

## Research Assignment Week -7

1. Research the SOLID principles of Object-Oriented Programming (OOP) as introduced by Robert Martin


In the field of software engineering, **SOLID** stands for the acronym of five design fundamentals. It's main objective is to make the software engineering more understable, flexible, robust and maintainable. Following these principles in coding means avoiding bad coding habits.

They are:

- S- Stands for the single responsibility principle
- O- Stands for the open-closed principle
- L- Stands for Likskov substitution principle
- I – Stands for interface segregation principle
- D- Stands for dependency inversion Principle

### 1. single responsibility principle

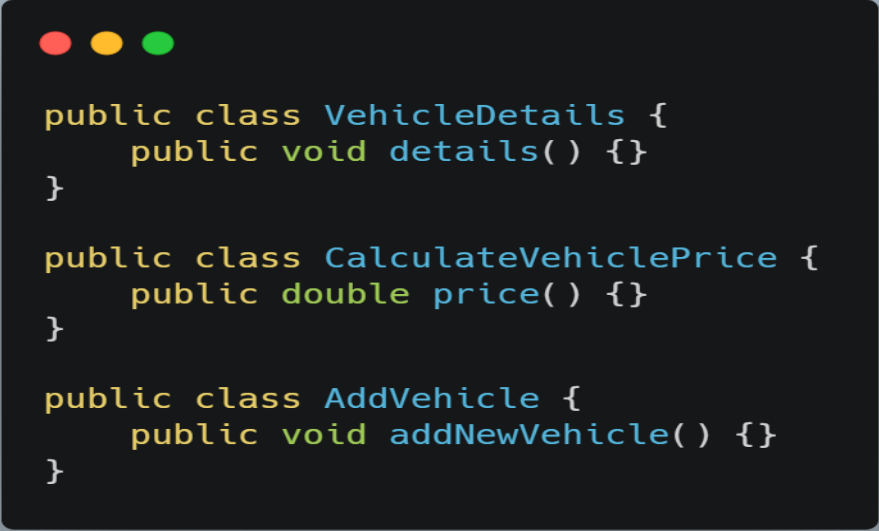
This principle states that if a class should have to change, it must be for one reason. In other ways, every class should have a single responsibility.



```
public class Vehicle {  
    public void details() {}  
    public double price() {}  
    public void addNewVehicle() {}  
}
```

For example, in the above coding, the vehicle class has three responsibilities. Therefore, it has three reasons to change. These are doing with details, price and adding new vehicle to the database.

Therefore, by the principle of single responsibility, we should implement a separate class that performs a single functionality. Example, see the below coding.




```
public class VehicleDetails {  
    public void details() {}  
}  
  
public class CalculateVehiclePrice {  
    public double price() {}  
}  
  
public class AddVehicle {  
    public void addNewVehicle() {}  
}
```

## 2. Open-Closed principle

This principle states that each entity in software, such as classes, should be open for extension, but it should be closed for modification. It means that without modification anything inside the class, it should be extendable.

For example, let us see the below coding.




```
public class NotificationService{  
    public void sendNotification(String medium) {  
        if (medium.equals("email")) {}  
    }  
}
```

In the above example, if you want to introduce a new medium other than email, let's say if you want to send a notification to a mobile number, then you will be forced to change the source code in the NotificationService class.

Therefore, to overcome this problem, a software designer needs to code in such a way that every one can reuse the code in the future by extending it and if they need any customization, they extend the class and add a new feature to it. For example, you can start your code by creating an

interface instead of a class. See the below example.

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following Java code:

```
public interface NotificationService {  
    public void sendNotification(String medium);  
}
```


Then, from this interface, we can implement and develop an Email and Mobile classes. Example, see the following code.

Email Class

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. It contains the following Java code:

```
public class EmailNotification implements NotificationService {  
    public void sendNotification(String medium){  
        // write Logic using for sending email  
    }  
}
```

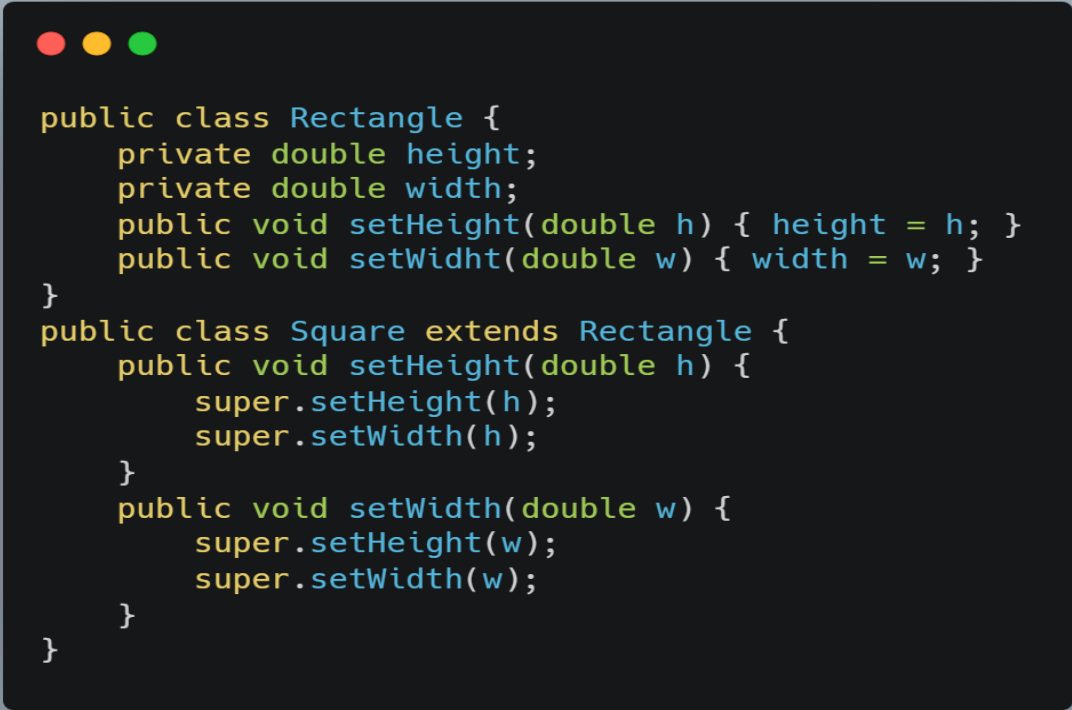
## Mobile Class



```
public class MobileNotification implements NotificationService {  
    public void sendNotification(String medium){  
        // write Logic using for sending notification via mobile  
    }  
}
```

