Design Patterns: Factory and Singleton

KAYA Youssef

1. Exercise 1: Singleton Pattern

1.1. Problem Description

We want to implement a **Database** class such that only one instance can ever be created. This instance represents a connection to a single database, identified by its `name` attribute and method `getConnection()` displaying a message.

1.2. Implementation

Here is the Java code:

```java
public class Database {
    public 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 + ".");
    }
}
```

Figure 1: Singleton Database Implementation

1.3. Testing the Singleton

In the `main` method, we attempt to create two different database objects:

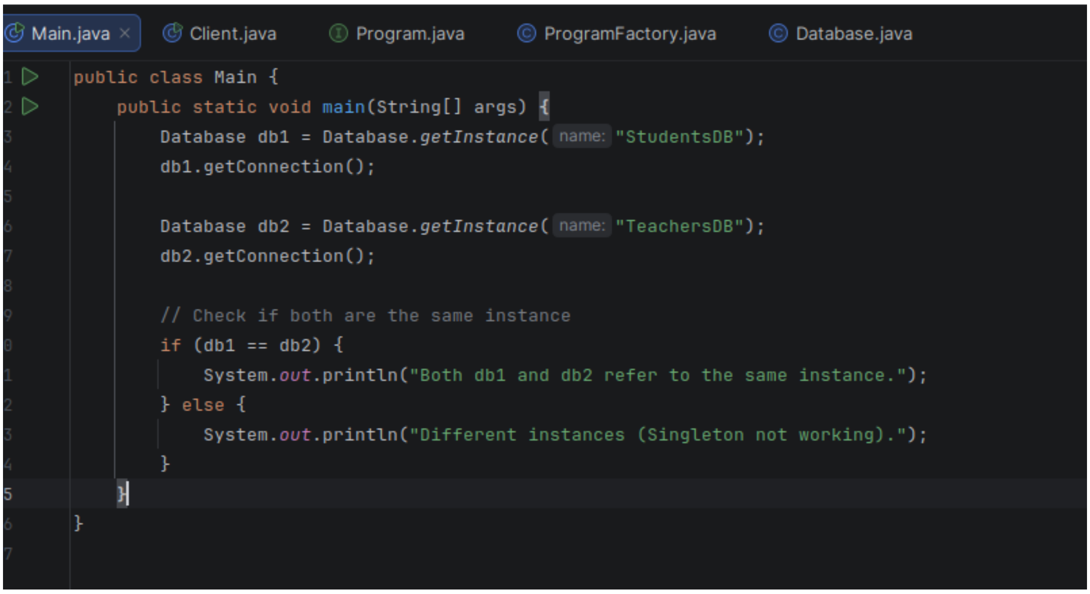


Figure 2: Singleton Database Implementation

Both references point to the same instance, proving that the singleton pattern works correctly.

1.4. Expected Output

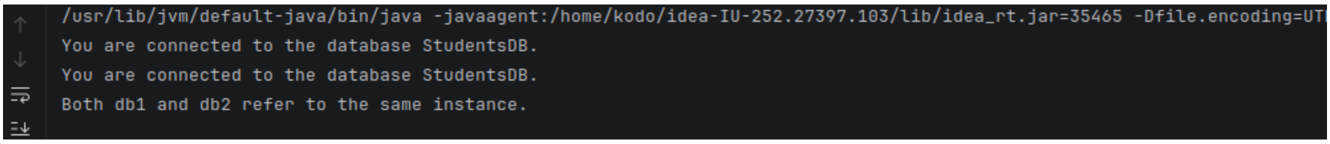


Figure 3: Singleton Database Testing Output

1.5. Conclusion

The Singleton pattern ensures a single instance of a class and provides a global access point. This is particularly useful for managing shared resources such as a database connection.

2. Exercise 2: Factory Pattern

2.1. Part 1: The Problem of Duplicated Code

Initially, we had three classes: `Program1`, `Program2`, and `Program3`. Each class simply displayed a message like:

```
I am Program 1
```

In the `Client` class, three methods `main1()`, `main2()`, and `main3()` each created a different Program object and executed it. This caused significant code duplication and poor maintainability.

2.2. Observation

The main problem with this naive design is code duplication. Each time we add a new program (e.g., `Program4`), we must:

- Create a new class.
- Modify the client code to include another main function.

This violates the **Open/Closed Principle** of software design.

2.3. Part 2: Using an Interface and Factory Class

To solve this, we introduced:

- A `Program` interface containing the method `go()`.
- Classes `Program1`, `Program2`, and `Program3` that implement this interface.
- A `ProgramFactory` class responsible for creating the correct Program object.

2.4. Implementation

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

class Program2 implements Program {
    public void go() {
        System.out.println("Je suis le traitement 2");
    }
}
class Program3 implements Program {
    public void go() {
        System.out.println("Je suis le traitement 3");
    }
}

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

Figure 4: Program Interface Implementation

```java
public class ProgramFactory {
    public static Program createProgram(int number) {
        switch (number) {
            case 1:
                return new Program1();
            case 2:
                return new Program2();
            case 3:
                return new Program3();
            default:
                System.out.println("Invalid program number!");
                return null;
        }
    }
}
```

Figure 5: Program Factory Implementation

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

        int choice = Integer.parseInt(args[0]);

        if (choice == 1) {
            Program1 p = new Program1();
            p.go();
        } else if (choice == 2) {
            Program2 p = new Program2();
            p.go();
        } else if (choice == 3) {
            Program3 p = new Program3();
            p.go();
        } else {
            System.out.println("Invalid program number.");
        }
    }
}
//We notice that there is no flexibility => violating Open-Closed Principle
//In fact the system isn't open for extension and closed for modification.
//Every time we add a new program, we must change existing code.
```

Figure 6: "Before" Implementation

