# Design Patterns

## SOLID Design Principle:

It is an acronym that represents five different design principles

Write a SQL query to find the region producing the 5th highest revenue from a table called region in the last 28 days.

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

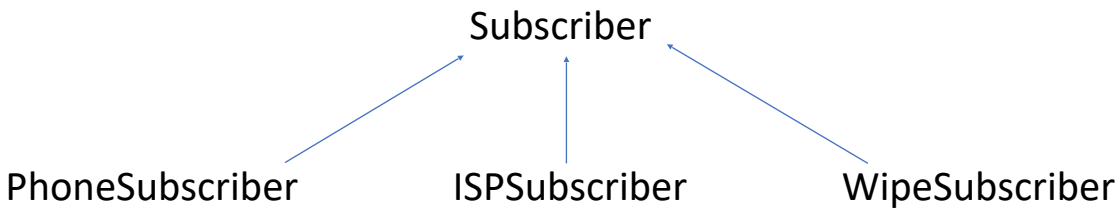
## Single Responsibility Principle

There should never be more than one reason for a class to change.

## Open Closed Principle

Software entities (Classes, Method and Modules) should be open for extension and closed for modification.
We cannot modify existing base class code, but we can extend the existing behavior. (Use Inheritance)

**Example**:



**Liskov Substitution Principle**

We should be substitute base class object with child class object and this should not alter the behavior/characteristics of Program.

**Example**:

Square is special type of rectangle. If we implement child parent relationship between them then it will violate the Liskov substitution Principle.

Before:

```
public class Rectangle {

    private int width;

    private int height;

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

    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 computeArea() {
        return width * height;
    }
}

public class Square extends Rectangle {

    public Square(int side) {
        super(side, side);
    }

    @Override
    public void setWidth(int width) {
        setSide(width);
    }

    @Override
    public void setHeight(int height) {
        setSide(height);
    }

    public void setSide(int side) {
        super.setWidth(side);
        super.setHeight(side);
    }

}

public class Main {

    public static void main(String[] args) {
```

```
        Rectangle rectangle = new Rectangle(10,
20);

System.out.println(rectangle.computeArea());

        Square square = new Square(10);

System.out.println(square.computeArea());

        useRectangle(rectangle);

        useRectangle(square);

    }

    private static void useRectangle(Rectangle
rectangle) {
        rectangle.setHeight(20);
        rectangle.setWidth(30);
        assert rectangle.getHeight() == 20 :
"Height Not equal to 20";
        assert rectangle.getWidth() == 30 :
"Width Not equal to 30";
    }
}
```

After:

```
public interface Shape {

    public int computeArea();

}

public class Square implements Shape {

    private int side;
```

```
    public Square(int side) {
        this.side = side;
    }

    public void setSide(int side) {
        this.side = side;
    }

    public int getSide() {
        return side;
    }

    @Override
    public int computeArea() {
        return side*side;
    }
}

public class Rectangle implements Shape {

    private int width;

    private int height;

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

    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 computeArea() {
        return width * height;
    }
}

public class Main {

    public static void main(String[] args) {

        Rectangle rectangle = new Rectangle(10,
20);

System.out.println(rectangle.computeArea());

        Square square = new Square(10);

System.out.println(square.computeArea());

        useRectangle(rectangle);

    }

    private static void useRectangle(Rectangle
rectangle) {
        rectangle.setHeight(20);
        rectangle.setWidth(30);
        assert rectangle.getHeight() == 20 :
"Height Not equal to 20";
        assert rectangle.getWidth() == 30 :
"Width Not equal to 30";
    }}
```

**Interface Segregation Principle**

Client should not be forced to depend upon interfaces (methods inside interface) that they do not use.

Avoid interface pollution. Write highly cohesive interfaces.

Do not provide a method i.e. not useful for implementation classes.

Try to avoid

@Override
Public m1(){
throw new UnsupportedOperationException("This method is not supported")
}

**Dependency Inversion Principle:**

High level modules should not depend upon low level modules, both should be depending upon abstractions.

Abstraction should not depend upon details. Details should be depending upon abstraction.

**Dependency**:

```
public void writeReport(){
Report report = new Report();
JSONFormatter formatter = new JSONFormatter();
String report = formatter.format(report);
FileWriter writer = new
FileWriter(report.json);
```

}

We are creating objects for JSONFormatter and Writer, so we are tightly coupling these objects.

```
public void writeReport(Formatter formatter,Writter writter){

Report report = new Report();

String report = formatter.format(report);
Writer.write("myreport");

}
```

Write code with interfaces.