Design Patterns: Factory & Singleton

Solutions to Exercises

SAFAR Fatima Ezzahra
UM6P - Mohammed VI Polytechnic University

October 29, 2025

Contents

1 Exercise 1: Singleton Design Pattern

1.1 Problem Statement

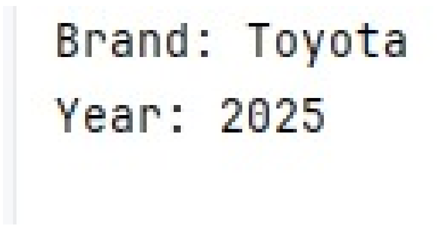
Create a Java program implementing the Singleton design pattern for a database connection. The database should have a `name` attribute and a `getConnection()` method.

1.2 Implementation

1.2.1 Database Class (Singleton)

```java
public class Database {
    private String name;
    private static Database instance;

    private Database(String name) {
        this.name = name;
    }

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

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

    public String getName() {
        return name;
    }
}
```

Listing 1: Database.java - Singleton Implementation



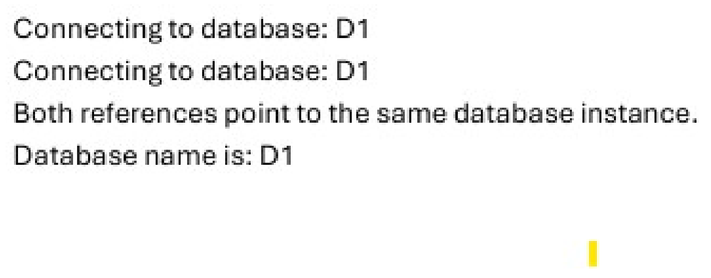
```
Brand: Toyota
Year: 2025
```

Figure 1: Database Class Implementation

1.2.2 Main Class - Testing Singleton

```java
class Main {
    public static void main(String[] args) {
        Database d1 = Database.getInstance("D1");
        Database d2 = Database.getInstance("D2");

        d1.getConnection();
        d2.getConnection();

        if (d1 == d2) {
            System.out.println("Both references point to the
               same database instance.");
        } else {
            System.out.println("Different instances (Singleton
                not working).");
        }

        System.out.println("Database name is: " + d1.getName()
           );
    }
}
```

Listing 2: Main.java - Singleton Test



```
Connecting to database: D1
Connecting to database: D1
Both references point to the same database instance.
Database name is: D1
```

Figure 2: Main Class - Testing Singleton

1.3 Key Observations

Singleton Key Points

- The constructor is **private**, preventing direct instantiation
- Only **one instance** is created throughout the application
- Both `d1` and `d2` refer to the **same object**
- The database name remains "D1" because it's set during the first instantiation

2 Exercise 2: Factory Design Pattern

2.1 Problem Statement

Implement a Factory pattern to manage the creation of Program objects. This eliminates code duplication and provides a centralized creation mechanism.

2.2 Implementation

2.2.1 Program Interface

```java
public interface Program {
    void go();
}
```

Listing 3: Program.java - Interface

```java
public class Program1 implements Program {
    public void go() {
        System.out.println("Je suis le traitement 1");
    }
}
```

Listing 4: Program2.java - First Implementation

```java
public class Program2 implements Program {
    public void go() {
        System.out.println("Je suis le traitement 2");
    }
}
```

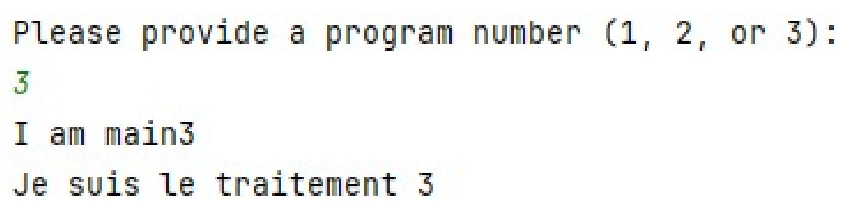
Listing 5: Program2.java - Second Implementation

```java
public class Program3 implements Program {
    public void go() {
        System.out.println("Je suis le traitement 3");
    }
}
```

Listing 6: Program3.java - Third Implementation

```java
import java.util.Scanner;

public class Client {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Please provide a program number (1,
           2, or 3): ");
        int choice = sc.nextInt();

        Program p;

        if (choice == 1)
            p = new Program1();
        else if (choice == 2)
            p = new Program2();
        else if (choice == 3)
            p = new Program3();
        else {
            System.out.println("Invalid choice!");
            return;
        }

        System.out.println("I am main" + choice);
        p.go();
    }
}
```

