Design Patterns: Factory & Singleton
TP Report

Ahmed Khalil EL ATRI

October 29, 2025

1 Exercise 1: Singleton Pattern

1.1 Objective

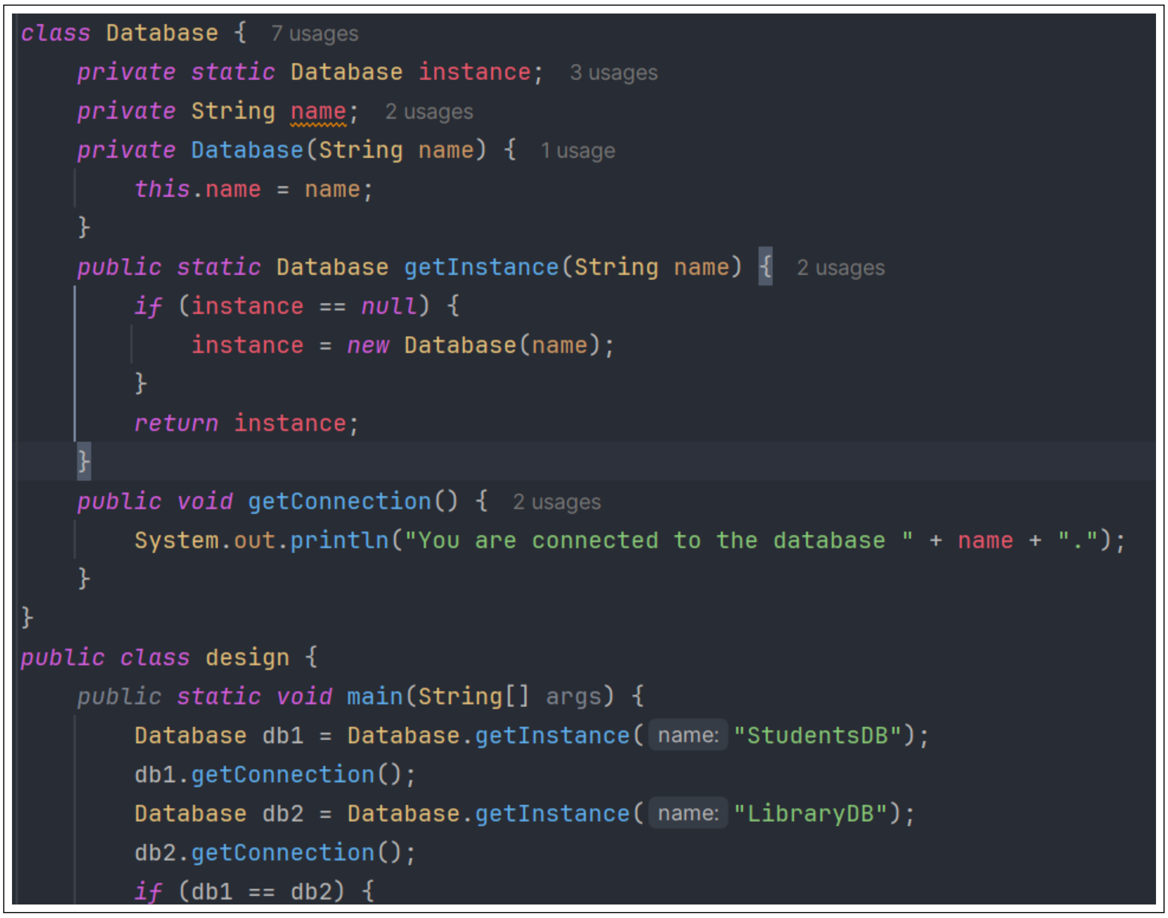
Implement a database class using the Singleton design pattern to ensure only one instance exists throughout the application.

1.2 Implementation

The `Database` class implements the Singleton pattern with:

- Private constructor to prevent external instantiation
- Static `getInstance()` method to control instance creation
- Single static instance variable

1.3 Code



```
class Database {  7 usages
    private static Database instance;  3 usages
    private String name;  2 usages
    private Database(String name) {  1 usage
        this.name = name;
    }
    public static Database getInstance(String name) {  2 usages
        if (instance == null) {
            instance = new Database(name);
        }
        return instance;
    }
    public void getConnection() {  2 usages
        System.out.println("You are connected to the database " + name + ".");
    }
}
public class design {
    public static void main(String[] args) {
        Database db1 = Database.getInstance( name: "StudentsDB");
        db1.getConnection();
        Database db2 = Database.getInstance( name: "LibraryDB");
        db2.getConnection();
        if (db1 == db2) {
```

