say we have a program for managing books in a library. Each book has a title, an author, number of pages and price. Now, you need to create multiple books that are identical to an existing book.

Steps to Create an Exact Copy of an Object:

Step 1: Create a new object of the same class as the original.
Step 2: Manually go through all the fields of the original object.
Step 3: Copy the values from each field of the original object into the new object.

Problems with the Direct Copying Approach:

1) **Private Fields:** Some fields (like price) may be private, meaning they can't be accessed directly outside the object. This makes it harder to copy them manually.
2) **Dependency on the Object's Class:** You need to know the exact class of the object to create a duplicate. This creates a dependency on that specific class, making the code less flexible.
3) **Manual Copying:** Copying every detail (e.g., title, author, pages) manually is time-consuming and error-prone, especially when the object has many fields or private properties.
4) **Unknown Concrete Class:** Sometimes, you only know the interface that the object follows, not the exact class. For example, if a method accepts objects that follow a certain interface but you don't know their concrete class, you can't directly copy the object.

Description of the Solution:

1) **Cloning Objects**: The Prototype Pattern solves this problem by allowing you to clone an existing object. Instead of starting from scratch, you can make a copy of an object that already exists.
2) **Common Interface:** The prototype pattern provides a common interface that all objects can follow to support cloning. This interface usually has just one method: clone().
3) **How the clone() Method Works:** The clone() method creates a new object of the same class as the original. It copies all the values (properties) from the original object into the new one. Even private fields can be copied because objects of the same class can access each other's private fields.
4) **Example:** For example, in a Book object, instead of copying the title, author, and pages manually, you can call the clone() method. The new book will have the same details as the original, but it is a separate object. You can change the new book without affecting the original one.

Code Implementation:

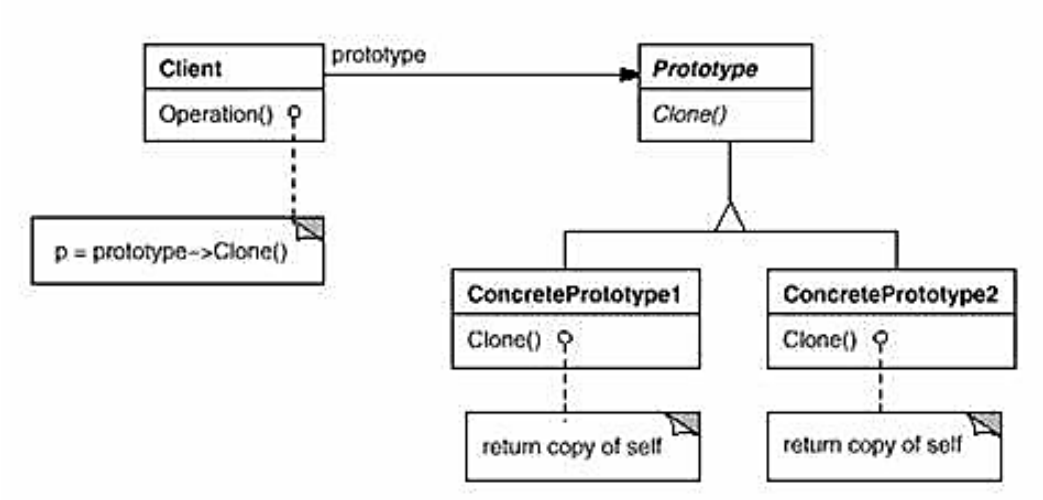
Book.java

```
public class Book
{
    String title;
    String author;
    int pages;
    private int price; // Private value

    // Constructor to create a new Book
    public Book(String title, String author, int pages, int price) {
        this.title = title;
        this.author = author;
        this.pages = pages;
        this.price = price;
    }

    // Clone method to make a copy of this book
    public Book clone()
      {  return new Book(this.title, this.author, this.pages, this.price); }

    // Method to display book details
    public void display() {
        System.out.println("Title: " + this.title);
        System.out.println("Author: " + this.author);
        System.out.println("Pages: " + this.pages);
        System.out.println("Price: " + this.price + " Taka"); // Display private value
        System.out.println();
    }
  public void updatePrice(int price) {
    this.price = price; }
}
```

