Design Patterns TP

Younes Menfalouti

Exercise 1: Singleton Database

In the main program, the uniqueness of the database instance was tested by attempting to create two databases with different names. Both references were shown to point to the same instance.

```java
public class Database {  7 usages
    private static Database database = null;  4 usages
    String name;  2 usages
    private Database() {  1 usage
    }
    public void getConnection() {  no usages
        System.out.println("You are connected to the database " + this.name);
    }
    public static Database getInstance(String name) {  2 usages
        if (database == null) {
            database = new Database();
        }
        database.name = name;
        return database;
    }
}
class MainSingleton {
    public static void main(String[] args) {
        Database X = Database.getInstance( name: "MySQL");
        Database Y = Database.getInstance( name: "SQLite");

        System.out.println(X==Y);
```

```
MainSingleton ×
"C:\Program Files\Java\jdk-25\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\I
true
```

Figure 1: Exercise1 Screenshot

Exercise 2: Program Factory

Naive Solution

The initial implementation included classes `Program1`, `Program2`, and `Program3`, each directly interacting with client code. The `Client` code has to be modified to launch the appropriate program.

This solution leads to tight coupling repeated changes in the client code which is not efficient.

Applying Design Patterns

A `ProgramFactory` class was introduced to delegate the creation of program objects. This improved maintainability.

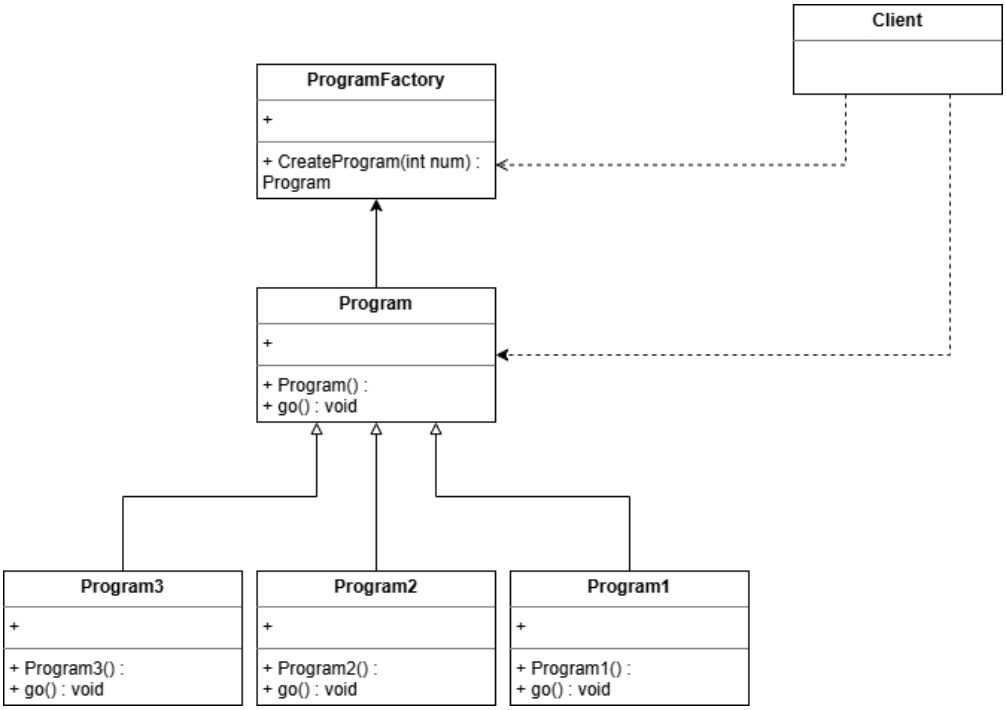

Figure 2: Exercise2 Class Diagram

Adding a new `Program4` class was straightforward and did not require changes to the client code, demonstrating the flexibility of the Factory pattern.

