SOLID Principles

What are SOLID Principles?

- The idea of SOLID Principles was first introduced by the famous Computer Scientist **Robert J. Martin** in the year 2000, while the acronym itself, **SOLID**, was given by **Michael Feathers**.
- The purpose of SOLID Principles is:

  > *"To create understandable, readable, and testable code that many developers can collaboratively work on."*

- So, the SOLID principles can be treated as a set of guidelines which helps us improve the design and maintainability of the software, effectively reducing the developer efforts.
- Let us learn about each of these principles one by one:
  - **S**ingle Responsibility Principle
  - **O**pen-Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principle

Note:

Kindly treat the examples of each of these principles separately as they are meant for better understanding of the individual concepts rather than what actually works in a production environment.

1. Single Responsibility Principle

- A class should have only one reason to change by only one of the stakeholders or group of stakeholders.
- Each class or module should have a single responsibility or job.
- This separation of responsibilities improves maintainability of the code.
- A class with single responsibility can have fewer test cases.

```
// Bad Example
class BillManagement { // Multiple responsibilities given to a single class
  private void calculateTotal() { // Generally used for internally calculating the total bill amount
    // Generate the total bill
  }

  public void printInvoice() { // Generally used by the cashier
    // Print the invoice to be given to the customer
    // public void calculateTotal();
  }

  public void storeToDatabase() { // Generally used by the Database Engineer
```

```
    // Store the invoice details into the database
    // public void calculateTotal();
  }
}
```

```


/*
Let us assume that the database engineer wants to add some functionality
e.g. add the payment method column to indicate whether a particular payment was
made using UPI, cash, card or Internet Banking, this info may not be relevant for
the other stakeholders (e.g. the cashier)
OR
add a functionality to verify if the bill is paid by the customer before leaving
the premises,
this is not relevant for the database engineer as only paid bills will
be forwarded to the database and stored for future reference
 */

// Good Example
class BillCalculator { // Individual responsibilities given to separate classes
  public void calculateTotal() { // Generally used for internally calculating the
total bill amount
    // Generate the total bill
  }
}

class BillPrinter { // Individual responsibilities given to separate classes
  public void printInvoice() { // Generally used by the cashier
    // Print the invoice to be given to the customer
  }
  public boolean isBillPaid() {
    boolean paymentStatus = false;
    // Verify the payment has been completed
    // Return true if that is the case
    // Return false otherwise
    return paymentStatus;
  }
}

class InvoiceToDatabase { // Individual responsibilities given to separate classes
  public void storeToDatabase() { // Generally used by the Database Engineer
    // Store the invoice details into the database
  }
  public String paymentMethod() {
    String paymentMethod = "";
    // Check whether payment was made using UPI, cash, card or Internet Banking
    return paymentMethod;
  }
}

/*
```

```
So basically, only one stakeholder or one group of stakeholders should change the
method
to avoid making changes that break the existing implementation of the software
 */
```

- Note: It does not mean that a class should have only one public method, it implies that a module should be changed by only one business stakeholder or a group of stakeholders.
- Design is subjective, there is no single right answer. If you keep decomposing, you will end up with a lot of classes. It is up to your judgement to decompose or combine the functionalities, but it is necessary that the software is fulfilling the stated business requirements and is also easy to maintain or change.

2. Open-Closed Principle

- Open-Closed Principle states that the classes should be open for extension but closed for modification.
- In this way, we can avoid modifying existing code in a manner that breaks the currently working application.

```
// Bad Example
class Vehicle {
  int numberOfWheels;

  public void moveForwards() {

  }

  public void moveBackwards() {

  }
}

class Car extends Vehicle {
  public void carryPassengers() {
    // Cars are generally meant for carrying people from one location to another
  }
}

class Truck extends Vehicle {
  public void transportGoods() {
    // Trucks are generally used for transporting goods from one location to
another
  }
}

class TowTruck extends Vehicle {
  // However, this needs additional accessories like hook / winch
  // How do you proceed here?
  // Would you add the hook / winch accessory in the vehicle class itself?
  public void towVehicles() {
    // Tow trucks are used for towing vehicles
    }
}
```

```
// Good Example
class Vehicle {
  int numberOfWheels;

  public void moveForwards() {

  }

  public void moveBackwards() {

  }
}

class Car extends Vehicle {
  public void carryPassengers() {
    // Cars are generally meant for carrying people from one location to another
  }
}

class Truck extends Vehicle {
  public void transportGoods() {
    // Trucks are generally used for transporting goods from one location to
another
  }
}

class TowTruck extends Vehicle {
  private Accesory accesory;
  public void towVehicles() {
    // Tow trucks are used for towing vehicles
  }
}
```