Listing 7: Program3.java - Third Implementation



```
Please provide a program number (1, 2, or 3):
3
I am main3
Je suis le traitement 3
```

Figure 3: Program Implementations (Program1, Program2, Program3)

2.2.2 What Do You Notice? Problems with Naive Solution

Problems with Naive Approach

Problem 1: Violation of Open/Closed Principle

- The `Client` depends directly on all program classes
- Each time you add a new `Program4`, you **must modify** the `Client` class
- This violates the **Open/Closed Principle (OCP)**: Classes should be *open for extension, closed for modification*

Problem 2: Complex Conditional Logic

- Too many `if-else` or `switch` statements
- Becomes messy and hard to maintain if there are many programs
- Code duplication in the switch structure

Problem 3: Tight Coupling

- The `Client` knows too much about each `Program` implementation
- Direct instantiation: `new Program1()`, `new Program2()`, etc.
- Hard to test, maintain, and extend

Problem 4: Code Duplication

- The pattern of creating objects is repeated
- Every place that needs to create a Program must duplicate this logic
- Violates the **DRY** (Don't Repeat Yourself) principle

2.2.3 Factory Class

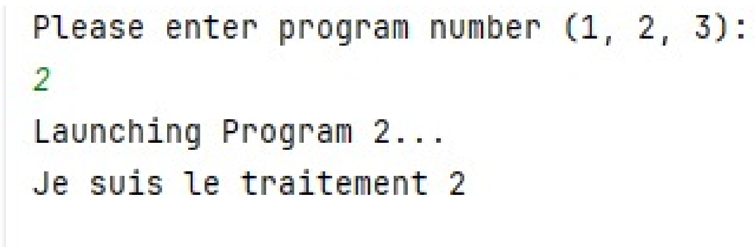
```java
public class ProgramFactory {
    public static Program createProgram(int choice) {
        switch (choice) {
            case 1:
                return new Program1();
            case 2:
                return new Program2();
            case 3:
                return new Program3();
            default:
                return null;
        }
    }
}
```

Listing 8: ProgramFactory.java - Factory Implementation

2.2.4 Client Class with Factory

```java
public class Client {
    public static void main1() {
        Program p = ProgramFactory.createProgram(1);
        System.out.println("I am main1");
        p.go();
    }

    public static void main2() {
        Program p = ProgramFactory.createProgram(2);
        System.out.println("I am main2");
        p.go();
    }

    public static void main3() {
        Program p = ProgramFactory.createProgram(3);
        System.out.println("I am main3");
        p.go();
    }
}
```

Listing 9: Client.java - Using Factory Pattern



```
Please enter program number (1, 2, 3):
2
Launching Program 2...
Je suis le traitement 2
```

