JOMO KENYATTA UNIVERSITY OF AGRICULTURE AND TECHNOLOGY.

M.Sc. SOFTWARE ENGINEERING

ICS 3105: - OBJECT ORIENTED SOFTWARE ENGINEERING.

ASSIGNMENT 1

Dr Eunice Njeri.

ALEX KEMBOI

SCT 313-0530/2023

A Research proposal submitted to Jomo Kenyatta University of Agriculture and Technology, School of Computing. Partial fulfillment of the requirement for the award of degree of Masters of Science in Software Engineering.

2025

SOLID Principles

SOLID is an acronym for five design principles that help create maintainable and scalable software.

Single Responsibility Principle (SRP)

● Open/Closed Principle (OCP)

● Liskov Substitution Principle (LSP)

● Interface Segregation Principle (ISP)

● Dependency Inversion Principle (DIP)

SOLID principles improve software **maintainability**, **scalability**, and **flexibility** by reducing tight coupling and enforcing best practices.

1. Single Responsibility Principle (SRP)

A class should have only one reason to change.
Each class should have only one responsibility.

Violation of SRP

```
class Report {
    public String generateReport() {
        return "Report Content";
    }

    public void saveToFile(String content, String filename) {
        System.out.println("Saving report to " + filename);
    }
}
```

Problem: The Report class handles both report generation and file saving.

Refactored Code (SRP Applied)

```java
class Report {
    public String generateReport() {
        return "Report Content";
    }
}

class FileManager {
    public void saveToFile(String content, String filename) {
        System.out.println("Saving report to " + filename);
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Report report = new Report();
        String content = report.generateReport();

        FileManager fileManager = new FileManager();
        fileManager.saveToFile(content, "report.txt");
    }
}
```

2. Open/Closed Principle (OCP)

Software entities should be open for extension but closed for modification.

Violation of OCP

```java
class DiscountCalculator {
    public double calculate(double amount, String discountType) {
        if ("percentage".equals(discountType)) {
            return amount * 0.9; // 10% discount
        } else if ("fixed".equals(discountType)) {
            return amount - 50;
        }
        return amount;
    }
}
```

Problem: Adding a new discount type requires modifying the class.

Refactored Code (OCP Applied)

```
interface Discount {
    double apply(double amount);
}

class PercentageDiscount implements Discount {
    public double apply(double amount) {
        return amount * 0.9;
    }
}

class FixedDiscount implements Discount {
    public double apply(double amount) {
        return amount - 50;
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Discount discount = new PercentageDiscount(); // Can be replaced with FixedDiscount
        System.out.println(discount.apply(500));
    }
}
```

3. Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types without breaking functionality.

Violation of LSP

```
class Bird {
    public void fly() {
        System.out.println("Flying...");
    }
}

class Penguin extends Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Penguins can't fly!");
    }
}
```

Problem: Penguin extends Bird but cannot fly, violating the expected behavior.

Refactored Code (LSP Applied)

```
abstract class Bird {}

abstract class FlyingBird extends Bird {
    abstract void fly();
}

class Sparrow extends FlyingBird {
    @Override
    public void fly() {
        System.out.println("Flying...");
    }
}

class Penguin extends Bird {
    public void swim() {
        System.out.println("Swimming...");
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        FlyingBird sparrow = new Sparrow();
        sparrow.fly();

        Penguin penguin = new Penguin();
        penguin.swim();
    }
```

4. Interface Segregation Principle (ISP)

A class should not be forced to implement interfaces it does not use.

Violation of ISP

```
interface Worker {
    void work();
    void eat();
}

class Robot implements Worker {
    @Override
    public void work() {
        System.out.println("Working...");
    }

    @Override
    public void eat() {
        throw new UnsupportedOperationException("Robots don't eat!");
    }
}
```

Problem: Robot is forced to implement eat(), which it does not need.

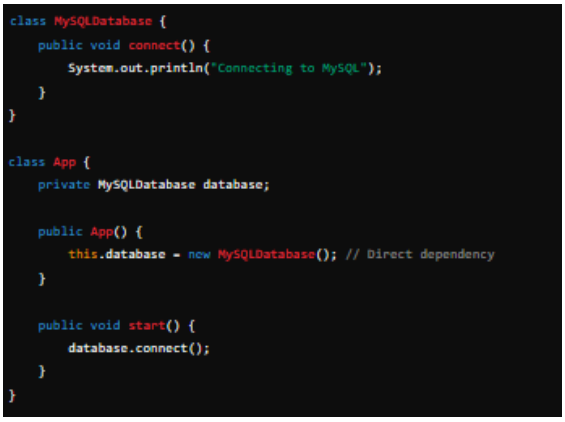
Refactored Code (ISP Applied)

```
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class Human implements Workable, Eatable {
    @Override
    public void work() {
        System.out.println("Working...");
    }

    @Override
    public void eat() {
        System.out.println("Eating...");
    }
}

class Robot implements Workable {
    @Override
    public void work() {
        System.out.println("Working...");
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Human human = new Human();
        human.work();
        human.eat();

        Robot robot = new Robot();
        robot.work();
    }
}
```

5. Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Violation of DIP



```
class MySQLDatabase {
    public void connect() {
        System.out.println("Connecting to MySQL");
    }
}

class App {
    private MySQLDatabase database;

    public App() {
        this.database = new MySQLDatabase(); // Direct dependency
    }

    public void start() {
        database.connect();
    }
}
```

Problem: App is tightly coupled to MySQLDatabase.

Refactored Code (DIP Applied)

```
interface Database {
    void connect();
}

class MySQLDatabase implements Database {
    @Override
    public void connect() {
        System.out.println("Connecting to MySQL");
    }
}

class PostgreSQLDatabase implements Database {
    @Override
    public void connect() {
        System.out.println("Connecting to PostgreSQL");
    }
}

class App {
    private Database database;

    public App(Database database) {
        this.database = database; // Dependency injection
    }

    public void start() {
        database.connect();
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        Database database = new MySQLDatabase(); // Easily switch to PostgreSQLDatabase
        App app = new App(database);
        app.start();
    }
}
```

GIT HUB LINK

https://github.com/alexkemboi/Solid-Principles.git