MINISTRY OF EDUCATION AND RESEARCH
OF THE REPUBLIC OF MOLDOVA

Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics
Department of Software Engineering and Automation

Petcov Nicolai FAF-233

Report

Laboratory Work No. 0

Software Design Techniques and Mechanisms

SOLID Principles Implementation

Checked by:
Guzun C., *university assistant*
DISA, FCIM, UTM

Chișinău – 2025

1 Objective

Implement a Car Factory system that demonstrates the practical application of SOLID principles, specifically focusing on:

- Single Responsibility Principle (S)
- Open-Closed Principle (O)
- Dependency Inversion Principle (D)

2 Tasks

1. Create a basic factory pattern implementation for car manufacturing
2. Apply the Single Responsibility Principle to separate concerns
3. Implement the Open-Closed Principle for extensibility
4. Utilize the Dependency Inversion Principle for loose coupling
5. Document the implementation and demonstrate SOLID principles usage

3 Implementation

3.1 Project Structure

The project follows a structured organization:

```
src/main/java/factory/
          car/
                Car.java
                CarType.java
                ElectricCar.java
                GasCar.java
          engine/
                Engine.java
                ElectricEngine.java
                GasEngine.java
          production/
              CarFactory.java
              FactoryApp.java
```

3.2 SOLID Principles Implementation

3.2.1 Single Responsibility Principle (S)

Each class in the project has a single, well-defined responsibility:

```java
// Car.java - Represents a car with basic properties
public abstract class Car {
    private Engine engine;
    private String model;

```

```java
    public Car(Engine engine, String model) {
        this.engine = engine;
        this.model = model;
    }

    public abstract void assemble();
}

// Engine.java - Handles engine-specific functionality
public interface Engine {
    void start();
    void stop();
    String getType();
}

// CarFactory.java - Manages car creation
public class CarFactory {
    public Car createCar(CarType type, String model) {
        switch (type) {
            case ELECTRIC:
                return new ElectricCar(new ElectricEngine(), model);
            case GAS:
                return new GasCar(new GasEngine(), model);
            default:
                throw new IllegalArgumentException("Unknown car type");
        }
    }
}
```

3.2.2 Open-Closed Principle (O)

The system is designed to be open for extension but closed for modification:

```java
// New car types can be added by extending Car
public class ElectricCar extends Car {
    public ElectricCar(Engine engine, String model) {
        super(engine, model);
    }

    @Override
    public void assemble() {
        // Electric car specific assembly
    }
}

// New engine types can be added by implementing Engine
public class ElectricEngine implements Engine {
    @Override
    public void start() {
        // Electric engine start logic
    }

    @Override
    public void stop() {
        // Electric engine stop logic
    }

    @Override
```

```
    public String getType() {
        return "Electric";
    }
}
```

3.2.3 Dependency Inversion Principle (D)

High-level modules depend on abstractions:

```
// Car depends on Engine interface, not concrete implementations
public abstract class Car {
    private Engine engine;  // Dependency on abstraction

    public Car(Engine engine, String model) {
        this.engine = engine;
    }
}

// Factory works with abstract Car class
public class CarFactory {
    public Car createCar(CarType type, String model) {
        // Creates different car types based on abstraction
    }
}
```

4 Results and Testing

4.1 Main Application

The main application demonstrates the usage of the factory pattern with SOLID principles:

```
public class Main {
    public static void main(String[] args) {
        CarFactory factory = new CarFactory();

        // Create different types of cars
        Car electricCar = factory.createCar(CarType.ELECTRIC, "Tesla");
        Car gasCar = factory.createCar(CarType.GAS, "Toyota");

        // Demonstrate functionality
        electricCar.assemble();
        gasCar.assemble();
    }
}
```

4.2 Benefits of Implementation

1. **Single Responsibility Principle**
    - Each class has a clear, single purpose
    - Easier maintenance and testing
    - Reduced code complexity

2. **Open-Closed Principle**
    - New car types can be added without modifying existing code
    - Extensible design through inheritance and interfaces
    - Reduced risk of breaking existing functionality
3. **Dependency Inversion Principle**
    - Loose coupling between components
    - Easy to swap implementations
    - Improved testability through abstractions

5 Conclusion

This laboratory work successfully demonstrated the implementation of three SOLID principles in a practical Car Factory system:

- The Single Responsibility Principle was applied by creating classes with focused responsibilities
- The Open-Closed Principle was implemented through extensible class hierarchies
- The Dependency Inversion Principle was utilized to create loose coupling between components

The implementation showcases how SOLID principles can be applied to create maintainable, extensible, and robust software systems. The factory pattern, combined with these principles, provides a flexible foundation for creating different types of cars while maintaining clean and organized code.

Key takeaways:

- SOLID principles improve code organization and maintainability
- Abstract classes and interfaces provide flexibility
- Factory pattern works well with SOLID principles
- Code is now easier to extend and test

Source Code

https://github.com/Nickseen/SDTM-Labs/tree/lab0