Shahjalal University Of Science And Technology, Sylhet
Software Engineering, IICT

Software Architecture and Design Patterns

ASSIGNMENT 1

Prepared By:

Md. Rakibul Hasan
2017831005

Supervisor:

Fazle Mohammed Tawsif
Lecturer, IICT, SUST

15/12/2020

1. Explain the SOLID principal

SOLID Principles is a coding standard that all developers should have a clear concept for developing software properly to avoid a bad design. It was promoted by Robert C Martin and is used across the object-oriented design spectrum. When applied properly it makes your code more extendable, logical, and easier to read.

When the developer builds software following a bad design, the code can become inflexible and more brittle. Small changes in the software can result in bugs. For these reasons, we should follow SOLID Principles.

S.O.L.I.D stands for:

- S - Single-responsiblity principle
- O - Open-closed principle
- L - Liskov substitution principle
- I - Interface segregation principle
- D - Dependency Inversion Principle

1.1. Single-responsibility Principle

A class should have one and only one reason to change, meaning that a class should have only one job.

Please look at the following code:

```
public class Vehicle {
    public void printDetails() {}
    public double calculateValue() {}
    public void addVehicleToDB() {}
}
```

The above class violates the single responsibility principle. Why should this class add Vehicle to the database? The Vehicle class has three separate responsibilities: reporting, calculation, and database. By applying SRP, we can separate the above class into three classes with separate responsibilities.

1.2. Open-closed Principle

Objects or entities should be open for extension, but closed for modification.

Consider the below method of the class VehicleCalculations:

```java
public class VehicleValueCalculator {
    // lets assume a simple method to calculate the total value of a vehicle
    // with extra cost depending the type.
    public double calculateVehicle(Vehicle v){
        double value = 0;
        if(v instanceof Car){
            value = v.getValue() + 2.0;
        } else if(v instanceof MotorBike) {
            value = v.getValue() + 0.4;
        }
        return value;
    }
}
```

Suppose we now want to add another subclass called Truck. We would have to modify the above class by adding another if statement, which goes against the Open-Closed Principle. A better approach would be for the subclasses Car and Truck to override the calculateValue method:

Solution

```java
public interface IVehicle {
      double calculateVehicle();
}
```

```java
public class Car implements IVehicle {
    @Override
    public double calculateVehicle() {
        return this.getValue() + 2.0;
    }
}
public class MotorBike implements IVehicle {
    @Override
    public double calculateVehicle() {
        return this.getValue() + 0.4;
    }
}
```

Our new Truck Vehicle

```java
public class Truck implements IVehicle {
    @Override
    public double calculateVehicle() {
        return this.getValue() + 3.4;
    }
}
```

This way by having a method that accepts an IVehicle , there is no need for refactoring/code modification in the future every time we add a new type of vehicle.

Example code

```java
public class Main {
    public static void main(String[] args){
        IVehicle car = new Car();
        IVhecile motorBike = new MotorBike();
        //new addition
        IVhecile truck = new Truck();
        double carValue       = getVehicleValue(car);
        double motorBikeValue = getVehicleValue(motorBike);
        double truckValue     = getVehicleValue(truck);
    }
    public double getVehicleValue(IVehicle v) {
        return v.calculateVehicle();
    }
}
```

1.3. Liskov substitution principle

The Liskov Substitution Principle (LSP) applies to inheritance hierarchies such that derived classes must be completely substitutable for their base classes.

To make this more clear, we are going to check out the simple example below:

```java
/**
 * The Base Rectangle class
 * This class defines the structure and properties of all types of rectangles
 */
public class Rectangle {

    private int width;
    private int height;

    public Rectangle(){}

    public Rectangle(int w,int h) {
        this.width = w;
        this.height = h;
    }

    public int getWidth() {
        return width;
    }

    public void setWidth(int width) {
        this.width = width;
    }

    public int getHeight() {
        return height;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return this.height * this.width;
    }

    /**
     * LSP violation is case of a Square reference.
     */
    public final static void setDimensions(Rectangle r,int w,int h) {
          r.setWidth(w);
          r.setHeight(h);
          //assert r.getArea() == w * h
    }
}
```

```java
/**
 * A Special kind of Rectangle
 */
public class Square extends Rectangle {
    @Override
    public void setHeight(int h){
        super.setHeight(h);
        super.setWidth(h);
    }

    @Override
    public void setWidth(int w) {
        super.setWidth(w);
        super.setHeight(w);
    }
}
```

When talking about LSP, we have the method setDimensions in the Rectangle class that accepts a type of Rectangle object and sets the width and height. This is a violation because the behavior changed and we have inconsistent data when we pass a square reference.

1.4. Interface segregation principle

A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use.

ISP states that we should split our interfaces into smaller and more specific ones.

Below is an example of an interface representing two different roles. One is the role of handling connections like opening and closing, and the other is sending and receiving data.

```java
public interface Connection {
    void open();
    void close();
    byte[] receive();
    void send(byte[] data);
}
```

After we applied ISP, we ended up with two different interfaces, with each one representing one exact role.

```java
public interface Channel {
    byte[] receive();
    void send(byte[] data);
}

public interface Connection {
    void open();
    void close();
}
```

1.5. Dependency Inversion principle

Entities must depend on abstractions not on concretions. It states that the high level module must not depend on the low level module, but they should depend on abstractions.

