Προσχεδιασμένος & ευέλικτος προγραμματισμός

S.O.L.I.D Principles

Introduced by Robert Martin (Uncle Bob), named by Michael Feathers

- Make code maintainable
- Add new functionality without breaking the existing ones
- Read & understand code easier, spend more time on develop than on figuring how things work

S.O.L.I.D Principles

- Single Responsibility
- Open-Closed
- Liskov Substitution
- Interface Segregation
- Dependency Inversion

Single Responsibility Principle

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

Single Responsibility Principle

```
public class AreaCalculator{
   public AreaCalculator() { }

   public <T> double sum(List<T> shapes) {
      double sum = 0;
      for (var shape: shapes) {
        if(shape.getClass().getName().equals("Circle"))
        {
            Circle c = (Circle) shape;
            sum += Math.PI * Math.pow(c.radius(), 2);
         } else if
(shape.getClass().getName().equals("Square")) {
            Square c = (Square) shape;
            sum += Math.pow(c.length(), 2);
         }
      }
      return sum;
   }
}
```

```
public class Circle {
   public  int radius;

   public Circle(int radius) {
      this.radius = radius;
   }
}

public class Output {
   public Output() { }

   public void defaultOutput(double value) {
      System.out.println(value);
   }
   public void detailedOutput(double value) {
      System.out.println("The calculated area is: "
+value);
   }
}
```

```
public class Square {
   public int length;

   public Square(int length) {
      this.length = length;
   }
}

public static void main(String[] args) {
   List<Object> shapes = Arrays.asList(
         new Circle(3), new Square(7));

   AreaCalculator calculator = new AreaCalculator();
   Output output = new Output();

   double sum = calculator.sum(shapes);
   output.defaultOutput(sum);
   output.detailedOutput(sum);
}
```

Single Responsibility Principle

- Code is easier to expand
- Code is easier to maintain
- Reduces the possibility of bugs
- Facilitating debugging processes
- Reduces unexpected side-effects
- better organization of the project

Open-Closed Principle

Software entities (classes, modules, functions, etc.) should be able for extension, but closed for modification.

Open-Closed Principle

```
public class AreaCalculator{
   public AreaCalculator() { }

   public <T> double sum(List<T> shapes) {
      double sum = 0;
      for (var shape: shapes) {
        if(shape.getClass().getName().equals("Circle"))
        {
            Circle c = (Circle) shape;
            sum += c.area();
         } else if(shape.getClass().getName().
                  equals("Square")) {
            Square c = (Square) shape;
            sum += c.area();
         }
      }
      return sum;
   }
}
```

```
public class Circle {
   private int radius;
   public Circle(int radius) {
      this.radius = radius;
   }
   public double area() {
      return Math.PI * Math.pow(radius, 2);
   }
}

public class Square {
   private int length;
   public Square(int length) {
      this.length = length;
   }
   public double area() {
      return Math.pow(length, 2);
   }
}
```

Open-Closed Principle

- Reduces possible errors
- Allows extended testing
- Extend existing code
- Deploy new code only

Liskov Substitution Principle

Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.

Derived classes must be substitutable for their base classes.

Liskov Substitution Principle

```java
public class AreaCalculator {
   private List<Object> shapes;
   public AreaCalculator() { }
   public AreaCalculator(List<Object> shapes) {
      this.shapes = shapes;
   }
   public double sum() {
      double sum = 0;
      for (var shape: shapes) {
        if(shape.getClass().getName().equals("Circle")) {
            Circle c = (Circle) shape;
            sum += c.area();
         }
         else if(shape.getClass().getName().equals("Square")) {
            Square c = (Square) shape;
            sum += c.area();
         }
      }
      return sum;
   }
}
```

```java
public class VolumeCalulator extends AreaCalculator{
   private List<Object> shapes;
   public VolumeCalulator(List<Object> shapes) {
      this.shapes = shapes;
   }
   @Override
   public double sum() {
      double sum = 0;
      for (var shape: shapes) {
         System.out.println(shape.getClass().getName());
         if (shape.getClass().getName().equals("Circle")) {
            Circle c = (Circle) shape;
            sum += c.volume();
         } else if (shape.getClass().getName().equals("Square")) {
            Square c = (Square) shape;
            sum += c.volume();
         }
      }
      return sum;
   }
}
```

Liskov Substitution Principle

- Offers consistency on using parent classes or its subclasses without any errors
- Loose coupled code

Interface Segregation Principle

Many client-specific interfaces are better than one general-purpose interface.

Interface Segregation Principle

```
public class AreaCalculator {
   private List<IShape> shapes;
   public AreaCalculator() {

   }

   public AreaCalculator(List<IShape> shapes) {
      this.shapes = shapes;
   }

   public double sum() {
      double sum = 0;

      for (var shape : shapes) {
         sum += shape.area();
      }
      return sum;
   }
}
```

```
public interface IShape {
   public double area();
   public double volume();
}

public class Circle implements IShape {
   private int radius;
   public Circle(int radius) {
      this.radius = radius;
   }

   public double area() {
      return Math.PI * Math.pow(radius, 2);
   }

   public double volume() {
      return Math.PI * Math.pow(radius, 3);
   }
}
```

```
public class Circle implements IShape {
   private int radius;
   public Circle(int radius) { this.radius = radius;  }
   public double area() {
      return Math.PI * Math.pow(radius, 2);
   }
   public double volume() {
      return Math.PI * Math.pow(radius, 3);
   }
}
public static void main(String[] args) {
   List<IShape> shapes = Arrays.asList(
         new Circle(3), new Square(7));
   AreaCalculator calculator = new
                  AreaCalculator(shapes);
   SumOutput output = new SumOutput(calculator);
   output.defaultOutput();
   output.detailedOutput();
}
```

Interface Segregation Principle

- Simplifies the code
- Prevention of coupling
- reduce the required changes
- Reduces the possibility of bugs
- Facilitating debugging processes

Dependency Inversion Principle

Depend on abstractions, not on concretions.

Dependency Inversion Principle

```
public interface ICalculator {
   public double sum();
}
public class AreaCalculator implements ICalculator {
   private List<IShape> shapes;

   public AreaCalculator() {

 }
   public AreaCalculator(List<IShape> shapes) {
      this.shapes = shapes;
   }

   public double sum() {
      double sum = 0;
      for (var shape : shapes) {
         sum += shape.area();
      }
      return sum;
   }
}
```

```
public class SumOutput {
   private ICalculator calculator;
   public SumOutput(ICalculator calculator) {
      this.calculator = calculator;
   }
   public void defaultOutput() {
      System.out.println(calculator.sum());
   }
   public void detailedOutput() {
      System.out.println("The value is: " + calculator.sum());
   }
}
   public static void main(String[] args) {
      List<IShape> shapes = Arrays.asList(new Circle(3), new
Square(7));
      SumOutput output = new SumOutput(new
AreaCalculator(shapes));
      output.defaultOutput();

   }
}
```

Dependency Inversion Principle

- Change an implementation easily
- Changes in one component does not affect others
- Reuse components across multiple applications

“the representatives of things do not share the relationship of the things they represent”

Robert C. Martin (Uncle Bob)

Thank you