Example of code execution :

Existing Program :

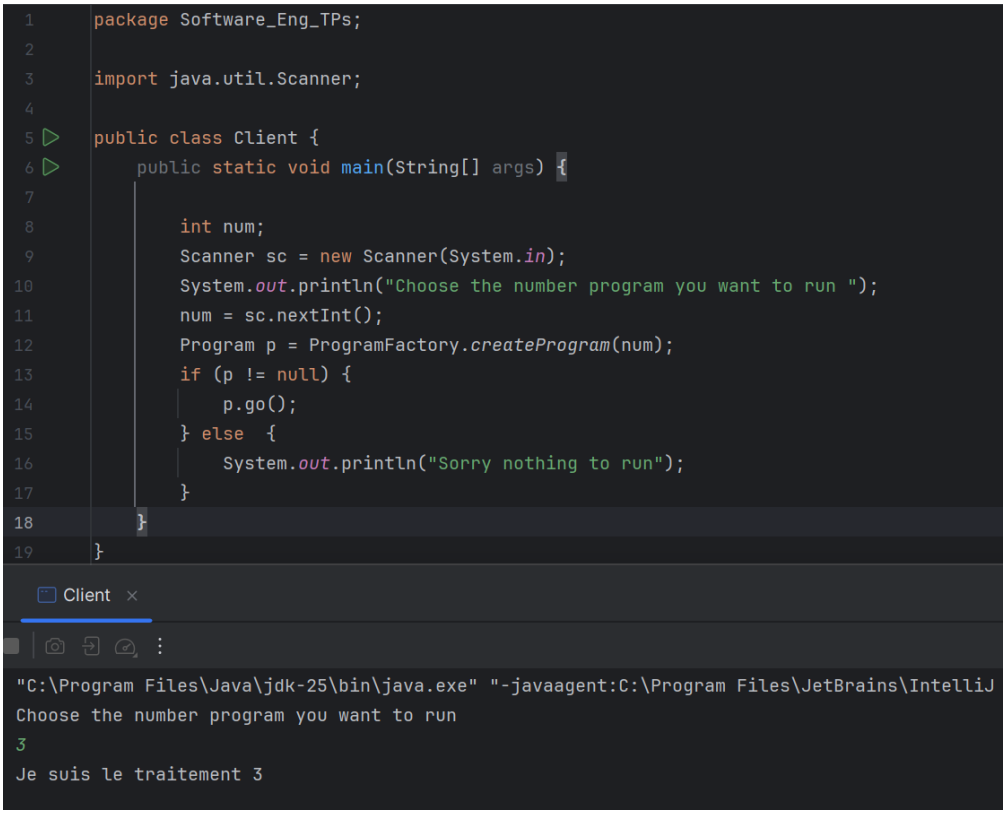

Figure 3: Exercise2 Screenshot

Non Existing Program :

```java
package Software_Eng_TPs;

import java.util.Scanner;

public class Client {
    public static void main(String[] args) {

        int num;
        Scanner sc = new Scanner(System.in);
        System.out.println("Choose the number program you want to run ");
        num = sc.nextInt();
        Program p = ProgramFactory.createProgram(num);
        if (p != null) {
            p.go();
        } else  {
            System.out.println("Sorry nothing to run");
        }
    }
}
```

```
Client
"C:\Program Files\Java\jdk-25\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ
Choose the number program you want to run
5
Sorry nothing to run
```

Figure 4: Exercise2 Screenshot

Exercise 3: Monster Battle Game

A turn-based monster battle game was implemented using the Singleton and Factory design patterns.

