Design Patterns Laboratory Report

Factory & Singleton Patterns

Mohamed Amine El Bacha

Course: software engineering
Institution: UM6P - college of computing

October 29, 2025

Contents

1 Introduction

This laboratory report presents the implementation and analysis of two fundamental design patterns in software engineering: the Singleton pattern and the Factory pattern. These patterns offer solutions to common design problems in object-oriented programming, promoting code reusability, maintainability, and scalability.

The Singleton pattern ensures that a class has only one instance throughout the application lifecycle, while the Factory pattern provides an interface for creating objects without specifying their exact classes. This report demonstrates practical implementations of both patterns through three progressive exercises.

2 Exercise 1: Singleton Pattern - Database Connection

2.1 Objective

Implement a database connection manager using the Singleton design pattern to ensure only one instance of the database exists throughout the application.

2.2 Implementation

2.2.1 Database Class (Singleton)

```
public class Database {
    // volatile to ensure visibility across threads
    private static volatile Database instance;
    private String name;

    // private constructor to prevent direct instantiation
    private Database(String name) {
        this.name = name;
    }

    // getInstance implements thread-safe lazy singleton
    public static Database getInstance(String name) {
        // first check (non-synchronized) to improve performance
        if (instance == null) {
            synchronized (Database.class) { // thread synchronization
                // second check to avoid creating multiple instances
                if (instance == null) {
                    instance = new Database(name);
                }
            }
        }
        return instance;
    }

    // simple method simulating a connection
    public void getConnection() {
        System.out.println("You are connected to database: " + name);
    }

```

```
30      public static void main(String[] args) {
31          // attempt to create two instances with different names
32          Database db1 = Database.getInstance("database1");
33          Database db2 = Database.getInstance("database2");
34
35          // verifying both variables reference the same instance
36          if (db1 == db2) {
37              System.out.println("The instances are matched");
38          } else {
39              System.out.println("The instances are not matched");
40          }
41
42          // calling instance method using both references
43          db1.getConnection();
44          db2.getConnection();
45      }
46  }
```

Listing 1: Database.java - Singleton Implementation

Explanation: The Database class implements the Singleton pattern using a private constructor and a static getInstance() method. The private constructor prevents direct instantiation, ensuring that objects can only be created through the getInstance() method. The static instance variable holds the single instance of the class, which is created only once when first requested. Subsequent calls to getInstance() return the same instance regardless of the name parameter passed, demonstrating the singleton behavior.

2.2.2 Program Output

```
The instances are matched
You are connected to database: database1
You are connected to database: database1
```

Listing 2: Output of Database.java

3 Exercise 2: Factory Pattern - Program Selection

3.1 Part 1: Naive Solution

3.1.1 Program1 Class

```
1  public class Program1 {
2      public Program1() {
3          // Constructor does nothing
4      }
5
6      public void go() {
7          System.out.println("Je suis le traitement 1");
8      }
9  }
```

Listing 3: Program1.java - Original Implementation

Explanation: Program1 is a simple class with an empty constructor and a go() method that prints a message identifying itself as treatment 1. This serves as the base implementation that will be extended by Program2 and Program3.

3.1.2 Program2 and Program3 Classes

```
public class Program2 {
    public Program2() {
        // Constructor does nothing
    }

    public void go() {
        System.out.println("Je suis le traitement 2");
    }
}
```

Listing 4: Program2.java

Explanation: Program2 follows the same structure as Program1 but displays a different message. Program3 (not shown) follows identical pattern with message "Je suis le traitement 3". This demonstrates code duplication that will be addressed by the factory pattern.

3.1.3 Client Class (Naive Implementation)

```
import java.util.Scanner;

public class client {
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        System.out.println("enter the number of the program (1, 2, 3) :
    ");
        int n = input.nextInt();
        switch (n){
            case 1 :{
                program1 p = new program1();
                p.go();
                break;
            }case 2 :{
                program2 p = new program2();
                p.go();
                break;
            }case 3 :{
                program3 p = new program3();
                p.go();
                break;
            } default:{
                System.out.println("Invalid input");
            }
        }
    }
}
```

Listing 5: Client.java - Naive Solution

Explanation: The naive client implementation uses switch statements to link the user input to the creation of the corresponding program. This approach has several problems, such as code duplication when creating different programs. When we want to modify a line in the client class, we need to modify the entire class, which can cause issues in the whole code.

