Design Patterns: Factory & Singleton

Lab Report

Lazrek Nassim

October 29, 2025

Contents

1 Exercise 1: Singleton Pattern - Database Connection

1.1 Objective

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

1.2 Implementation

```
class Exercise1 {
    static class Database {
        private static Database instance = null;
        private String name;

        private Database() {
            System.out.println("Constructor called");
            this.name = "DefaultDB";
        }

        private Database(String name) {
            this.name = name;
            System.out.println("Constructor called with name: " + name);
        }

        public void getConnection() {
            System.out.println("You are connected to the database: "
                + this.name);
        }

        public static Database getInstance() {
            if (instance == null) {
                instance = new Database();
            }
            return instance;
        }

        public static Database getInstance(String name) {
            if (instance == null) {
                instance = new Database(name);
            }
            return instance;
        }

        public String getName() {
            return name;
        }
    }

    public static void run() {
        System.out.println("========== EXERCISE 1: SINGLETON PATTERN
    ==========\n");

        Database db1 = Database.getInstance("ProductionDB");
        Database db2 = Database.getInstance("TestDB");

        System.out.println("\n--- Testing Singleton ---");
        System.out.println("db1 reference: " + db1);
        System.out.println("db2 reference: " + db2);
        System.out.println("db1 == db2 ? " + (db1 == db2));
        System.out.println("identityHashCode(db1): "
            + System.identityHashCode(db1));
        System.out.println("identityHashCode(db2): "
```

```
53              + System.identityHashCode(db2));
54
55          System.out.println("\n--- Connection Tests ---");
56          db1.getConnection();
57          db2.getConnection();
58
59          System.out.println("\n--- Name Verification ---");
60          System.out.println("db1 name: " + db1.getName());
61          System.out.println("db2 name: " + db2.getName());
62
63          if (db1 == db2) {
64              System.out.println("\ n   SUCCESS! Only one database instance
    exists.");
65          }
66      }
67  }
```

Listing 1: Database Singleton Implementation

1.3 Output

```
1  ========== EXERCISE 1: SINGLETON PATTERN ==========
2
3  Constructor called with name: ProductionDB
4
5  --- Testing Singleton ---
6  db1 reference: Exercise1$Database@14ae5a5
7  db2 reference: Exercise1$Database@14ae5a5
8  db1 == db2 ? true
9  identityHashCode(db1): 21685669
10 identityHashCode(db2): 21685669
11
12 --- Connection Tests ---
13 You are connected to the database: ProductionDB
14 You are connected to the database: ProductionDB
15
16 --- Name Verification ---
17 db1 name: ProductionDB
18 db2 name: ProductionDB
19
20 ? SUCCESS! Only one database instance exists.
```

Listing 2: Exercise 1 Output

1.4 Analysis

Key Observations

1. **Single Instance**: Both `db1` and `db2` reference the same object (same memory address).
2. **Identity Verification**: The `identityHashCode` values are identical, confirming they're the same instance.
3. The first call to `getInstance("ProductionDB")` creates the instance, and subsequent calls return this instance regardless of the name parameter.
4. **Private Constructor**: Prevents direct instantiation using `new Database()`.

1.5 Answers to Questions

Q: What happens when we try to create two databases with different names?

A: Only the first database name is used. The Singleton pattern ensures that once an instance is created with a specific name, all subsequent attempts to get an instance (even with different names) return the same original instance. This demonstrates the core principle of the Singleton pattern: exactly one instance is created entirely.

2 Exercise 2: Factory Pattern - Program Creation

2.1 Objective

Implement a Factory pattern to eliminate code duplication when creating different program objects and to provide a centralized creation mechanism.

2.2 Part 1: Naive Solution

The naive approach involves duplicating the object creation code across multiple methods:

```
public static void main1() {
    Program1 p = new Program1();
    System.out.println("I am main1");
    p.go();
}

public static void main2() {
    Program2 p = new Program2();
    System.out.println("I am main2");
    p.go();
}

public static void main3() {
    Program3 p = new Program3();
    System.out.println("I am main3");
    p.go();
}
```

Listing 3: Naive Solution - Code Duplication

Problems with Naive Solution:

- Code duplication across multiple methods
- Difficult to maintain and extend (adding Program4 requires modifying Client)