Main.java

```
    public static void main(String[] args) {
        // Create a new book
        Book originalBook = new Book("The Alchemist", "Paulo Coelho", 208, 1050);

        // Display the original book
        System.out.println("Original Book:");
        originalBook.display();

        // Clone the original book
        Book clonedBook = originalBook.clone();

        // Display the cloned book
        System.out.println("Cloned Book:");
        clonedBook.display();

        // Change the title of the cloned book
        clonedBook.title = "The Alchemist - Special Edition";
        clonedBook.updatePrice(1200);

        // Display both books to show they are now different
        System.out.println("After changing the cloned book:");
        System.out.println("Original Book:");
        originalBook.display();
        System.out.println("Cloned Book:");
        clonedBook.display();
}
```

UML Diagram:



Output:

```
Original Book:
Title: The Alchemist
Author: Paulo Coelho
Pages: 208
Price: 1050 Taka

Cloned Book:
Title: The Alchemist
Author: Paulo Coelho
Pages: 208
Price: 1050 Taka

After changing the cloned book:
Original Book:
Title: The Alchemist
Author: Paulo Coelho
Pages: 208
Price: 1050 Taka



Cloned Book:
Title: The Alchemist - Special Edition
Author: Paulo Coelho
Pages: 208
Price: 1200 Taka

PS E:\0 3rd Even\Software Engineering\lab\prototype>
```

Design Pattern Used: Singleton Design Pattern

Problems Solved by the Singleton Pattern:

1. **Ensuring a Single Instance:**
    - Ensures that a class has only **one instance**.
    - This is useful for controlling access to shared resources like a **database** or a **file**.
    - Example: If you try to create a new instance, the pattern ensures you receive the already existing instance instead of creating a fresh one.
2. **Global Access to the Instance:**
    - Provides a **global access point** to the single instance of the class.
    - Similar to a **global variable**, but safer:
        - Prevents the instance from being accidentally overwritten by other parts of the code.
        - Ensures all parts of the program work with the same instance, maintaining consistency.
3. **Avoiding Scattered Code:**
    - Centralizes the logic for managing the single instance in one place (inside the Singleton class).
    - Reduces the need to duplicate logic across the program.

Issues with Regular Constructors:

- A regular constructor always returns a new object by design, making it impossible to restrict the number of instances.
- To solve this, the Singleton pattern **makes the constructor private**, ensuring no external class can directly create a new instance.

Solution:

1. **Restrict Instance Creation:**
    - Make the class's **default constructor private** to prevent using the new operator.
2. **Provide a Static Creation Method:**
    - Create a **static method** (e.g., getInstance()) that:
        - Calls the private constructor internally (if the instance doesn't already exist).
        - Stores the instance in a **static field**.
        - Returns the same instance for all subsequent calls.
3. **How It Works:**
    - When the getInstance() method is called for the first time:
        - The Singleton object is created and stored in a static field.
    - For all following calls:
        - The cached (already created) instance is returned.

CODE:

```
// Government class implementing Singleton
class Government {
    // Static field to hold the singleton instance
    private static Government instance;

    // Private constructor to prevent instantiation from outside
    private Government() {
        System.out.println("Government formed.");
    }

    // Public method to provide access to the singleton instance
    public static Government getInstance() {
        if (instance == null) { // Check if instance is null
            synchronized (Government.class) { // Ensure thread safety
                if (instance == null) { // Double-check locking
                    instance = new Government(); // Create the singleton instance
                }
            }
        }
        return instance;
    }

    // Example method to implement policies
    public void implementPolicy(String policy) {
        System.out.println("Policy implemented: " + policy);
    }
}

// Application class to demonstrate Singleton Government
public class Application {
    public static void main(String[] args) {
        // Access the singleton instance of Government
        Government gov1 = Government.getInstance();
        gov1.implementPolicy("Free education for all.");

        // Access the same singleton instance of Government
        Government gov2 = Government.getInstance();
        gov2.implementPolicy("Universal healthcare for all.");

        // Verify that both references point to the same instance
        System.out.println("gov1 and gov2 refer to the same government: " + (gov1 ==
gov2));
    }
}
```

OUTPUT

Government formed.
Policy implemented: Free education for all.
Policy implemented: Universal healthcare for all.
gov1 and gov2 refer to the same government: true

Adapter Design Pattern:

A structural design pattern is a blueprint in software development that focuses on how to organize and connect classes or objects to create larger, more efficient, and flexible systems. It ensures that components work well together while remaining easy to modify and reuse

The Adapter Design Pattern bridges the gap between incompatible interfaces, enabling them to work together. It involves:

- Target: The expected interface.
- Adaptee: The incompatible class.
- Adapter: A middleman class that converts the adaptee's interface to the target's interface.