```java
public class Client {
    public static void main(String[] args) {
        if (args.length == 0) {
            System.out.println("Please provide a program number (1, 2, or 3).");
            return;
        }

        int number = Integer.parseInt(args[0]);
        Program p = ProgramFactory.createProgram(number);

        if (p != null) {
            System.out.println("I am main" + number);
            p.go();
        }
    }
}
```

Figure 7: ”After” Implementation

2.5. Output Test

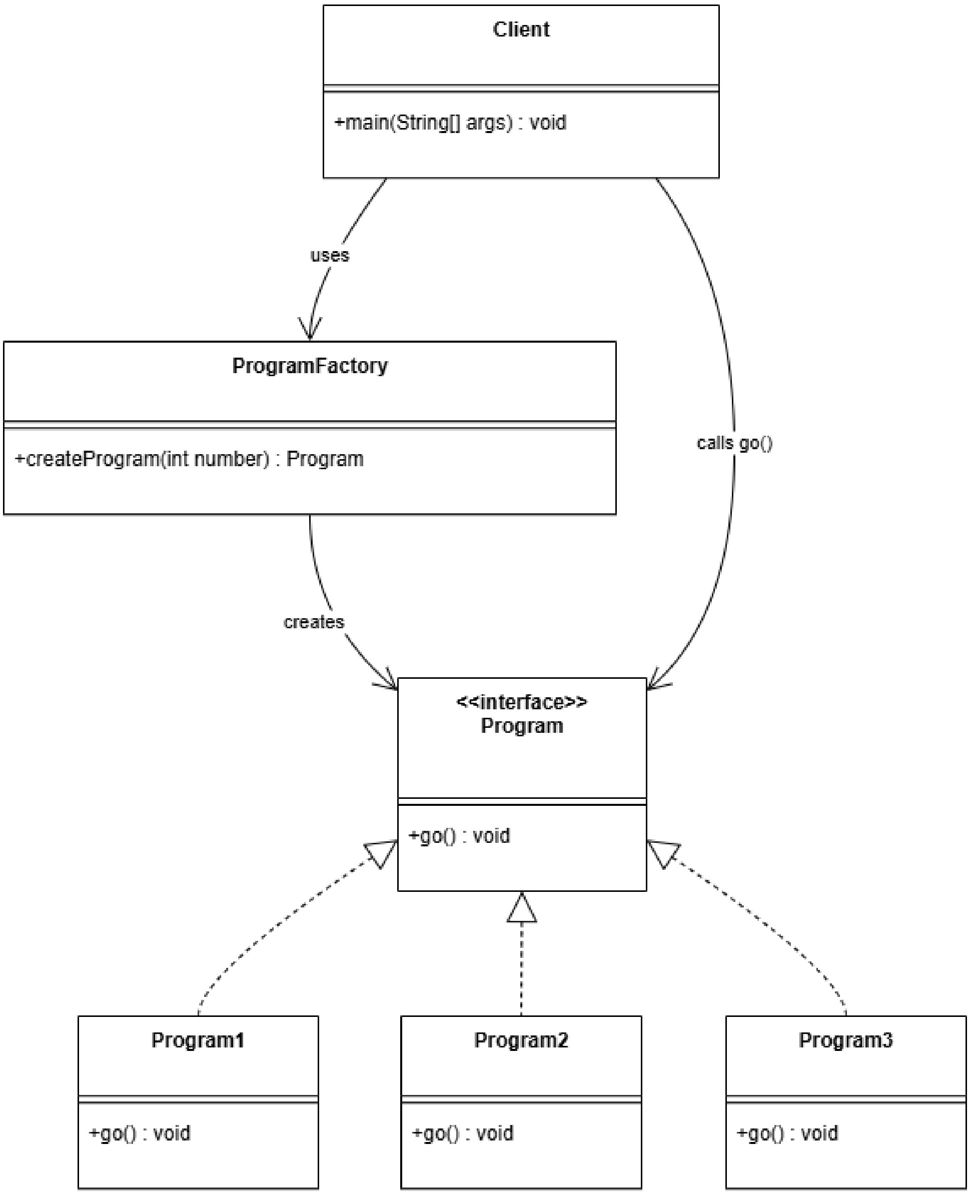
```
┌─[x]─[kodo@parrot]─[~/IdeaProjects/LabSE/src]
└──╼ $javac *.java

┌─[kodo@parrot]─[~/IdeaProjects/LabSE/src]
└──╼ $
┌─[kodo@parrot]─[~/IdeaProjects/LabSE/src]
└──╼ $java Client 1
Je suis le traitement 1
┌─[kodo@parrot]─[~/IdeaProjects/LabSE/src]
└──╼ $java Client 4
Invalid program number.

┌─[kodo@parrot]─[~/IdeaProjects/LabSE/src]
└──╼ $java Client 2
Je suis le traitement 2
```

Figure 8: Expected Output

2.6. Class Diagram



2.7. Results and Discussion

With the factory design:

- Code duplication was eliminated.
- The creation logic was centralized in `ProgramFactory`.
- Adding `Program4` became easy ,we now just create the class and update the factory, without touching the client.

2.8. Conclusion

The Factory pattern provides a clean and scalable design for object creation. It separates object instantiation from usage, promoting flexibility and maintainability.