Figure 1: Database Singleton Implementation (design.java)

1.4 Testing Results

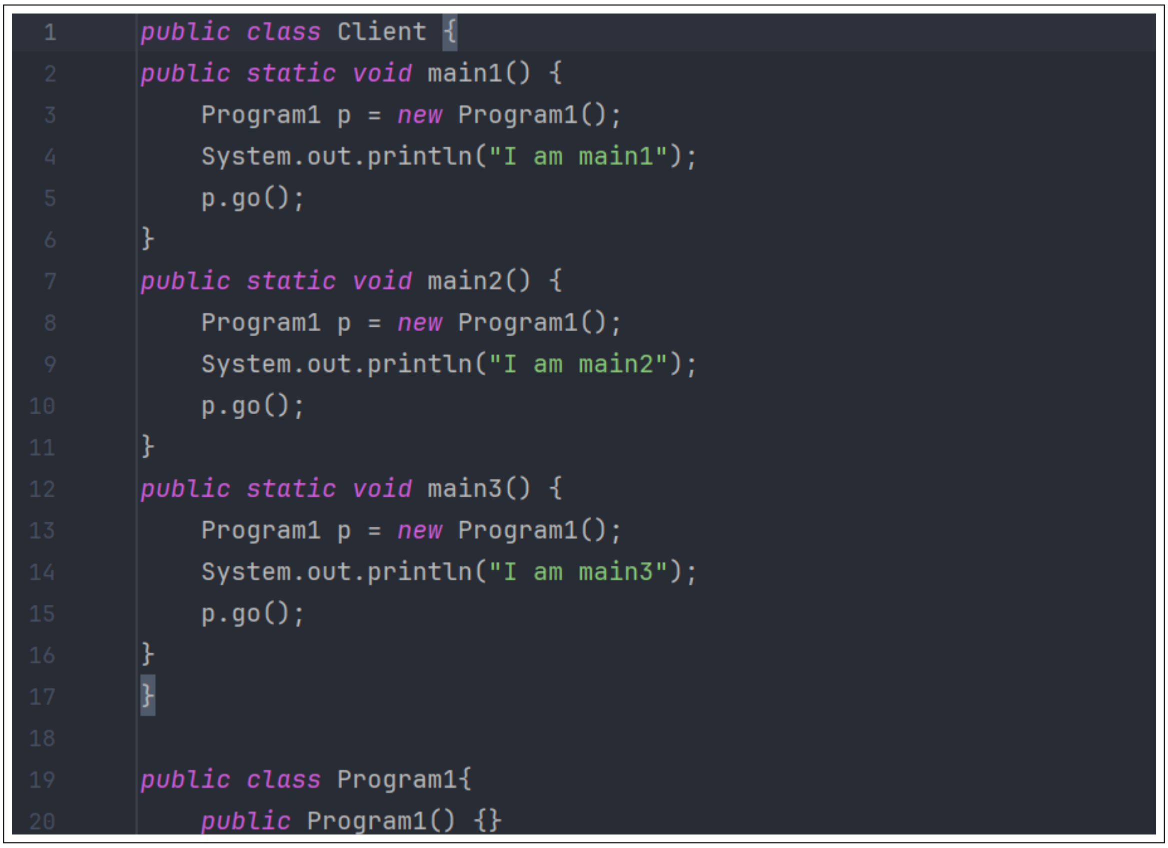
When attempting to create two databases with different names (StudentsDB and LibraryDB), both references point to the same instance. The database retains the first name provided (StudentsDB), confirming that only one instance exists.