Next is a violation of DIP. We have an Emailer class depending on a direct SpellChecker class:

```java
public class Emailer{
    private SpellChecker spellChecker;
    public Emailer(SpellChecker sc) {
        this.spellChecker = sc;
    }
    public void checkEmail() {
        this.spellChecker.check();
    }
}
```

And the Spellchecker class:

```java
public class SpellChecker {
    public void check() throws SpellFormatException {
    }
}
```

It may work at the moment, but after a while, we have two different implementations of spellcheckers we want to include. We have the default spell checker and a new greek spell checker.

With the current implementation, refactoring is needed because the Emailer class uses only the SpellChecker class.

A simple solution is to create the interface for the different spell checkers to implement.

```java
// The interface to be implemented by any new spell checker.
public interface ISpellChecker {
    void check() throws SpellFormatException;
}
```

Now, the Emailer class accepts only an ISpellChecker reference on the constructor. Below, we changed the Emailer class to not care/depend on the implementation (concrete class) but rely on the interface (ISpellChecker)

```java
public class Emailer{
    private ISpellChecker spellChecker;
    public Emailer(ISpellChecker sc) {
        this.spellChecker = sc;
    }
    public void checkEmail() {
        this.spellChecker.check();
    }
}
```

And we have many implementations for the ISpellChecker:

```java
public class SpellChecker implements ISpellChecker {
    @Override
    public void check() throws SpellFormatException {
    }
}

public class GreekSpellChecker implements ISpellChecker {
    @Override
    public void check() throws SpellFormatException {
    }
}
```

2. What is Refused Bequest? Give an example of the problem and solution for the problem. Depict in a UML Diagram.

Refused Bequest is kind of a code smell. It is based on the LISKOV Substitution Principle because it violates this principle. The contract of the base class is not honored by the derived class, and that forms this code smell.

There is a child class and it inherits from a base class, but the child class doesn't need all behavior provided by the base class. Other behaviors which are not required to child class are refused. That refused behavior is class refused bequest.

Example:
Code with Refused Bequest.

```java
public class Vehicle
    {
        protected void Drive() { }
    }

    public class Car : Vehicle
    {
    }

    public class Plane : Vehicle
    {
    }
```

Refused bequest code smell can be solved by using two methods.

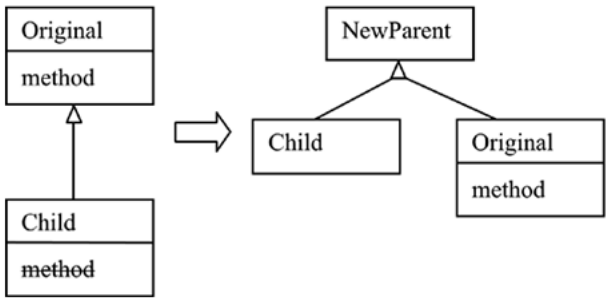
1. Push down method and push down field
2. Replace Inheritance with Delegation

Push Down Method and Push Down Field It is a common way to solve the issue of Refused Bequest. Remove the method or property from Base class and move it to that subclass where it fits.

Code solving the problem of Refused Bequest.

```
01. public class Vehicle
02. {
03. }
04.
05. public class Car : Vehicle
06. {
07.    void Drive() { }
08. }
09.
10. public class Plane : Vehicle
11. {
12. }
```

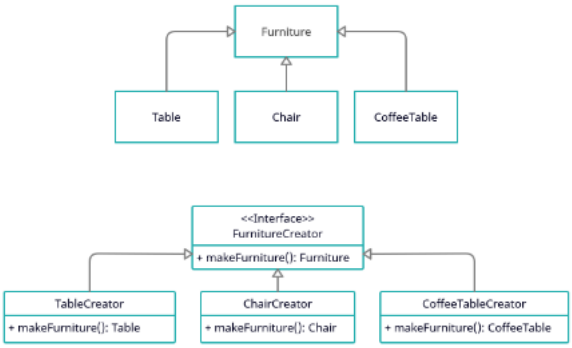
Depict in a UML Diagram



3. Suppose you run a Furniture company which sells chairs and tables. Later on, a coffee table item is introduced. Propose a design for this scenario where more items can be accommodated easily into the system. Provide a UML diagram and pseudo code for the proposed solution.

As we are operating a furniture company. We have to create different types of furniture. So I think we can solve this problem my Factory design pattern.

We can create a factory interface for the furnitures named **FurnitureCreator**. We can create **TableCreator**, **ChairCreator** and **CoffeeTableCreator** by implementing the **FurnitureCreator** interface. We can create a Furniture class. And then we can createChair, Table and CoffeeTable by extending the Furniture class.



Now if we want to add one more furniture, we can just extend theFurnitureclass. To createthe factory we can implement theFurnitureCreatorinterface.Pseudocode for this pattern:

```java
public class Furniture{
    // Do something
}

class Table extends Furniture{
    // Do something
}

class Chair extends Furniture{
    // Do something
}

class CoffeeTable extends Furniture{
    // Do something
}


interface FurnitureFactory{

    public Furniture createFurniture();
}
```

```java
class TableCreator implements FurnitureFactory{

    @Override
    public Furniture createFurniture() {
        return new Table();
    }
}

class ChairCreator implements FurnitureFactory{

    @Override
    public Furniture createFurniture() {
        return new Chair();
    }
}

class CoffeeTableCreator implements FurnitureFactory{

    @Override
    public Furniture createFurniture() {
        return new CoffeeTable();
    }
}
```