SOLID Principles

What is a Bad Design?

- There are 3 important characteristics that you should avoid in your software design
  - **Rigidity** => Hard to change design, every change affects too many parts in the design
  - **Fragility** => When you make a change in the design, unexpected parts of the system broke
  - **Immobility** => The design components are hard to reuse in another application these components are coupled with each other and cannot be disentangled from the system easily

SOLID Principles

- **S.O.L.I.D** is an acronym for the **first five object-oriented design**(**OOD**)** principles** by Robert C. Martin
- These principles, when combined together, make it easy for a programmer to develop software that are easy to maintain and extend. They also make it easy for developers to avoid code smells, easily refactor code, and are also a part of the agile or adaptive software development.

SOLID Principles

- ***S* => Single responsibility Principle**: A class should have only one reason to change
- ***O* => Open close Principle:** Software entities like classes, modules and functions should be *open for extension but closed for modifications.*
- ***L* => *Liskov substitution principle***: Derived types must be completely substitutable for their base types
- ***I* => Interface segregation principle:** Clients should not be forced to depend upon interfaces that they don't use
- ***D* => Dependency inversion principle:** Abstractions should not depend on details. Details should depend on abstractions

S - Single Responsibility principle

- **One class at the most is responsible for doing one task or functionality among the whole set of responsibilities that it has.**
- **And only when there is a change needed in that specific task or functionality should this class be changed.**

Example

```
public class Employee{
  private String employeeId;
  private String name;
  private string address;
  private Date dateOfJoining;
  public boolean isPromotionDueThisYear(){
    //promotion logic implementation
  }
  public Double calcIncomeTaxForCurrentYear(){
    //income tax logic implementation
  }
  //Getters & Setters for all the private attributes
}
```

Problem

- **This class breaks the Single Responsibility Principle**
- The logic of determining whether the employee promotion is due this year or the income tax calculation is not the employee responsibility.
- Employee class should have the single responsibility of maintaining core attributes of an employee.

Solution

```
public class HRPromotions{
  public boolean isPromotionDueThisYear(Employee emp){
    //promotion logic implementation using the employee information passed
  }
}
public class FinITCalculations{
  public Double calcIncomeTaxForCurrentYear(Employee emp){
    //income tax logic implementation using the employee information passed
  }
}
public class Employee{
  private String employeeId;
  private String name;
  private string address;
  private Date dateOfJoining;
  //Getters & Setters for all the private attributes
}
```

O – Open Close Principle

- **Objects or entities should be open for extension, but closed for modification.**
- *A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients.”*

Example

```
public interface IOperation {

}
```

```
public class Addition implements IOperation
{
    private double firstOperand;
    private double secondOperand;
    private double result = 0.0;

    public Addition(double firstOperand, double secondOperand) {
        this.firstOperand = firstOperand;
        this.secondOperand = secondOperand;
    }

    //Setters and getters
}
```

```
public class Subtraction implements IOperation
{
    private double firstOperand;
    private double secondOperand;
    private double result = 0.0;

    public Subtraction(double firstOperand, double secondOperand) {
        this.firstOperand = firstOperand;
        this.secondOperand = secondOperand;
    }

    //Setters and getters
}
```

Example - Continued

```
public interface ICalculator {
    void calculate(IOperation operation) throws Exception;
}

public class SimpleCalculator implements ICalculator
{
    public void calculate(IOperation operation) throws Exception
    {
        if(operation == null) {
            throw new Exception("Some message");
        }

        if(operation instanceof Addition) {
            Addition obj = (Addition) operation;
            obj.setResult(obj.getFirstOperand() + obj.getSecondOperand());
        } else if(operation instanceof Subtraction) {
            Addition obj = (Addition) operation;
            obj.setResult(obj.getFirstOperand() - obj.getSecondOperand());
        }
    }
}
```

Problem

- **This breaks the Open Close Principle**
- Every time a new operation is added, SimpleCalculator class will need to be changed (not close).

Solution

```
public interface IOperation {
    void performOperation();
}

public class Addition implements IOperation
{
    private double firstOperand;
    private double secondOperand;
    private double result = 0.0;

    public Addition(double firstOperand, double secondOperand) {
        this.firstOperand = firstOperand;
        this.secondOperand = secondOperand;
    }

    //Setters and getters

    public void performOperation() {
        result = firstOperand + secondOperand;
    }
}
```

Solution - Continued

```java
public class Subtraction implements IOperation
{
    private double firstOperand;
    private double secondOperand;
    private double result = 0.0;

    public Subtraction(double firstOperand, double secondOperand) {
        this.firstOperand = firstOperand;
        this.secondOperand = secondOperand;
    }

    //Setters and getters

    public void performOperation() {
        result = firstOperand - secondOperand;
    }
}
```

```java
public interface ICalculator {
    void calculate(IOperation operation);
}
```

```java
public class SimpleCalculator implements ICalculator
{

    public void calculate(IOperation operation) throws Except
    {
        if(operation == null) {
            throw new Exception("Some message");
        }

        operation.performOperation();
    }
}
```

References

- https://www.javabrahman.com/programming-principles/liskov-substitution-principal-java-example/
- https://www.oodesign.com/design-principles.html
- https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design
- https://dzone.com/articles/solid-principles-by-examples-liskov-substitution-p
- https://howtodoinjava.com/design-patterns/open-closed-principle/
- https://raygun.com/blog/solid-design-principles/