### 3. Liskov Substitution Principle

This principle of software engineering states that derived classes must be able to substitute for their base classes. In other words, if class B is a child class of A, then we should be able to replace A with B without interrupting the current behavior of the program. In conclusion, the Liskov substitute principle wants that the object of subclasses behaves the same way as its super classes. See the example below.




```
public class Rectangle {
    private double height;
    private double width;
    public void setHeight(double h) { height = h; }
    public void setWidht(double w) { width = w; }
}
public class Square extends Rectangle {
    public void setHeight(double h) {
        super.setHeight(h);
        super.setWidth(h);
    }
    public void setWidth(double w) {
        super.setHeight(w);
        super.setWidth(w);
    }
}
```

#### 4. Interface Segregation Principle

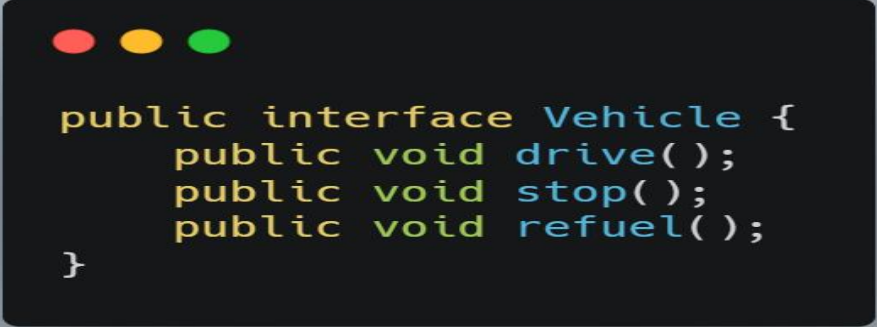
The principle of interface segregation principle states that a client should never be forced to implement an interface that doesn't use, and/or clients shouldn't depend on methods that doesn't use. See the below codes for example.

Let's say we have an interface called vehicle. And this vehicle interface has four unimplemented methods. And let's say from that interface we implement and developed a bike class. But since bike has no doors, we can't implement the openDoors() method in the bike class.




```
public interface Vehicle {  
    public void drive();  
    public void stop();  
    public void refuel();  
    public void openDoors();  
}
```

Therefore, to handle this situation, the software engineering principles recommended us to break down the interfaces into small multiple interfaces, so that no class is forced to implement any methods of the interfaces of no use of it. Please see the below codes on how we break down in different interfaces.



```
public interface Vehicle {  
    public void drive();  
    public void stop();  
    public void refuel();  
}
```



```
public interface Doors{  
    public void openDoors();  
}
```





```
public class Car implements Vehicle, Door {  
    public void drive() {}  
    public void stop() {}  
    public void refuel() {}  
    public void openDoors() {}  
}
```



```
public class Bike implements Vehicle {  
    public void drive() {}  
    public void stop() {}  
    public void refuel() {}  
}
```

## **5. Dependency Inversion Principle (DIP)**

The states that entities must depend on abstraction (abstract classes and interfaces), not on concrete implementation or classes. In addition to that the high-level module mustn't to depend on the low-level module. However, both could depend on abstraction. Therefore, happily we can say that abstraction should not depend on details. On the other hand, details should depend on abstraction.

Example, in the below code of class calculator, if we try to add the multiply method, we will be breaking the principle of SOLID which is Open-Close Principle(OCP). Therefore, we must create an interface of calculator.



```
public class Calculator
{
    public double Add(double x, double y)
    {
        return x + y;
    }
    public double Subtract(double x, double y)
    {
        return x - y;
    }
}
```

2.What are wildcards in MySQL? How are they useful?

The wildcards in MySQL are characters that help us to search data from a table easily and quickly. We can use it with string by substituting one or more characters and produce the result after matching the string into the table.

2. What is your favorite thing you learned this week?

My favorite thing that I learned this week was about MYSQL and Maven.

Reference: <https://medium.com/@kedren.villena/simplifying-dependency-inversion-principle-dip-59228122649a>

Horstmann, C. S. (2016). Core java (Tenth ed.). Prentice Hall.