2 Exercise 2: Factory Pattern

2.1 Part 1: Naive Solution

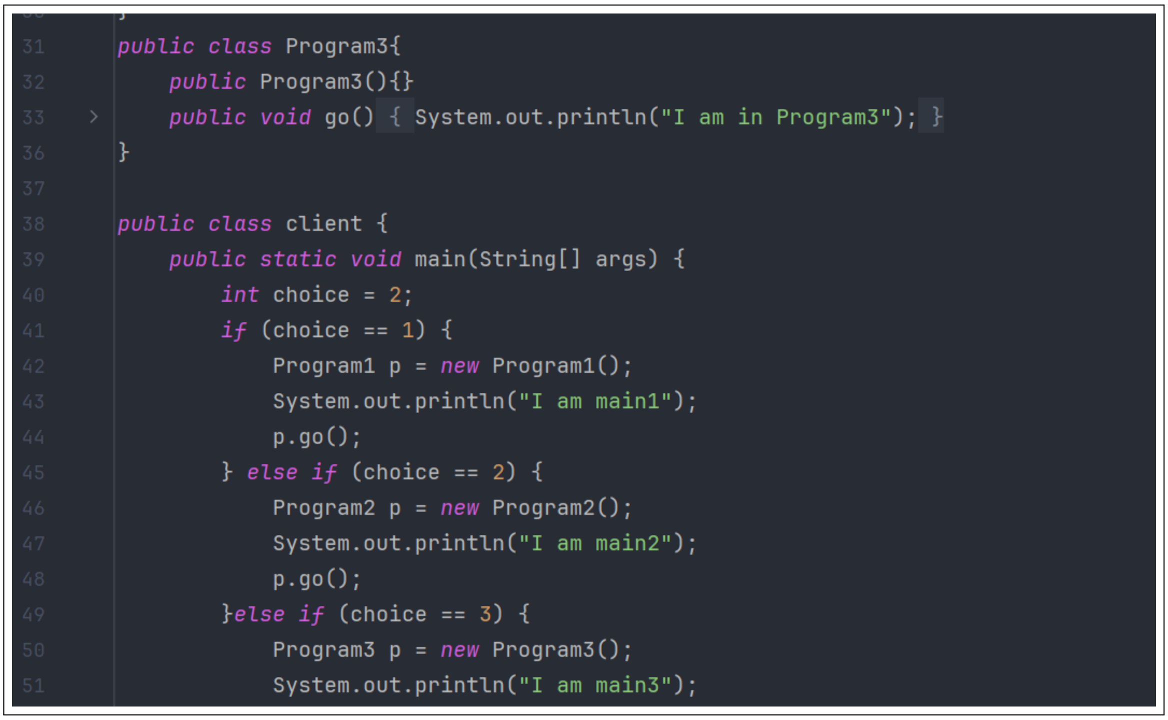
2.1.1 Implementation

The naive solution shows the original `Client` class with duplicated code in multiple methods, and a modified version using `if-else` statements:



```java
public class Client {
public static void main1() {
    Program1 p = new Program1();
    System.out.println("I am main1");
    p.go();
}
public static void main2() {
    Program1 p = new Program1();
    System.out.println("I am main2");
    p.go();
}
public static void main3() {
    Program1 p = new Program1();
    System.out.println("I am main3");
    p.go();
}
}

public class Program1{
    public Program1() {}
```



```java
}
public class Program3{
    public Program3(){}
    public void go() { System.out.println("I am in Program3"); }
}

public class client {
    public static void main(String[] args) {
        int choice = 2;
        if (choice == 1) {
            Program1 p = new Program1();
            System.out.println("I am main1");
            p.go();
        } else if (choice == 2) {
            Program2 p = new Program2();
            System.out.println("I am main2");
            p.go();
        }else if (choice == 3) {
            Program3 p = new Program3();
            System.out.println("I am main3");
```

Figure 2: Naive Solution with Code Duplication (client.java)

2.1.2 What do you notice?

The naive solution requires duplicating object creation code across multiple methods. Each method contains conditional logic (`if-else` statements) to determine which program to instantiate based on the input parameter.

2.2 Part 2: Factory Pattern Solution

To eliminate code duplication, we delegate object creation to a `ProgramFactory` class. This centralizes the creation logic and decouples the client from concrete implementations.