Figure 4: Client Class Using Factory

2.3 Complete Class Diagram

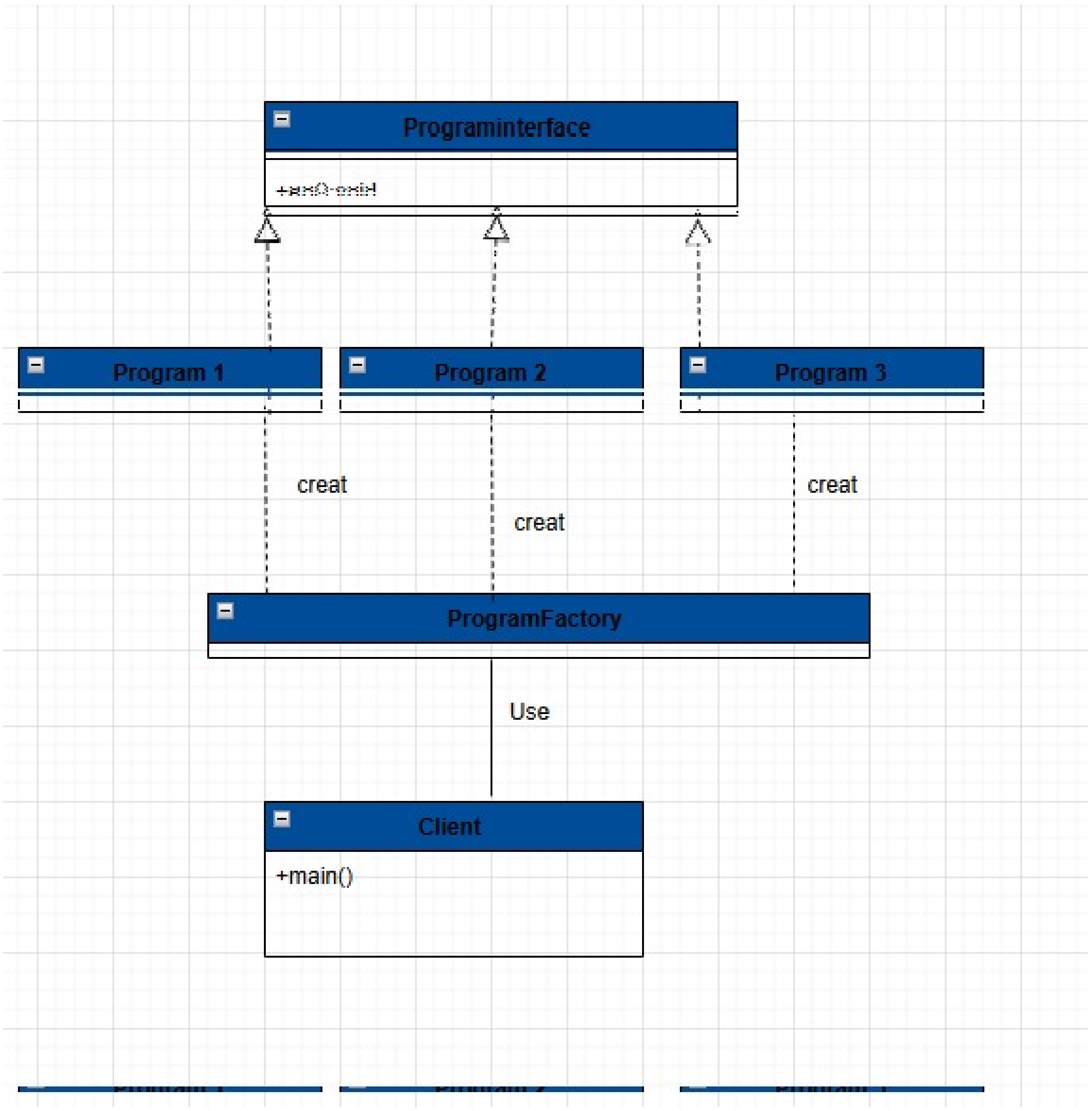


Figure 5: Factory Pattern - Complete Class Diagram

2.4 Advantages of Factory Pattern

Benefits of Factory Pattern

- **Centralized Object Creation**: All object creation logic is in one place
- **No Code Duplication**: Client code doesn't repeat instantiation logic
- **Easy Extensibility**: Adding Program4 only requires:
  1. Creating the new Program4 class
  2. Adding a new case in the factory
  3. NO modification to existing client code
- **Loose Coupling**: Client depends on the interface, not concrete classes
- **Open/Closed Principle**: Open for extension, closed for modification

2.5 Adding Program4

```java
public class Program4 implements Program {
    public void go() {
        System.out.println("Je suis le traitement 4");
    }
}
```

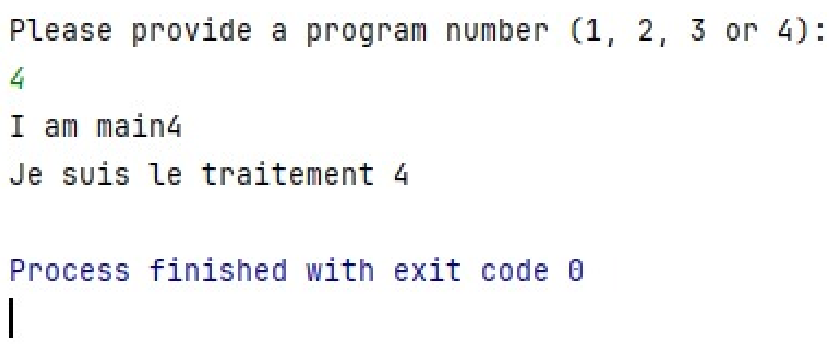
Listing 10: Program4.java - Easy Extension

```java
public class ProgramFactory {
    public static Program createProgram(int choice) {
        switch (choice) {
            case 1:
                return new Program1();
            case 2:
                return new Program2();
            case 3:
                return new Program3();
            case 4:
                return new Program4();  // Only this line
                    added!
            default:
                return null;
        }
    }
}
```

Listing 11: Updated ProgramFactory.java

```java
import java.util.Scanner;

public class Client {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Please provide a program number (1,
           2,3 or 4): ");
        int choice = sc.nextInt();

        Program p;

        if (choice == 1)
            p = new Program1();
        else if (choice == 2)
            p = new Program2();
        else if (choice == 3)
            p = new Program3();
        else if (choice == 4)
            p = new Program4();
        else {
            System.out.println("Invalid choice!");
            return;
        }

        System.out.println("I am main" + choice);
        p.go();
    }
}
```

