Lab Report: Design Patterns
Factory and Singleton

Mohamed Amine El Kalai

October 29, 2025

1 Exercise 1: Singleton Pattern - Database Connection

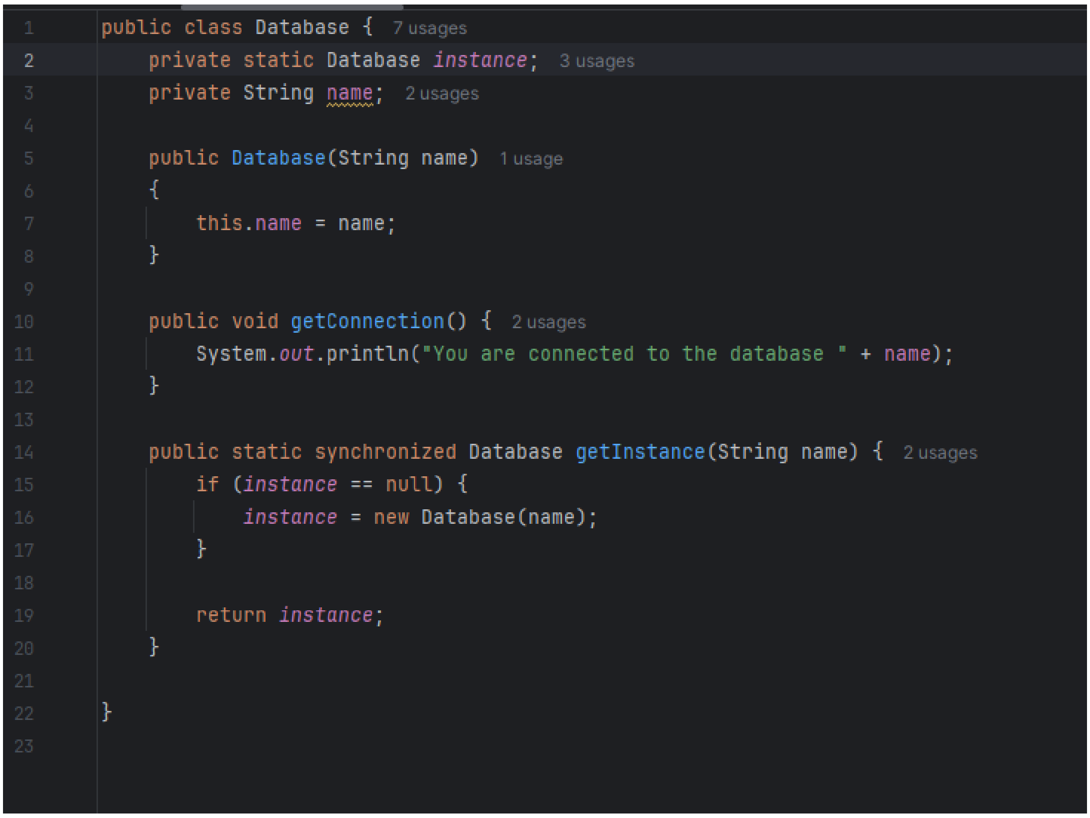
1.1 Objective

The goal was to create a database class that can only have one instance in the entire program. This is useful because we want to make sure there is only one connection to the database at any time.

1.2 Implementation

I created a `Database` class with the following features:

- A private static instance variable
- A private constructor to prevent creating objects directly
- A public static `getInstance()` method that returns the single instance
- A `getConnection()` method that prints a connection message