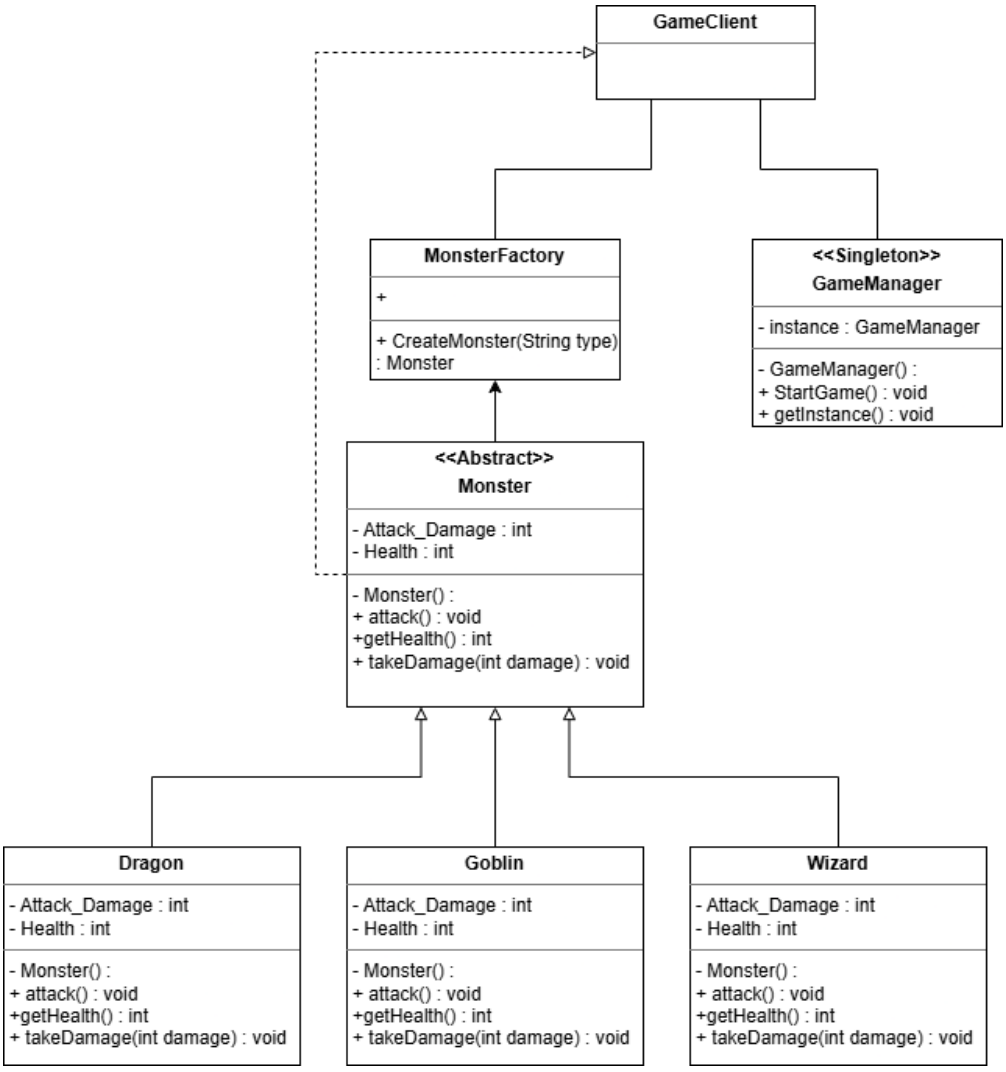

Figure 5: Exercise3 Class Diagram

Part 1: Game Manager

The `GameManager` class was implemented as a Singleton to ensures only one instance exists.

Figure 6: Exercise3 Screenshot

Part 2: Monsters

Three monster types—Dragon, Goblin, and Wizard—were implemented, each with unique health points and attack behaviors. All monsters implement the methods: `attack()`, `getHealth()`, and `takeDamage(int damage)`.

```java
package Software_Eng_TPs.Game;

public class MonsterFactory {  2 usages
    static Monster CreateMonster(String Type) {  2 usages
        if (Type.equals("Dragon")) {
            return new Dragon( attack_dmg: 100, health: 800);
        }
        else if (Type.equals("Goblin")) {
            return new Goblin( attack_dmg: 90, health: 1000);
        }
        else if (Type.equals("Wizard")) {
            return new Wizard( attack_dmg: 200, health: 500);
        }
        else {
            return null;
        }
    }
}
```

Figure 7: Exercise3 Screenshot

Part 3: Game Client

The `GameClient` class allows the user to choose two monsters, which are created via the `MonsterFactory`. The game alternates attacks until one monster is defeated, displaying health after each Round.