Listing 12: Updated Client.java



```
Please provide a program number (1, 2, 3 or 4):
4
I am main4
Je suis le traitement 4

Process finished with exit code 0
```

Figure 6: Adding Program4 - No Client Changes Required

Here's a revised version that transforms the bullet points into flowing paragraphs: —

3 Comparison: Before vs After Factory Pattern

3.1 Before Factory Pattern (Naive Solution)

Problems with Naive Approach

The naive approach to object creation suffers from several critical design flaws. The most apparent issue is the extensive code duplication that occurs within the Client class, where instantiation logic such as `new ProgramX()` is repeated throughout the codebase. This repetition creates tight coupling between the Client and the concrete classes it instantiates, making the system rigid and difficult to modify. The lack of abstraction violates the DRY (Don't Repeat Yourself) principle, leading to maintenance nightmares where a single change might require modifications in multiple locations. Furthermore, the architecture makes it nearly impossible to extend the system gracefully—adding new program types necessitates invasive changes to the Client code itself, increasing the risk of introducing bugs and making the codebase increasingly fragile over time.

3.2 After Factory Pattern

Solutions with Factory Pattern

The Factory Pattern elegantly resolves these issues by introducing a centralized, well-defined approach to object creation. By establishing a single point of instantiation, the pattern eliminates code duplication and provides a consistent interface for creating objects throughout the application. The use of interfaces and abstract classes promotes loose coupling, allowing the Client to depend on abstractions rather than concrete implementations. This architectural decision dramatically improves maintainability and testability, as components can be easily mocked, extended, or replaced without rippling changes across the system. The pattern naturally aligns with SOLID principles, particularly the Open/Closed Principle, which states that software entities should be open for extension but closed for modification. Most importantly, when new program types need to be added to the system, they can be integrated seamlessly without touching the Client code—the factory handles the complexity of instantiation while the Client remains blissfully unaware of the underlying implementation details.

4 Exercise 3: Monster Battle Game

4.1 Problem Statement

Build a Monster Battle Game using Singleton and Factory design patterns with:

- **GameManager** (Singleton) controlling game state
- **Three Monster Types**: Dragon, Goblin, Wizard
- **Turn-based Combat** system with health and damage
- **MonsterFactory** for creating monsters