2.3 Class Diagram

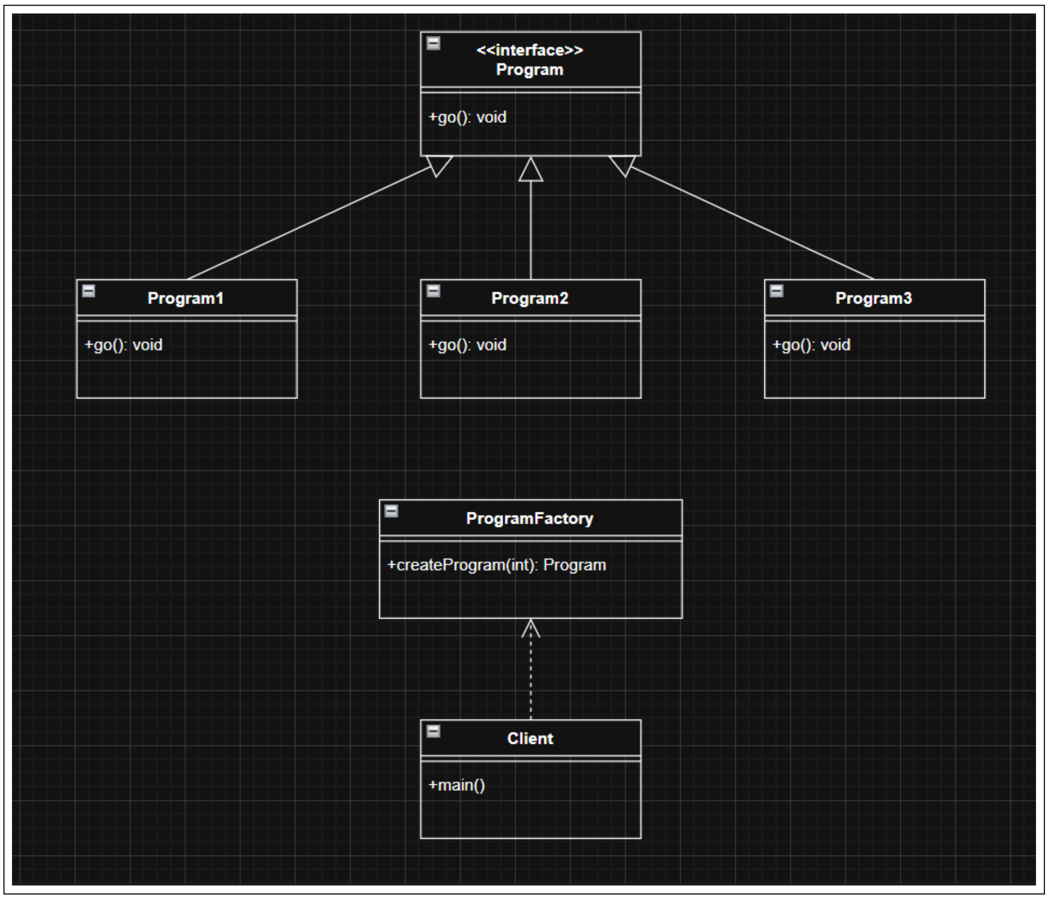
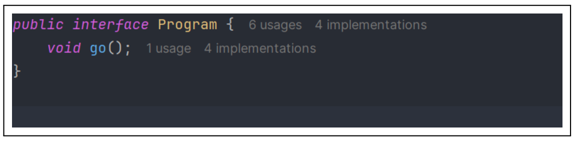


Figure 3: Factory Pattern Class Diagram

2.4 Implementation

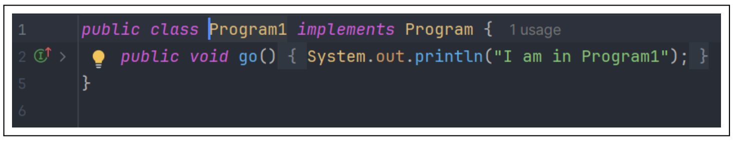
2.4.1 Program Interface



```java
public interface Program {  6 usages  4 implementations
    void go();  1 usage  4 implementations
}
```

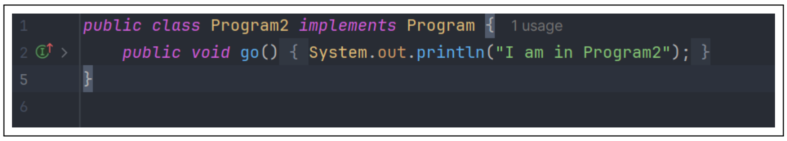
Figure 4: Program Interface

2.4.2 Concrete Program Classes



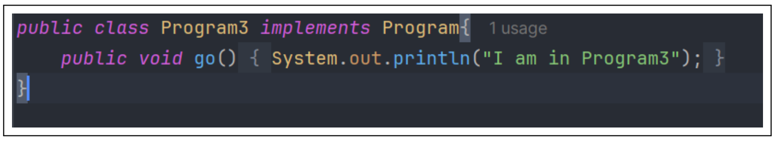
```java
public class Program1 implements Program {
    public void go() { System.out.println("I am in Program1"); }
}
```