This allows reuse of existing code without modifications.

Problem Description:

1. ***Data Format Mismatch**:*
    - *The stock market app processes data in **XML format**.*
    - *The 3rd-party analytics library only works with **JSON format**, creating a compatibility issue.*
2. ***Challenges in Modifying the Library**:*
    - *Directly modifying the library to support XML might:*
        - *Break existing code that relies on the library.*
        - *Be impossible if the library's source code is inaccessible.*
3. ***Need for a Flexible Solution**:*
    - *A solution is required to allow the app to use the analytics library without altering the library or its expected JSON format.*

Solution Description:

1. ***Using an Adapter Design Pattern**:*
    - *Create an **adapter** object that acts as a bridge between the app and the analytics library.*
    - *The adapter hides the complexity of format conversion from XML to JSON.*
2. ***Functionality of the Adapter**:*
    - *Wraps the analytics library's objects and methods.*
    - *Translates XML data from the app into JSON format before passing it to the library.*
    - *Receives responses from the library (if any) and converts them back if needed.*
3. ***Key Benefits**:*
    - *Ensures compatibility between the app and the analytics library without modifying either.*
    - *Keeps the app code clean and focused on its functionality while handling conversion behind the scenes.*
    - *Supports flexibility and scalability, as additional adapters can be created for other formats or libraries.*
4. ***Optional Enhancement**:*
    - *Implement a **two-way adapter** if the app and library need to exchange data in both directions.*

Code:

```
// RoundHole class
class RoundHole {
    private double radius;


    // Constructor
    public RoundHole(double radius) {
        this.radius = radius;
    }


    // Getter for the radius
    public double getRadius() {
        return radius;
    }


    // Method to check if a peg fits into the hole
    public boolean fits(RoundPeg peg) {
        return this.radius >= peg.getRadius();
    }
}


// RoundPeg class
class RoundPeg {
    private double radius;


    // Default constructor
    public RoundPeg() {}


    // Constructor
    public RoundPeg(double radius) {
        this.radius = radius;
    }


    // Getter for the radius
    public double getRadius() {
        return radius;
    }
}


// SquarePeg class
class SquarePeg {
    private double width;


    // Constructor
    public SquarePeg(double width) {
        this.width = width;
    }


    // Getter for the width
    public double getWidth() {
        return width;
```

```
    }
}


// SquarePegAdapter class
class SquarePegAdapter extends RoundPeg {
    private SquarePeg squarePeg;


    // Constructor
    public SquarePegAdapter(SquarePeg squarePeg) {
        this.squarePeg = squarePeg;
    }


    // Override getRadius() to calculate the effective radius
    @Override
    public double getRadius() {
        // Calculate and return the effective radius of the square peg
        return squarePeg.getWidth() * Math.sqrt(2) / 2;
    }
}


// Client code
public class AdapterPatternDemo {
    public static void main(String[] args) {
        // Create a RoundHole with radius 5
        RoundHole hole = new RoundHole(5);


        // Create a RoundPeg with radius 5
        RoundPeg roundPeg = new RoundPeg(5);


        // Check if the round peg fits into the round hole
        System.out.println("Round peg fits: " + hole.fits(roundPeg)); // true


        // Create square pegs
        SquarePeg smallSquarePeg = new SquarePeg(5);
        SquarePeg largeSquarePeg = new SquarePeg(10);


        // Try fitting square pegs into the round hole (will not work directly)
        // System.out.println(hole.fits(smallSquarePeg)); // This line won't compile!


        // Use adapters for square pegs
        SquarePegAdapter smallSquarePegAdapter = new SquarePegAdapter(smallSquarePeg);
        SquarePegAdapter largeSquarePegAdapter = new SquarePegAdapter(largeSquarePeg);


        // Check if the adapted square pegs fit into the round hole
        System.out.println("Small square peg fits: " +
hole.fits(smallSquarePegAdapter)); // true
        System.out.println("Large square peg fits: " +
hole.fits(largeSquarePegAdapter)); // false
    }
}
```

Output

```
Round peg fits: true
Small square peg fits: true
Large square peg fits: false

=== Code Execution Successful ===
```