4.2 Complete Class Diagram

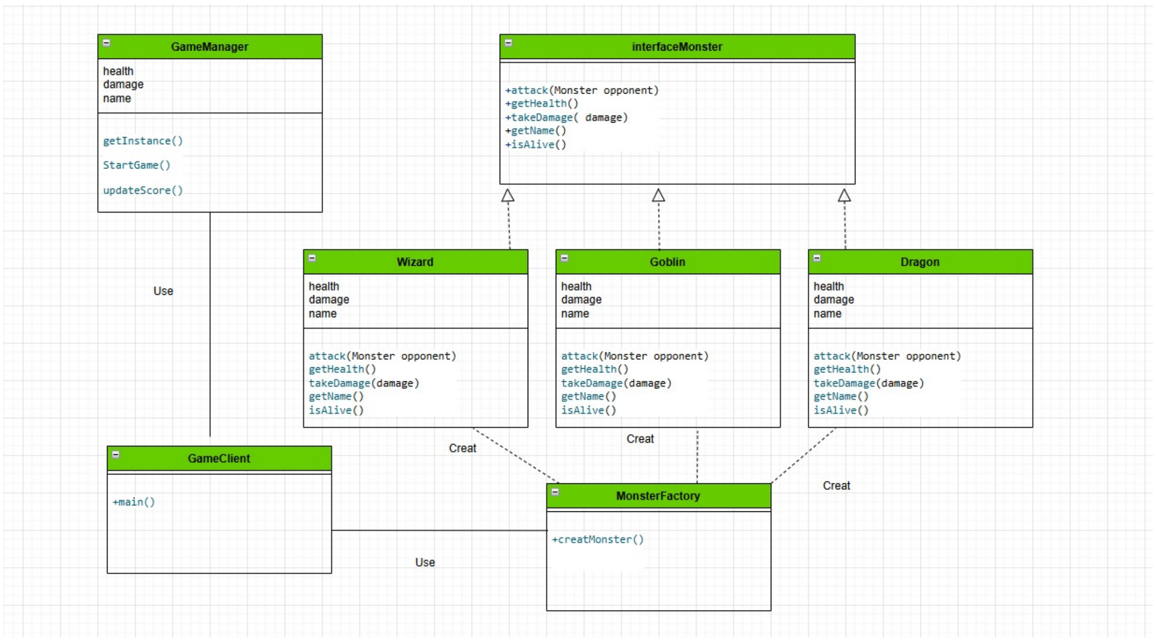


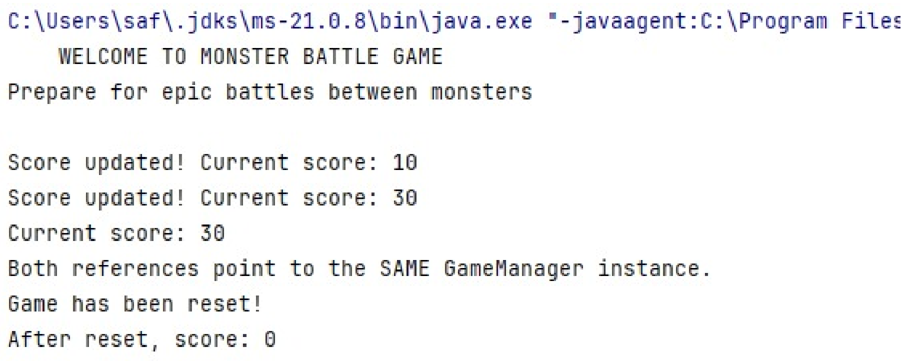
Figure 7: Monster Battle Game - Complete UML Class Diagram

4.3 Part 1: Game Manager (Singleton)

4.3.1 GameManager Implementation

```java
public class GameManager {
    private static GameManager instance;
    private int playerScore;

    // Private constructor
    private GameManager() {
        this.playerScore = 0;
    }

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

    public void startGame() {

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

        System.out.println("Prepare for epic battles between monsters");
        System.out.println();
    }


    public void updateScore(int points) {
        playerScore += points;
        System.out.println("Score updated! Current score: " + playerScore);
    }

    public int getScore() {
        return playerScore;
    }


    public void resetGame() {
        playerScore = 0;
        System.out.println("Game has been reset!");
    }
}
```

Listing 13: GameManager.java - Singleton Game Controller



```
C:\Users\saf\.jdks\ms-21.0.8\bin\java.exe "-javaagent:C:\Program Files
    WELCOME TO MONSTER BATTLE GAME
Prepare for epic battles between monsters

Score updated! Current score: 10
Score updated! Current score: 30
Current score: 30
Both references point to the SAME GameManager instance.
Game has been reset!
After reset, score: 0
```

Figure 8: GameManager Class - Singleton Implementation

4.4 Part 2: Monster System

4.4.1 Monster Interface

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

Listing 14: Monster.java - Monster Interface

4.4.2 Dragon Monster

```java
public class Dragon implements Monster {
    private int health;
    private final int attackDamage = 35;
    private final String name = "Dragon";

    public Dragon() {
        this.health = 150;
    }


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


    public int getHealth() {
        return health;
    }


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


    public String getName() {
        return name;
    }


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

Listing 15: Dragon.java - Dragon Implementation

4.4.3 Goblin Monster

```java
public class Goblin implements Monster {
    private int health;
    private final int attackDamage = 20;
    private final String name = "Goblin";