2.3 Part 2: Factory Pattern Solution

```
class Exercise2 {
    // Interface that all programs must implement
    interface Program {
        void go();
    }

    // Program implementations
    static class Program1 implements Program {
        public Program1() {
            System.out.println("A program was initialized.");
        }

        public void go() {
            System.out.println("Je suis le traitement 1");
        }
    }

    static class Program2 implements Program {
        public Program2() {
            System.out.println("A program was initialized.");
        }

        public void go() {
```

```
            System.out.println("Je suis le traitement 2");
        }
    }

    static class Program3 implements Program {
        public Program3() {
            System.out.println("A program was initialized.");
        }

        public void go() {
            System.out.println("Je suis le traitement 3");
        }
    }

    // Factory class to create Program objects
    static class ProgramFactory {
        public static Program createProgram(int programNumber) {
            switch (programNumber) {
                case 1: return new Program1();
                case 2: return new Program2();
                case 3: return new Program3();
                case 4: return new Program4();
                default:
                    throw new IllegalArgumentException(
                        "Invalid program number: " + programNumber);
            }
        }
    }

    // Simplified Client class
    static class Client {
        public static void main(int programNumber) {
            System.out.println("C'est main" + programNumber);
            Program p = ProgramFactory.createProgram(programNumber);
            p.go();
        }
    }

    public static void run(int num) {
        System.out.println("========== EXERCISE 2: FACTORY PATTERN ==========\n
    ");
        Client.main(num);
    }
}
```

Listing 4: Factory Pattern Implementation

2.4 Class Diagram

2.5 Output Examples

```
========== EXERCISE 2: FACTORY PATTERN ==========

C'est main1
A program was initialized.
Je suis le traitement 1
```

Listing 5: Exercise 2 - Program 1 Output

```
========== EXERCISE 2: FACTORY PATTERN ==========

C'est main2
A program was initialized.
Je suis le traitement 2
```

Listing 6: Exercise 2 - Program 2 Output

2.6 Answers to Questions

Q1: Did you duplicate the code for creating Program objects?

A: No, the Factory pattern eliminates code duplication by centralizing object creation in the `ProgramFactory.createProgram()` method. Instead of three separate methods each creating objects, we have a single factory method that handles all creation logic.

Q2: Can you add a Program4? Was it complicated to implement?

A: Yes, adding Program4 is straightforward:

1. Create the Program4 class implementing the Program interface
2. Add a case in the factory's switch statement
3. No changes needed to the Client code

Q3: Did you have to modify the Client code?

A: No, the Client code remains unchanged when adding new program types. This is a key benefit of the Factory pattern - it decouples the client from concrete implementations, making the system more maintainable and extensible.

3 Exercise 3: Monster Battle Game

3.1 Objective

Build a turn-based monster battle game that combines both Singleton and Factory patterns to manage game state and monster creation.

3.2 System Architecture

Design Patterns Applied

- **Singleton Pattern**: GameManager ensures single game state instance
- **Factory Pattern**: MonsterFactory creates different monster types
- **Interface**: Monster interface defines contract for all monsters

3.3 Class Diagram

SEE THE DRAW.io FILE PRESENT IN THE SAME FOLDER.

3.4 Implementation

3.4.1 Part 1: GameManager (Singleton)

```
static class GameManager {
    private static GameManager instance = null;
    private int player1Score = 0;
    private int player2Score = 0;

    private GameManager() {
        System.out.println("GameManager instance created.");
    }

    public static GameManager getInstance() {
        if (instance == null) {
            instance = new GameManager();
        }
        return instance;
    }

    public void startGame() {
        System.out.println("

    ");
        System.out.println("      WELCOME TO MONSTER BATTLE GAME!
        ");
        System.out.println("

    ");
    }

    public void updateScore(String player) {
        if (player.equals("Player 1")) {
            player1Score++;
        } else if (player.equals("Player 2")) {
            player2Score++;
        }
    }

```

```
    public void displayScores() {
        System.out.println("\n--- Current Scores ---");
        System.out.println("Player 1: " + player1Score);
        System.out.println("Player 2: " + player2Score);
    }
}
```