3.1.4 Program Output

```
enter the number of the program (1, 2, 3) :
1
Je suis le traitement 1
```

Listing 6: Output when entering 1

3.2 Problems with Naive Solution

The naive solution exhibits several issues: code duplication across the entire switch statement, tight coupling making the system rigid, low maintainability requiring client modifications for new programs, violation of the Open-Closed Principle, and poor scalability.

3.3 Part 2: Factory Pattern Solution

3.3.1 Program Interface

```
1 public interface Program {
2     void go();
3 }
```

Listing 7: Program.java - Interface

Explanation: The Program interface defines the blueprint that all the program classes that will be created, this helps the client to work with pram without knowing their types

3.3.2 Refactored Program Classes

```
1 public class program1 implements programmingInterface {
2     // the constructor does nothing
3     public program1(){
4     }
5
6     public void go(){
7         System.out.println("je suis le traitement 1");
8     }
9 }
```

Listing 8: Program1.java - Implementing Interface

Explanation: Program1 now override the go() methode. program2 and program3 follows the same logic.

3.3.3 ProgramFactory Class

```
public class ProgramFactory {
    public ProgramFactory() {}
    static void createProgram(int index){
        switch (index) {
            case 1 : {
                program1 p = new program1();
                p.go();
            } case 2 : {
                program2 p = new program2();
                p.go();
            } case 3 : {
                program3 p = new program3();
                p.go();
            } default : {
                System.out.println("Invalid input");
            }
        }
    }
}
```

Listing 9: ProgramFactory.java

Explanation: The ProgramFactory class centralizes creation logic using a static factory method. The createProgram() method takes an integer and returns the appropriate Program implementation. This encapsulates object creation, making client code simpler and more maintainable.

3.3.4 Refactored Client Class

```
import java.util.Scanner;

public class newClient {
    public static void main(String[] args) {
        // take the user input
        Scanner input = new Scanner(System.in);
        System.out.print(" enter the number of the program : ");
        int num = input.nextInt();
        // create the wanted programs
        ProgramFactory pf = new ProgramFactory();
        pf.createProgram(num);
    }
}
```

Listing 10: Client.java - Using Factory

Explanation: The refactored client is cleaner and more maintainable. Each method uses the factory to create program instances instead of directly instantiating concrete classes. This eliminates duplication and reduces coupling.

3.3.5 Program Output

```
 enter the number of the program : 2
je suis le traitement 2
```

Listing 11: Output of newClient.java

3.3.6 Adding Program4

```java
public class program4 implements programmingInterface {
    // the constructor does nothing
    public program4(){
    }

    public void go(){
        System.out.println("je suis le traitement 4");
    }
}
```

Listing 12: Program4.java

Explanation: Adding Program4 demonstrates extensibility. The new class implements the Program interface following the same structure.

```java
public class ProgramFactory {
    public ProgramFactory() {}
    static void createProgram(int index){
        switch (index) {
            case 1 : {
                program1 p = new program1();
                p.go();
            } case 2 : {
                program2 p = new program2();
                p.go();
            } case 3 : {
                program3 p = new program3();
                p.go();
            } case 4 : {
            // we only add those lines of code
                program4 p = new program4();
                p.go();
            }default: {
                System.out.println("Invalid input");
            }
        }
    }
}
```

Listing 13: ProgramFactory.java - Updated

Explanation: Adding Program4 only requires modifying the ProgramFactory by adding one case. The Client class remains unchanged, demonstrating the Open-Closed Principle.

4 Exercise 3: Monster Battle Game

4.1 Objective

Develop a turn-based monster battle game combining Singleton and Factory patterns to manage game state and monster creation.