    public Goblin() {
        this.health = 80;
    }


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


    public int getHealth() {
        return health;
    }

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


    public String getName() {
        return name;
    }


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

Listing 16: Goblin.java - Goblin Implementation

4.4.4 Wizard Monster

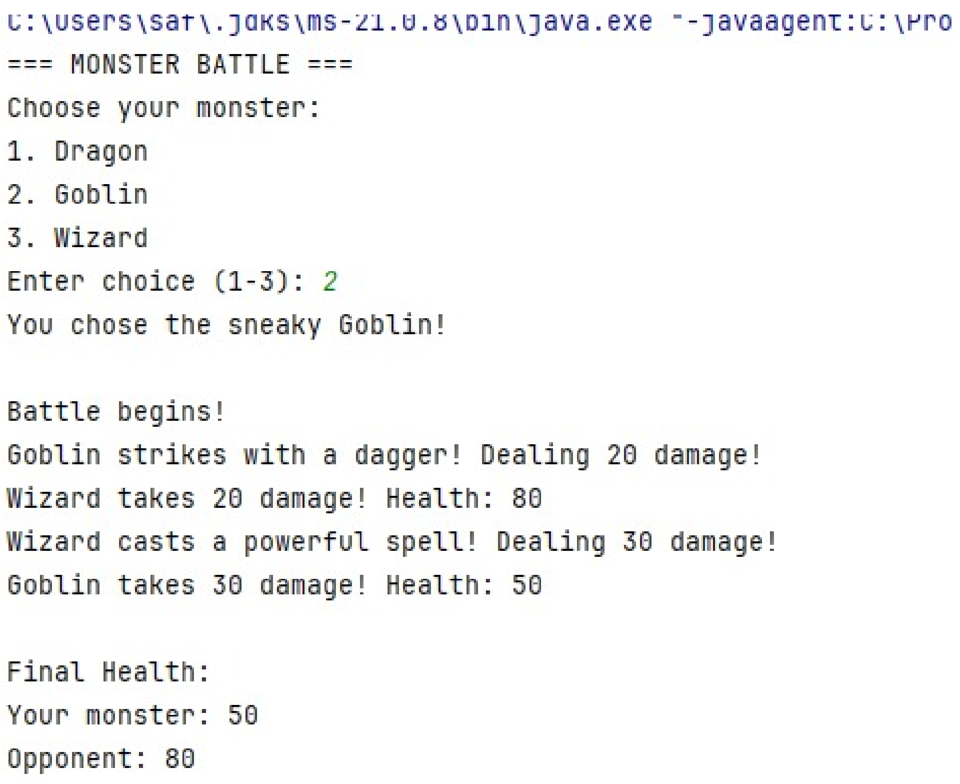
```java
public class Wizard implements Monster {
    private int health;
    private final int attackDamage = 30;
    private final String name = "Wizard";

    public Wizard() {
        this.health = 100;
    }

    public void attack(Monster opponent) {
        System.out.println(name + " casts a powerful spell!
           Dealing " +
                           attackDamage + " damage!");
        opponent.takeDamage(attackDamage);
    }


    public int getHealth() {
        return health;
    }


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


    public String getName() {
        return name;
    }


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

Listing 17: Wizard.java - Wizard Implementation

4.5 Monster Factory

```java
public class MonsterFactory {

    public static Monster createMonster(String type) {
        if (type == null || type.isEmpty()) {
            return null;
        }

        switch (type.toUpperCase()) {
            case "DRAGON":
                return new Dragon();
            case "GOBLIN":
                return new Goblin();
            case "WIZARD":
                return new Wizard();
            default:
                System.out.println("Unknown monster type: " +
                   type);
                return null;
        }
    }

    public static void displayAvailableMonsters() {
        System.out.println("\n Available Monsters ");
        System.out.println("1. DRAGON - Health: 150, Attack:
           35");
        System.out.println("2. GOBLIN - Health: 80, Attack: 20
           ");
        System.out.println("3. WIZARD - Health: 100, Attack:
           30");
        System.out.println("\n");
    }
}
```

Listing 18: MonsterFactory.java - Factory for Creating Monsters



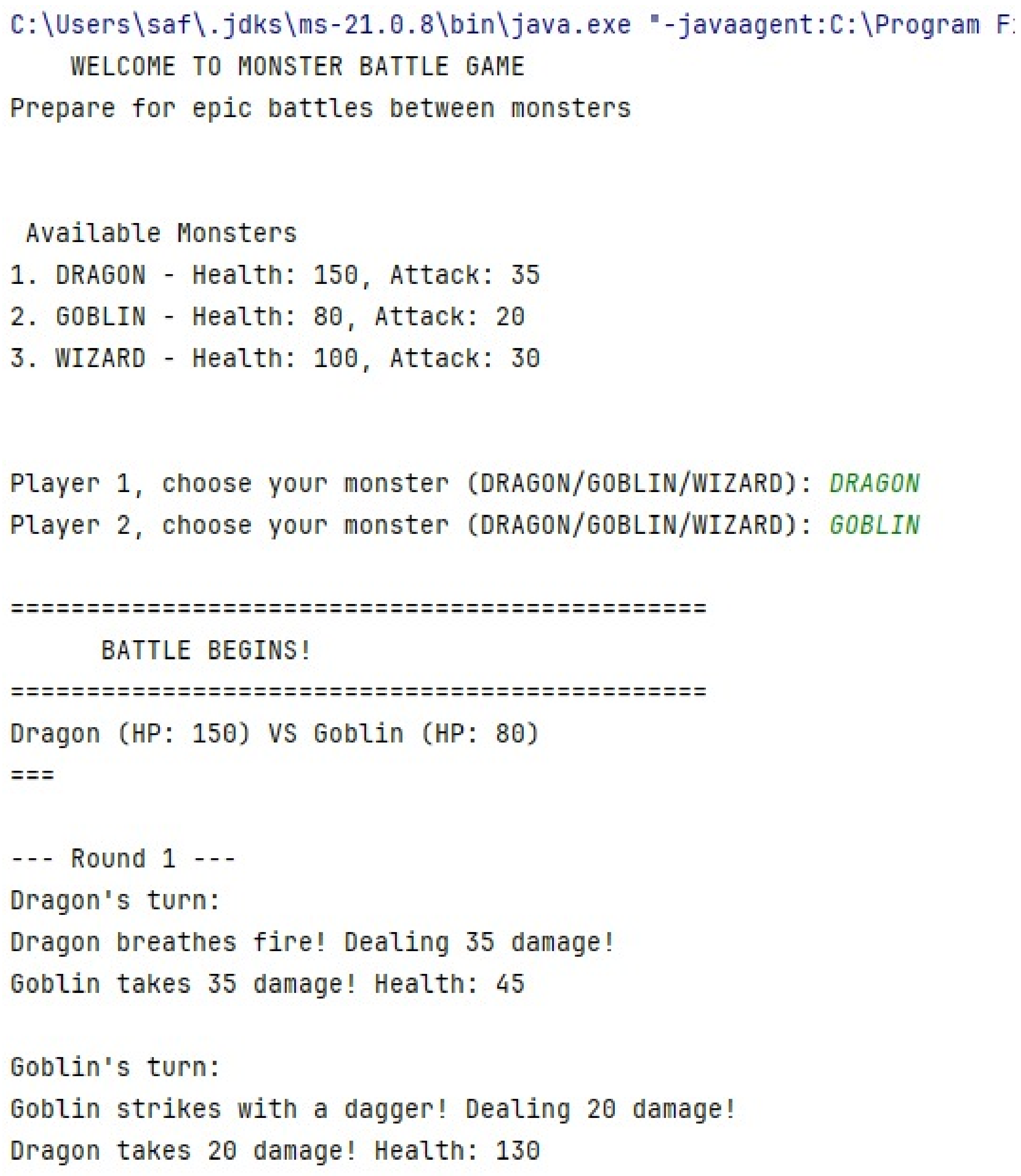
```
C:\Users\saf\.jdks\ms-21.0.8\bin\java.exe "-javaagent:C:\Pro
=== MONSTER BATTLE ===
Choose your monster:
1. Dragon
2. Goblin
3. Wizard
Enter choice (1-3): 2
You chose the sneaky Goblin!