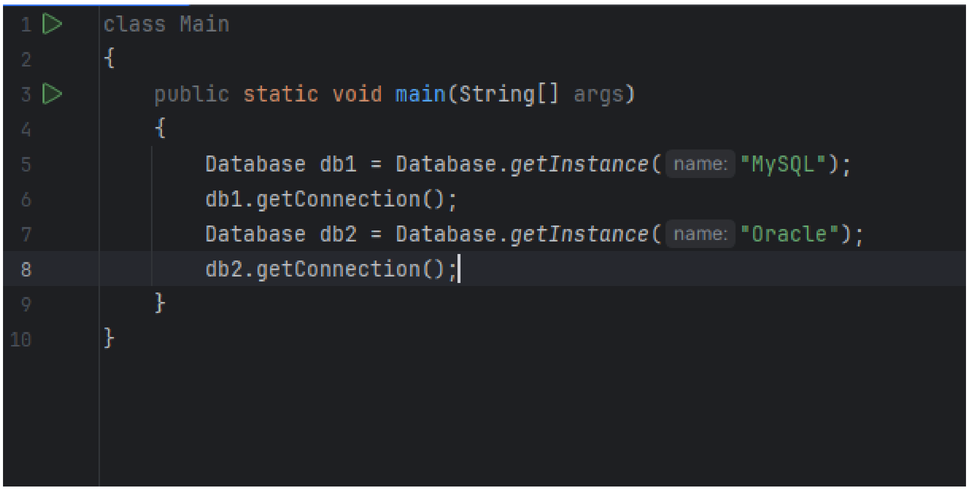


```java
public class Database {  7 usages
    private static Database instance;  3 usages
    private String name;  2 usages

    public Database(String name)  1 usage
    {
        this.name = name;
    }

    public void getConnection() {  2 usages
        System.out.println("You are connected to the database " + name);
    }

    public static synchronized Database getInstance(String name) {  2 usages
        if (instance == null) {
            instance = new Database(name);
        }

        return instance;
    }

}
```

Figure 1: Database class implementation using Singleton pattern

1.3 Testing

In the main method, I tried to create two database objects with different names (”MySQL” and ”Oracle”). The test checks if both variables point to the same instance.

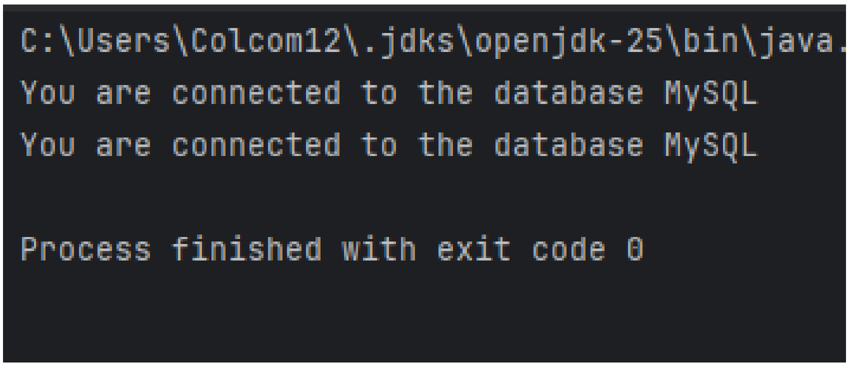


```
class Main
{
    public static void main(String[] args)
    {
        Database db1 = Database.getInstance( name: "MySQL");
        db1.getConnection();
        Database db2 = Database.getInstance( name: "Oracle");
        db2.getConnection();
    }
}
```

Figure 2: Main method testing the Singleton pattern

1.4 Results

The output showed "You are connected to the database MySQL" twice, which means the second call to `getInstance()` with "Oracle" did not create a new instance. This confirms that only one database object exists in the program.



```
C:\Users\Colcom12\.jdks\openjdk-25\bin\java.
You are connected to the database MySQL
You are connected to the database MySQL

Process finished with exit code 0
```