4.2 Implementation

4.2.1 GameManager (Singleton)

```java
public class GameManager {
    private static volatile GameManager instance;
    private GameManager() {
    }
    public static GameManager getInstance() {
        if (instance == null){
            synchronized (GameManager.class){
                if (instance == null){
                    instance = new GameManager();
                }
            }
        }
        return instance;
    }

    public void getGame() {
        System.out.println("============================");
        System.out.println("WELCOME TO MONSTER BATTLE");
        System.out.println("============================\n");
    }

    static public void main(String[] args){
        GameManager gm1 = GameManager.getInstance();
        GameManager gm2 = GameManager.getInstance();

        if (gm1 == gm2) {
            System.out.println("The instances are matched");
        } else {
            System.out.println("The instances are not matched");
        }
    }
}
```

Listing 14: GameManager.java

Explanation: GameManager implements Singleton to ensure one game state exists. It manages player scores and provides methods for game operations. The private constructor prevents external instantiation.

4.2.2 Program Output

```
The instances are matched
```

Listing 15: Output of GameManager.java

4.2.3 Monster Interface

```java
public interface MonsterInterface {
    public void attack();
    public int getHealth();
    public void takeDamage(int damage);
}
```

Listing 16: Monster.java

Explanation: The Monster interface defines the contract for all monster types, including methods for combat, health management, and status checking.

4.2.4 Dragon Class

```
public class Dragon implements Monster {
    private int health;
    private int attackBehavior;
    private String name;


    public Dragon() {
        this.name = "Dragon";
        this.health = 150;
        this.attackBehavior = 30;
        System.out.println("Dragon entered! (HP: " + health + ")");
    }


    public void attack(Monster opponent) {
        System.out.println(name + " breathes fire!");
        opponent.takeDamage(attackBehavior);
    }


    public int getHealth() {
        return health;
    }

    public void takeDamage(int damage) {
        health -= damage;
        if (health < 0) health = 0;
        System.out.println(name + " takes " + damage
                + " damage! (HP: " + health + ")");
    }

    public String getName() {
        return name;
    }

    public boolean isAlive() {
        return health > 0;
    }
}
```

Listing 17: Dragon.java

Explanation: Dragon represents a powerful monster with high health and strong attacks. It implements all Monster interface methods with thematic combat messages.

4.2.5 Goblin and Wizard Classes

```java
public class Goblin implements Monster {


    private int health = 80;
    private int attackBehavior = 15;
    private String name = "Goblin";

    public Goblin() {
        this.name = "Goblin";
        this.health = 150;
        this.attackBehavior = 15;
        System.out.println("Goblin entered! (HP: " + health + ")");
    }


    public void attack(Monster opponent) {
        System.out.println(name + " strikes with dagger!");
        opponent.takeDamage(attackBehavior);
    }

    // Other methods similar to Dragon


    public int getHealth() { return health; }


    public void takeDamage(int damage) {
        health -= damage;
        if (health < 0) health = 0;
        System.out.println(name + " takes " + damage
                + " damage! (HP: " + health + ")");
    }


    public String getName() { return name; }

    public boolean isAlive() { return health > 0; }
}
```

Listing 18: Goblin.java

Explanation: Goblin is a balanced fighter with moderate stats. Wizard (not shown in full) follows similar pattern with 100 HP and 25 attack power, using spell-casting attacks.

4.2.6 MonsterFactory Class

```java
public class MonsterFactory {
    public static Monster createMonster(String type) {
        switch (type.toLowerCase()) {
            case "dragon":
                return new Dragon();
            case "goblin":
                return new Goblin();
            case "wizard":
                return new Wizard();
            default:
```

```
                throw new IllegalArgumentException(
                        "Unknown monster: " + type);
        }
    }

    public static void displayMonsters() {
        System.out.println("\nAvailable Monsters:");
        System.out.println("1. Dragon - HP:150, ATK:30");
        System.out.println("2. Goblin - HP:80, ATK:15");
        System.out.println("3. Wizard - HP:100, ATK:25");
    }
}
```

Listing 19: MonsterFactory.java

Explanation: MonsterFactory encapsulates monster creation using the Factory pattern. The createMonster() method handles instantiation based on string input, making it easy to add new monster types.