Listing 7: GameManager Singleton

3.4.2 Part 2: Monster Interface and Implementations

```
interface Monster {
    void attack(Monster opponent);
    int getHealth();
    void takeDamage(int damage);
    String getName();
    boolean isAlive();
}

static abstract class AbstractMonster implements Monster {
    protected String name;
    protected int health;
    protected int attackPower;

    public AbstractMonster(String name, int health, int attackPower) {
        this.name = name;
        this.health = health;
        this.attackPower = attackPower;
    }

    @Override
    public int getHealth() { return health; }

    @Override
    public String getName() { return name; }

    @Override
    public void takeDamage(int damage) {
        health -= damage;
        if (health < 0) health = 0;
    }

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

    @Override
    public void attack(Monster opponent) {
        System.out.println(name + " attacks " + opponent.getName()
            + " for " + attackPower + " damage!");
        opponent.takeDamage(attackPower);
    }
}

static class Dragon extends AbstractMonster {
    public Dragon() {
        super("Dragon", 150, 30);
        System.out.println("A mighty Dragon has been summoned! (HP: 150, ATK:
    30)");
    }

    @Override
    public void attack(Monster opponent) {
```

```
        System.out.println(name + " breathes fire at "
            + opponent.getName() + " for " + attackPower + " damage!       ");
        opponent.takeDamage(attackPower);
    }
}
```

Listing 8: Monster Interface and Dragon Class

3.4.3 Monster Statistics

Monster	Health Points	Attack Power	Special Ability
Dragon	150	30	Fire Breath
Goblin	80	15	Dagger Strike
Wizard	100	25	Magic Spell

3.4.4 Part 3: MonsterFactory

```
static 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: " +
    type);
        }
    }
}
```

Listing 9: MonsterFactory Implementation

3.4.5 Part 4: GameClient

```
static class GameClient {
    public static void runGame() {
        Scanner scanner = new Scanner(System.in);

        GameManager gameManager = GameManager.getInstance();
        gameManager.startGame();

        System.out.println("\nAvailable monsters: Dragon, Goblin, Wizard");

        System.out.print("\nPlayer 1, choose your monster: ");
        String choice1 = scanner.nextLine();
        Monster monster1 = MonsterFactory.createMonster(choice1);

        System.out.print("\nPlayer 2, choose your monster: ");
        String choice2 = scanner.nextLine();
        Monster monster2 = MonsterFactory.createMonster(choice2);

        System.out.println("\n" + "=".repeat(50));
        System.out.println("BATTLE BEGINS!");
        System.out.println("=".repeat(50) + "\n");

        int round = 1;
```

```
        while (monster1.isAlive() && monster2.isAlive()) {
            System.out.println("--- Round " + round + " ---");

            monster1.attack(monster2);
            System.out.println(monster2.getName() + " HP: " + monster2.
    getHealth());

            if (!monster2.isAlive()) break;

            System.out.println();

            monster2.attack(monster1);
            System.out.println(monster1.getName() + " HP: " + monster1.
    getHealth());

            System.out.println();
            round++;
        }

        System.out.println("=".repeat(50));
        if (monster1.isAlive()) {
            System.out.println("       PLAYER 1 WINS with " + monster1.getName
    () + "!       ");
            gameManager.updateScore("Player 1");
        } else {
            System.out.println("       PLAYER 2 WINS with " + monster2.getName
    () + "!       ");
            gameManager.updateScore("Player 2");
        }
        System.out.println("=".repeat(50));

        gameManager.displayScores();
    }
}
```

Listing 10: GameClient - Battle System

3.5 Sample Output

```
========== EXERCISE 3: MONSTER BATTLE GAME ==========

GameManager instance created.


      WELCOME TO MONSTER BATTLE GAME!



Available monsters: Dragon, Goblin, Wizard

Player 1, choose your monster: A mighty Dragon has been summoned! (HP: 150, ATK
    : 30)

Player 2, choose your monster: A sneaky Goblin has appeared! (HP: 80, ATK: 15)

==================================================
BATTLE BEGINS!
==================================================

--- Round 1 ---
Dragon breathes fire at Goblin for 30 damage!
Goblin HP: 50
```

```
21
22 Goblin strikes with a dagger at Dragon for 15 damage!
23 Dragon HP: 135
24
25 --- Round 2 ---
26 Dragon breathes fire at Goblin for 30 damage!
27 Goblin HP: 20
28
29 Goblin strikes with a dagger at Dragon for 15 damage!
30 Dragon HP: 120
31
32 --- Round 3 ---
33 Dragon breathes fire at Goblin for 30 damage!
34 Goblin HP: 0
35 ================================================
36        PLAYER 1 WINS with Dragon!
37 ================================================
38
39 --- Current Scores ---
40 Player 1: 1
41 Player 2: 0
```

Listing 11: Exercise 3 - Battle Output (Dragon vs Goblin)