Figure 3: Result

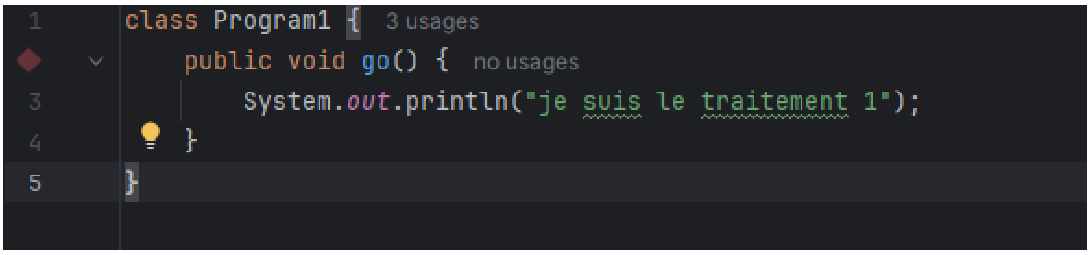
2 Exercise 2: Factory Pattern

2.1 Part 1: Understanding the Problem

The exercise started with a simple project where a Client class creates Program objects. I needed to add Program2 and Program3 classes, and make the Client choose which program to run based on user input.

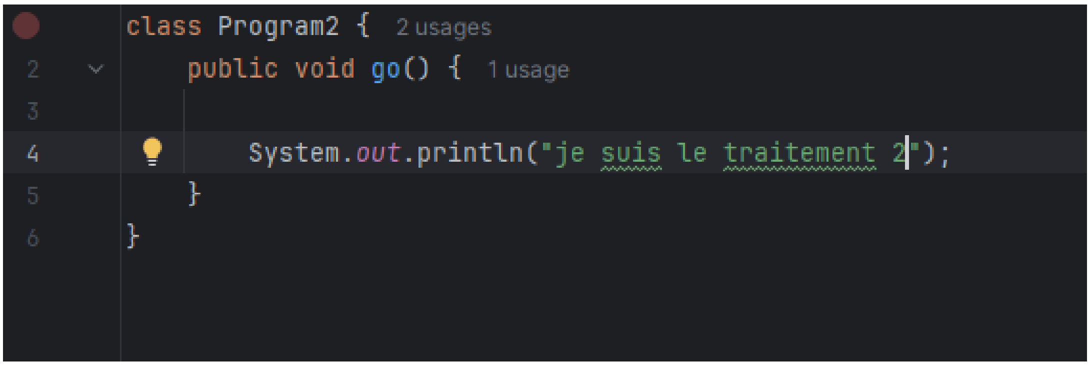
2.2 Creating Program Classes

First we had to create 3 classes (Program1, Program2, Program3) and then choose which one to use based on user's input in the Client Class



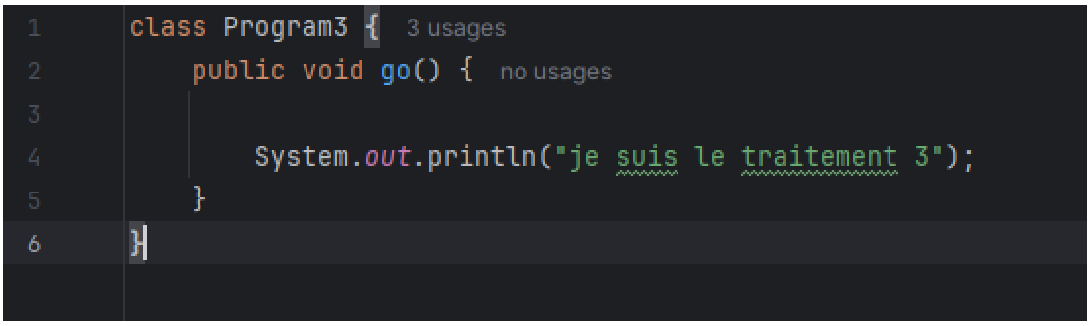
```java
class Program1 {
    public void go() {
        System.out.println("je suis le traitement 1");
    }
}
```