Figure 5: Program1 Implementation



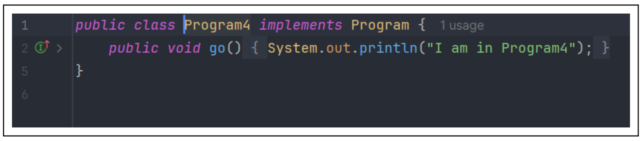
```java
public class Program2 implements Program {
    public void go() { System.out.println("I am in Program2"); }
}
```

Figure 6: Program2 Implementation



```java
public class Program3 implements Program{
    public void go() { System.out.println("I am in Program3"); }
}
```

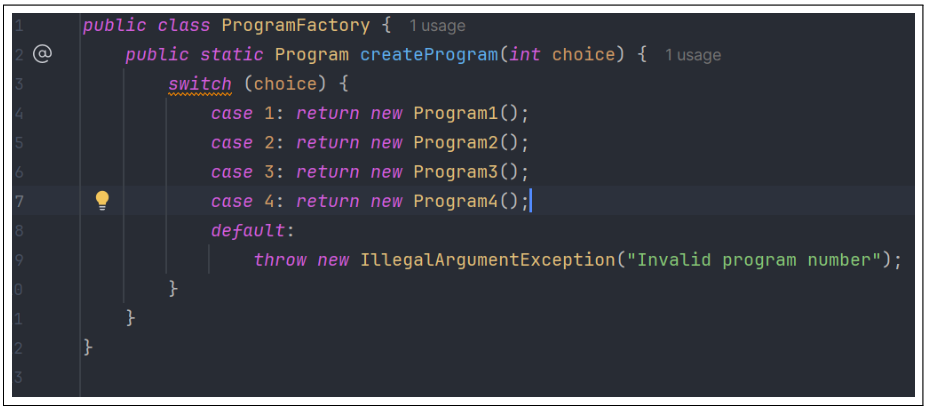
Figure 7: Program3 Implementation



```java
public class Program4 implements Program {
    public void go() { System.out.println("I am in Program4"); }
}
```

Figure 8: Program4 Implementation

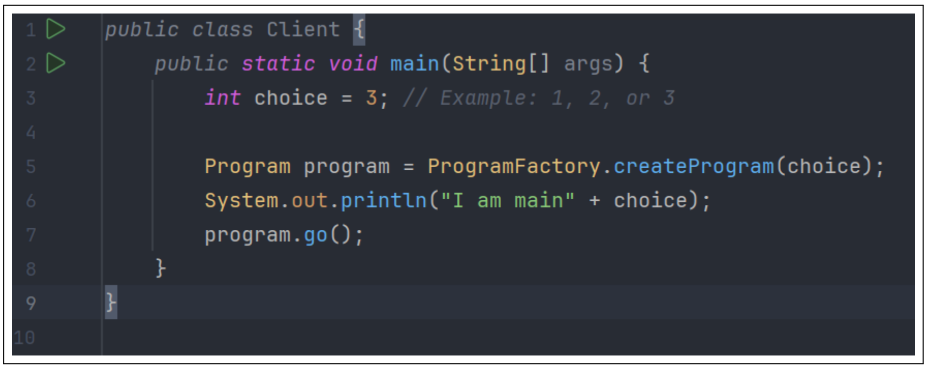
2.4.3 Factory Class



```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();
            case 4: return new Program4();
            default:
                throw new IllegalArgumentException("Invalid program number");
        }
    }
}
```

Figure 9: ProgramFactory Class

2.4.4 Client Class (Refactored)



```java
public class Client {
    public static void main(String[] args) {
        int choice = 3; // Example: 1, 2, or 3

        Program program = ProgramFactory.createProgram(choice);
        System.out.println("I am main" + choice);
        program.go();
    }
}
```

Figure 10: Refactored Client Class using Factory (Client.java)

2.5 Adding Program4

2.5.1 Was it complicated to implement?

No, adding `Program4` was straightforward. It only required:

1. Creating a new `Program4` class implementing the `Program` interface
2. Adding one case in the `ProgramFactory` switch statement

2.5.2 Did you have to modify the Client code?

No modifications were needed to the `Client` class structure. The client code remains unchanged and automatically supports the new program through the factory. Only the choice parameter needs to be set to 4 to use `Program4`.

3 Conclusion

The Singleton pattern ensures controlled access to a single shared instance, while the Factory pattern encapsulates object creation logic. Together, these patterns improve code maintainability, reduce duplication, and promote loose coupling in object-oriented systems.