Battle begins!
Goblin strikes with a dagger! Dealing 20 damage!
Wizard takes 20 damage! Health: 80
Wizard casts a powerful spell! Dealing 30 damage!
Goblin takes 30 damage! Health: 50

Final Health:
Your monster: 50
Opponent: 80
```

Figure 9: the Main of part 2

4.6 Part 3: Game Client - Battle Interaction

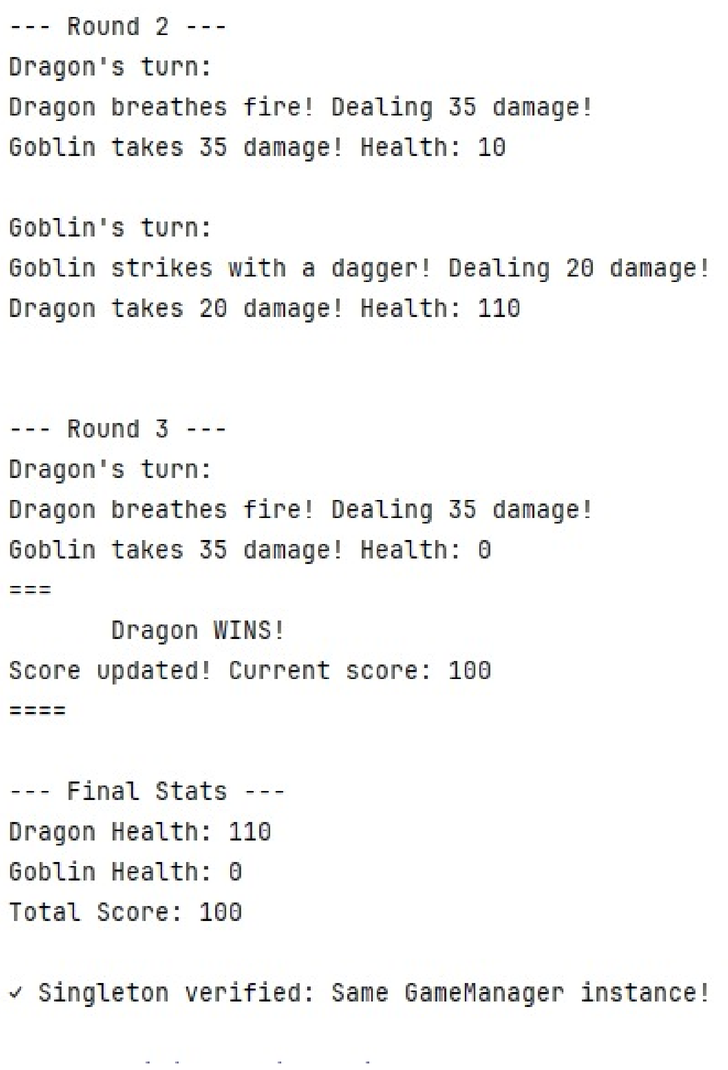
```java
import java.util.Scanner;

public class GameClient {

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

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

        MonsterFactory.displayAvailableMonsters();