Figure 4: Program1.java



```java
class Program2 {
    public void go() {

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

Figure 5: Program2.java



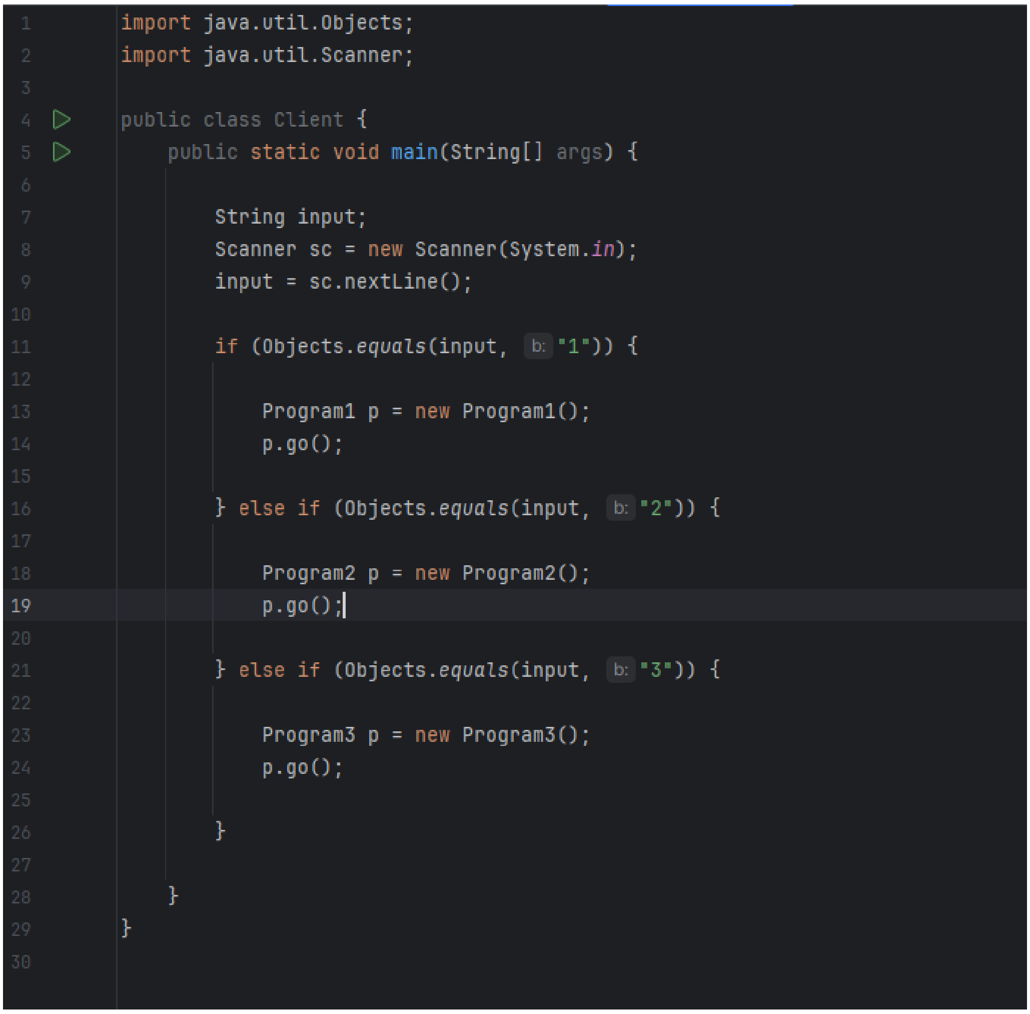
```java
class Program3 {
    public void go() {