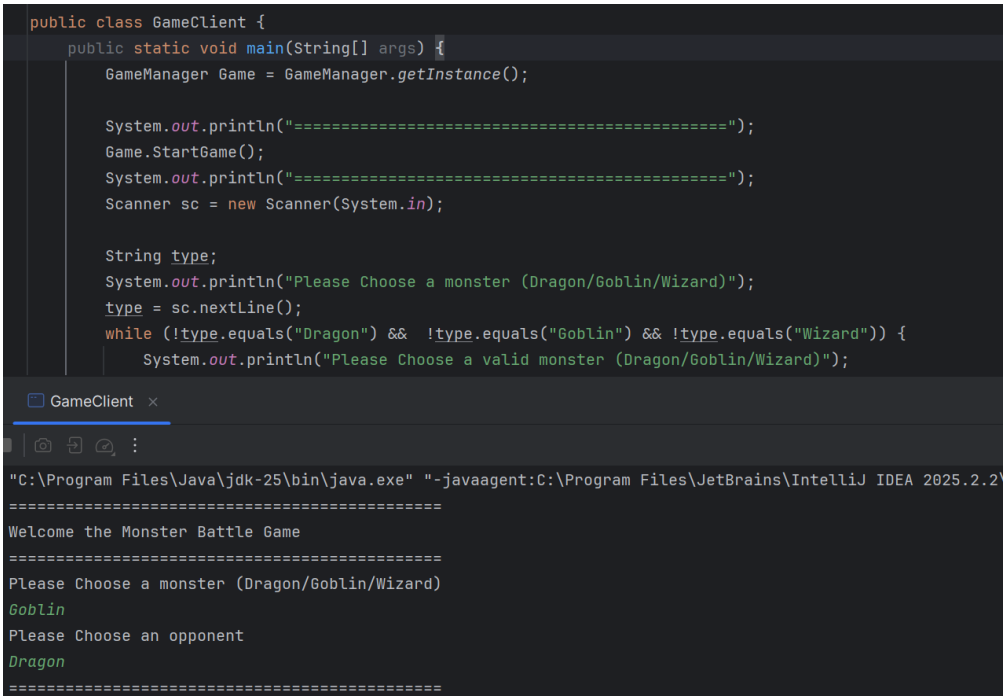

Figure 8: Exercise3 Screenshot

The monster's hits could either land or not which makes it less dependent on the type of monsters the user chooses :

```
Fight is starting

================================================
============== ROUND 1===============
It's Goblin's turn to attack
The attack did not land !
It's Dragon's turn to attack
The attack landed !
Attacking The Opponent
============== END OF ROUND 1===============
Results :

Goblin's remaining health : 900

Dragon's remaining health : 800

============== ROUND 2===============
It's Goblin's turn to attack
The attack landed !
Attacking The Opponent
It's Dragon's turn to attack
The attack did not land !
============== END OF ROUND 2===============
Results :

Goblin's remaining health : 900
```

Figure 9: Exercise3 Screenshot

In the end only one monster will survive :

```
============== ROUND 16===============
It's Goblin's turn to attack
The attack did not land !
It's Dragon's turn to attack
The attack did not land !
============== END OF ROUND 16===============
Results :

Goblin's remaining health : 100

Dragon's remaining health : 350

============== ROUND 17===============
It's Goblin's turn to attack
The attack did not land !
It's Dragon's turn to attack
The attack landed !
Attacking The Opponent
============== END OF BATTLE ===============
Goblin's remaining health : 0

Dragon's remaining health : 350

Your opponent Dragon Has won the battle !
```

Figure 10: Exercise3 Screenshot