4.2.7 GameClient Class

```
import java.util.Scanner;

public class GameClient {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Get singleton and start game
        GameManager gm = GameManager.getInstance();
        gm.startGame();

        // Display monsters
        MonsterFactory.displayMonsters();

        // Choose monsters
        System.out.print("\nPlayer 1 monster: ");
        String c1 = scanner.nextLine();
        Monster m1 = MonsterFactory.createMonster(c1);

        System.out.print("Player 2 monster: ");
        String c2 = scanner.nextLine();
        Monster m2 = MonsterFactory.createMonster(c2);

        System.out.println("\n=== BATTLE BEGINS ===\n");

        // Battle loop
        int turn = 1;
        while (m1.isAlive() && m2.isAlive()) {
            System.out.println("--- Turn " + turn + " ---");

            // Player 1 attacks
            System.out.println("Player 1's turn:");
            m1.attack(m2);

            if (!m2.isAlive()) break;

            System.out.println();
```

```
37
38              // Player 2 attacks
39              System.out.println("Player 2's turn:");
40              m2.attack(m1);
41
42              System.out.println("\n--- Status ---");
43              System.out.println("P1 " + m1.getName()
44                  + ": " + m1.getHealth() + " HP");
45              System.out.println("P2 " + m2.getName()
46                  + ": " + m2.getHealth() + " HP\n");
47
48              turn++;
49          }
50
51          // Announce winner
52          System.out.println("\n=== BATTLE ENDED ===");
53          if (m1.isAlive()) {
54              System.out.println("Player 1 wins!");
55              gm.updateScore(1);
56          } else {
57              System.out.println("Player 2 wins!");
58              gm.updateScore(2);
59          }
60
61          gm.displayScores();
62          scanner.close();
63      }
64  }
```

Listing 20: GameClient.java

Explanation: GameClient orchestrates the battle using both design patterns. It obtains the GameManager singleton, uses MonsterFactory for creation, implements turn-based combat, and manages game flow with user interaction.

4.2.8 Program Output

```
============================
WELCOME TO MONSTER BATTLE
============================

Available Monsters:
1. Dragon - HP:150, ATK:30
2. Goblin - HP:80, ATK:15
3. Wizard - HP:100, ATK:25

Player 1 monster: dragon
Dragon entered! (HP: 150)
Player 2 monster: goblin
Goblin entered! (HP: 150)

=== BATTLE BEGINS ===

--- Turn 1 ---
Player 1's turn:
Dragon breathes fire!
Goblin takes 30 damage! (HP: 120)
```

```
Player 2's turn:
Goblin strikes with dagger!
Dragon takes 15 damage! (HP: 135)

--- Status ---
P1 Dragon: 135 HP
P2 Goblin: 120 HP

--- Turn 2 ---
Player 1's turn:
Dragon breathes fire!
Goblin takes 30 damage! (HP: 90)

Player 2's turn:
Goblin strikes with dagger!
Dragon takes 15 damage! (HP: 120)

--- Status ---
P1 Dragon: 120 HP
P2 Goblin: 90 HP

[Battle continues...]

=== BATTLE ENDED ===
Player 1 wins!

--- Scores ---
Player 1: 1
Player 2: 0
```

Listing 21: Sample Output of GameClient.java

5 Conclusion

This laboratory successfully implemented and demonstrated two essential design patterns. The Singleton pattern provided centralized state management in the Database and GameManager classes, ensuring consistency across the application. The Factory pattern decoupled object creation from client code in both the Program selection system and Monster creation, making the codebase more maintainable and extensible.

Key learnings include understanding when to apply each pattern, recognizing code smells that indicate pattern needs, and appreciating how patterns improve code quality. The Monster Battle Game combined both patterns effectively, showcasing their practical application in a real-world scenario.