        System.out.println("je suis le traitement 3");
    }
}
```

Figure 6: Program3.java

2.3 Naive Solution

My first solution used if-else statements in the Client class to decide which program to create. This approach had several problems:

- Code duplication in multiple places
- Hard to add new programs without changing the Client code
- The Client class needs to know about all program classes

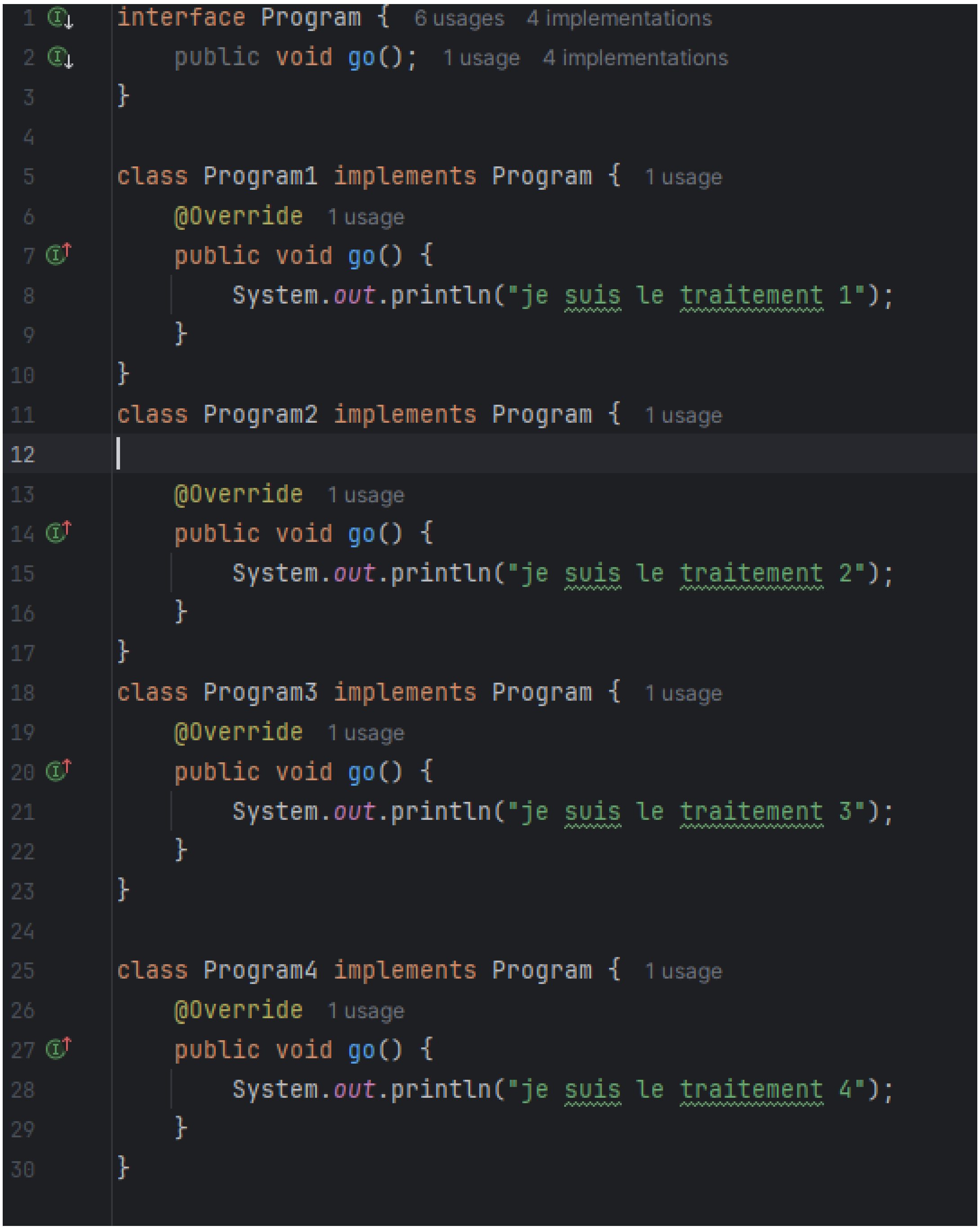


```java
import java.util.Objects;
import java.util.Scanner;

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

        String input;
        Scanner sc = new Scanner(System.in);
        input = sc.nextLine();

        if (Objects.equals(input, b: "1")) {

            Program1 p = new Program1();
            p.go();

        } else if (Objects.equals(input, b: "2")) {

            Program2 p = new Program2();
            p.go();

        } else if (Objects.equals(input, b: "3")) {

            Program3 p = new Program3();
            p.go();