        System.out.print("Player 1, choose your monster (DRAGON/GOBLIN/WIZARD): ");
        String choice1 = scanner.nextLine();
        Monster monster1 = MonsterFactory.createMonster(choice1);

        if (monster1 == null) {
            System.out.println("Invalid choice! Exiting...");
            return;
        }

        System.out.print("Player 2, choose your monster (DRAGON/GOBLIN/WIZARD): ");
        String choice2 = scanner.nextLine();
        Monster monster2 = MonsterFactory.createMonster(choice2);

        if (monster2 == null) {
            System.out.println("Invalid choice! Exiting...");
            return;
        }

        System.out.println("\n==============================================");
        System.out.println("      BATTLE BEGINS!");
        System.out.println("==============================================");
        System.out.println(monster1.getName() + " (HP: " + monster1.getHealth() +
                           ") VS " + monster2.getName() + " (HP: " +
                           monster2.getHealth() + ")");
        System.out.println("===\n");

        int round = 1;
        while (monster1.isAlive() && monster2.isAlive()) {
            System.out.println("--- Round " + round + " ---");

            System.out.println(monster1.getName() + "'s turn:");
            monster1.attack(monster2);

            if (!monster2.isAlive()) {
                break;
            }

            System.out.println();

            // Monster 2 attacks
            System.out.println(monster2.getName() + "'s turn:");
            monster2.attack(monster1);

            System.out.println("\n");
            round++;
        }


        System.out.println("===");
        if (monster1.isAlive()) {
            System.out.println("        " + monster1.getName() + " WINS!");
            gameManager.updateScore(100);
        } else {
            System.out.println("        " + monster2.getName() + " WINS!");
            gameManager.updateScore(100);
        }
        System.out.println("====");

        System.out.println("\n--- Final Stats ---");
        System.out.println(monster1.getName() + " Health: " + monster1.getHealth());
        System.out.println(monster2.getName() + " Health: " + monster2.getHealth());
        System.out.println("Total Score: " + gameManager.getScore());


        GameManager gm2 = GameManager.getInstance();
        if (gameManager == gm2) {
            System.out.println("\n   Singleton verified: Same GameManager instance!");
        }

        scanner.close();
        }
}
```

Listing 19: GameClient.java - Main Game Loop



```
C:\Users\saf\.jdks\ms-21.0.8\bin\java.exe "-javaagent:C:\Program F
    WELCOME TO MONSTER BATTLE GAME
Prepare for epic battles between monsters


 Available Monsters
1. DRAGON - Health: 150, Attack: 35
2. GOBLIN - Health: 80, Attack: 20
3. WIZARD - Health: 100, Attack: 30


Player 1, choose your monster (DRAGON/GOBLIN/WIZARD): DRAGON
Player 2, choose your monster (DRAGON/GOBLIN/WIZARD): GOBLIN

==============================================
      BATTLE BEGINS!
==============================================
Dragon (HP: 150) VS Goblin (HP: 80)
===

--- Round 1 ---
Dragon's turn:
Dragon breathes fire! Dealing 35 damage!
Goblin takes 35 damage! Health: 45

Goblin's turn:
Goblin strikes with a dagger! Dealing 20 damage!
Dragon takes 20 damage! Health: 130
```

Figure 10: GameClient - Main Game Implementation PART 3



```
--- Round 2 ---
Dragon's turn:
Dragon breathes fire! Dealing 35 damage!
Goblin takes 35 damage! Health: 10

Goblin's turn:
Goblin strikes with a dagger! Dealing 20 damage!
Dragon takes 20 damage! Health: 110


--- Round 3 ---
Dragon's turn:
Dragon breathes fire! Dealing 35 damage!
Goblin takes 35 damage! Health: 0
===
       Dragon WINS!
Score updated! Current score: 100
====

--- Final Stats ---
Dragon Health: 110
Goblin Health: 0
Total Score: 100

✓ Singleton verified: Same GameManager instance!
```

Figure 11: GameClient - Main Game Implementation PART 3 THE END