        }

    }
}
```

Figure 7: Naive solution using if-else statements in Client class

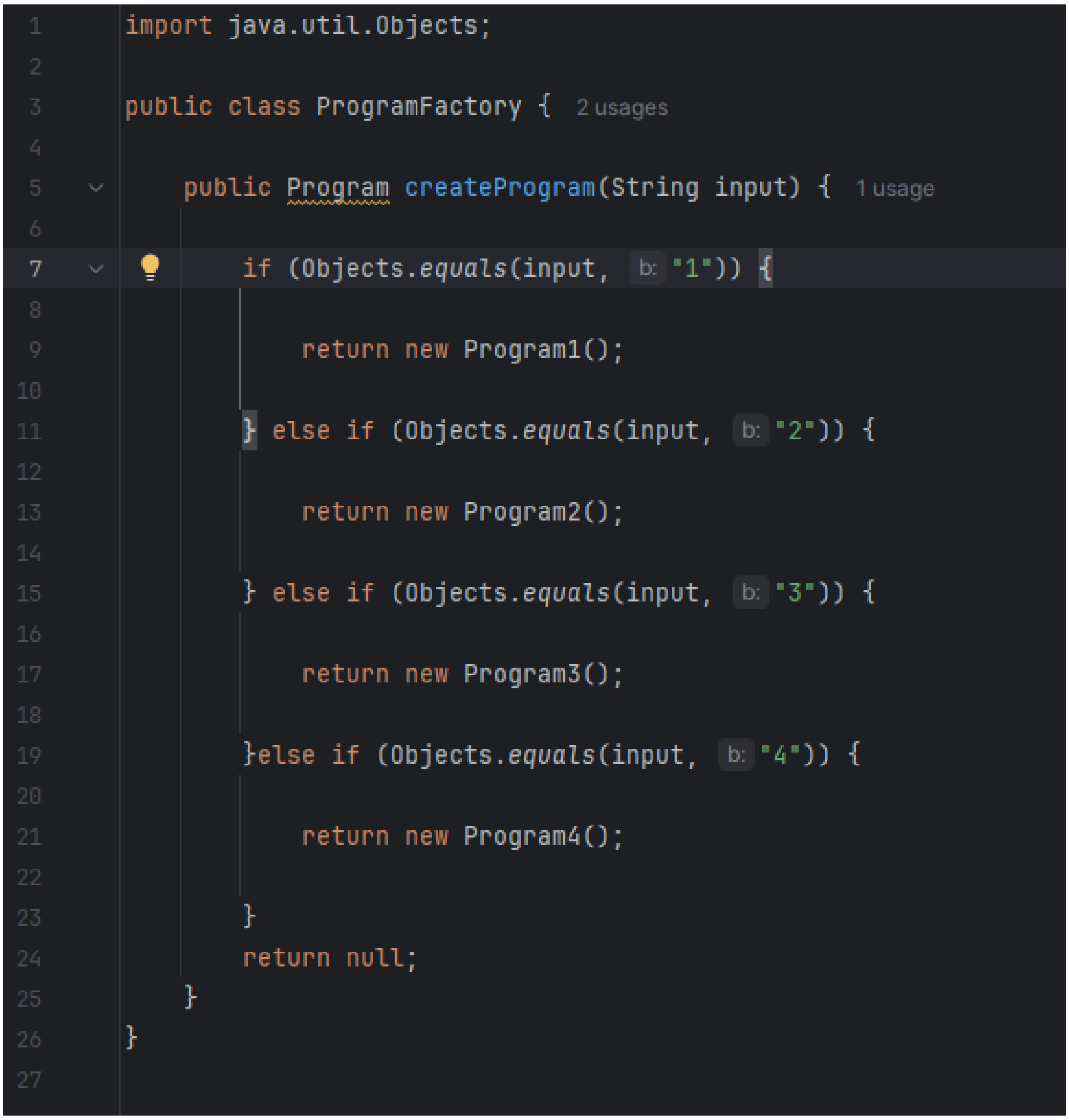
2.4 Factory Pattern Solution

To solve these problems, I created an `Interface` that has the 3 programs (+ an additional 4th one). Then I created a `ProgramFactory` class. This class has one method called `createProgram()` that takes a string input and returns the correct Program object.



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

class Program1 implements Program {
    @Override
    public void go() {
        System.out.println("je suis le traitement 1");
    }
}
class Program2 implements Program {

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

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

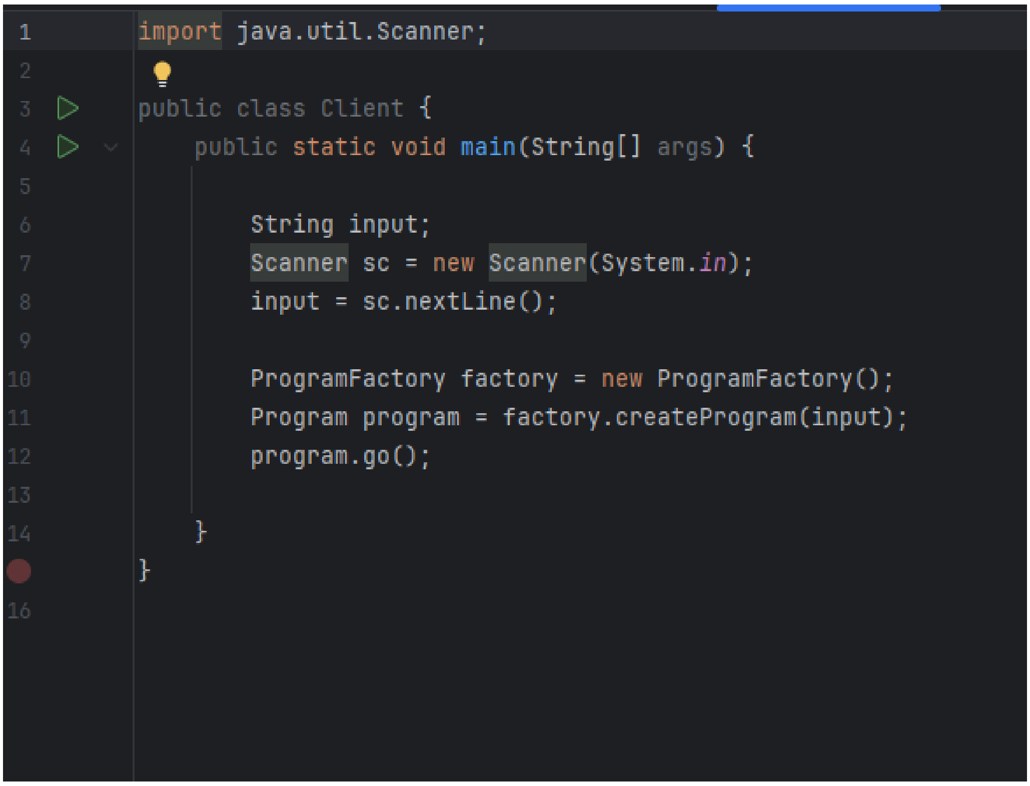
Figure 8: Program Interface



```java
import java.util.Objects;

public class ProgramFactory {

    public Program createProgram(String input) {

        if (Objects.equals(input, "1")) {

            return new Program1();

        } else if (Objects.equals(input, "2")) {

            return new Program2();

        } else if (Objects.equals(input, "3")) {

            return new Program3();

        }else if (Objects.equals(input, "4")) {

            return new Program4();

        }
        return null;
    }
}

```

Figure 9: ProgramFactory class that creates Program objects

Now the Client class is much simpler. It only needs to ask the factory to create a program and then call the `go()` method.



```java
import java.util.Scanner;

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

        String input;
        Scanner sc = new Scanner(System.in);
        input = sc.nextLine();

        ProgramFactory factory = new ProgramFactory();
        Program program = factory.createProgram(input);
        program.go();

    }
}
```

Figure 10: Simplified Client class using ProgramFactory

2.5 Adding New Programs

Adding a new Program4 class was easy with the Factory pattern. I only needed to:

1. Create the new Program4 class that implements Program interface
2. Add one condition in the ProgramFactory class

I did not need to change the Client class at all. This shows that the Factory pattern makes the code easier to maintain and respects the Open-Closed Principle.

2.6 Class Diagram

The final design follows the Factory pattern structure with a clear separation between the client, factory, and product classes.

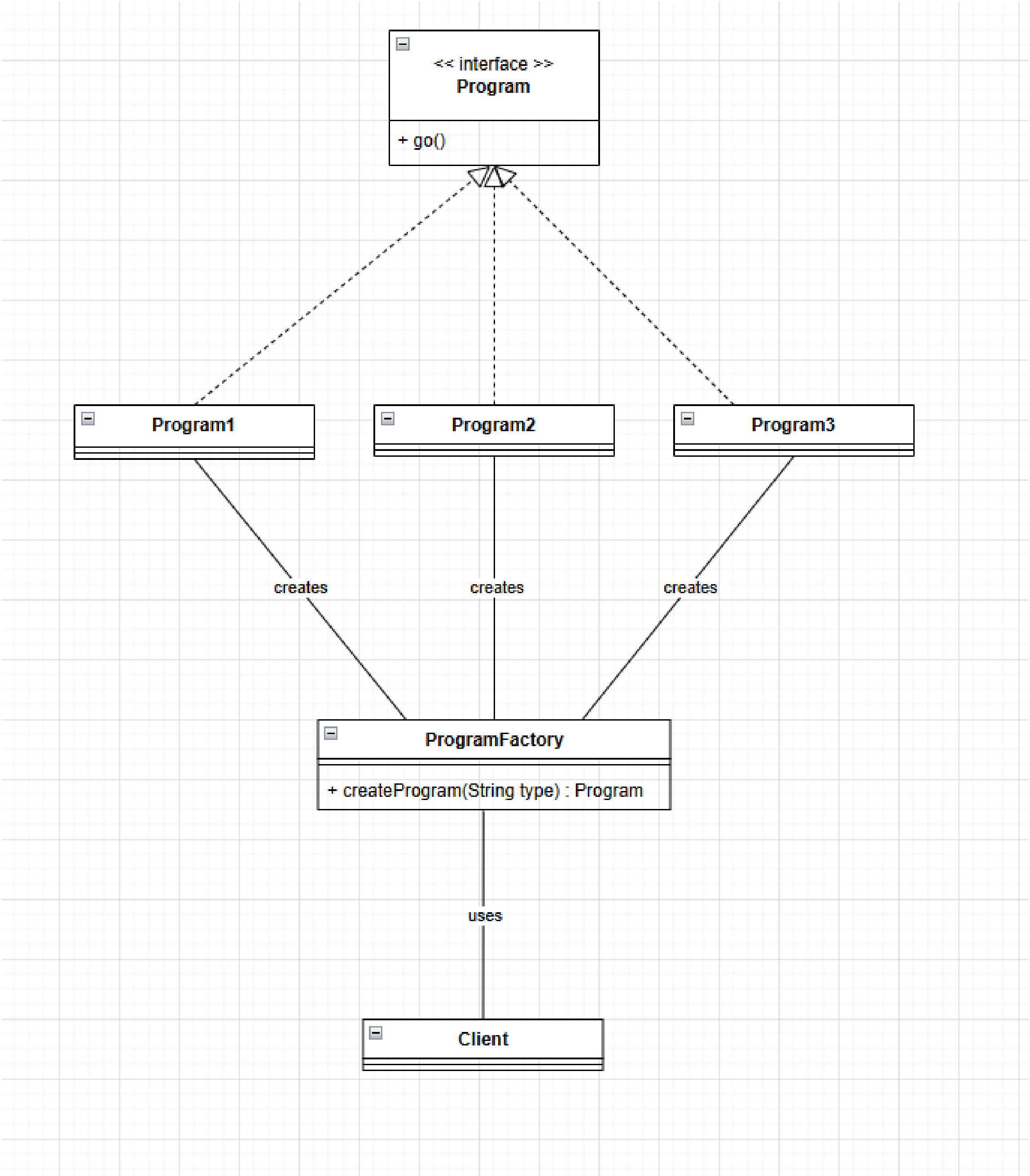


Figure 11: Class diagram showing the Factory pattern structure