3. Liskov Substitution Principle

- It states that:

> ```
> “Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should
> be true for objects y of type S where S is a subtype of T”.
> ```

- Replacing objects of superclass with objects of a subclass in a program should not affect the program's functionality.

```
// Bad Example
class Pen {
  public void write() {
    System.out.println("A Pen can write");
  }
  public boolean isSafeForAirTravel() {
    return true; // We're assuming all pens are safe for air travel
  }
}

class FountainPen extends Pen{
  @Override
  public void write() {
    System.out.println("A FountainPen can also write");
  }

  @Override
  public boolean isSafeForAirTravel() {
    return false;
  }
}

class BallPointPen extends Pen {
  @Override
  public void write() {
    System.out.println("A BallPointPen can also write");
  }
  public boolean isSafeForAirTravel() {
    return true;
  }
}
```

```
// Good example
class Pen {
  public void write() {
    System.out.println("A Pen can write");
  }
}

interface AirTravel {
  public boolean isSafeForAirTravel();
}

class FountainPen extends Pen /* implements AirTravel */{
  @Override
  public void write() {
    System.out.println("A FountainPen can also write");
  }
}

class BallPointPen extends Pen implements AirTravel{
```

```java
    @Override
    public void write() {
      System.out.println("A BallPointPen can also write");
    }

    @Override
    public boolean isSafeForAirTravel() {
      return true;
    }
  }
```

4. Interface Segregation Principle

- The Interface Segregation Principle states that clients should not be forced to implement interface methods that are not useful / applicable to them.
- It simply means that larger interfaces should be decomposed into smaller ones.
- In this way, we can ensure that the classes only implement interfaces needed and are not forced to implement unnecessary methods.

```java
// Bad Example
interface Vehicle {
  public void moveForwards();
  public void moveBackwards();
  public void refuel();
  public void openDoors();
}

class Car implements Vehicle {
  @Override
  public void moveForwards() {

  }
  @Override
  public void moveBackwards() {

  }
  @Override
  public void refuel() {

  }
  @Override
  public void openDoors() {

  }
}

class Bike implements Vehicle {
  @Override
  public void moveForwards() {
  }
  @Override
  public void moveBackwards() {

  }
  @Override
  public void refuel() {

  }
  /*
  @Override
  public void openDoors() {

  }
  */
}
```

```
// Good Example
interface Vehicle {
  public void moveForwards();
  public void moveBackwards();
  public void refuel();
}
interface hasDoors {
  public void openDoors();
}
class Car implements Vehicle, hasDoors {

  @Override
  public void moveForwards() {

  }
  @Override
  public void moveBackwards() {

  }
  @Override
  public void refuel() {

  }
  @Override
  public void openDoors() {

  }
}

class Bike implements Vehicle {
  @Override
  public void moveForwards() {
  }
  @Override
  public void moveBackwards() {

  }
  @Override
  public void refuel() {

  }
  // No need to Override openDoors() method
}
```

5. Dependency Inversion Principle

- It states that:

> High-level modules should not depend on low-level modules; both should depend on abstractions.

> Abstractions should not depend on details; details should depend on abstractions.

- It refers to the decoupling of software modules.
- In this way, instead of high-level modules depending on low-level modules, both will depend on abstractions.
- The abstractions should not depend on details; instead, the details should depend on abstractions.

```
// Bad Example
class Engine {
    String manufacturer;
    String fuelType;

    public void start() {
    }
}
class Car {
    private Engine engine;
    public Car() {
        this.engine = new Engine();
    }
    public void start() {
        engine.start();
    }
}
```

```
// Good Example
interface Engine {
    String manufacturer = "Hyundai";

    public void start();
}
```

```
interface fuelTypePetrol {
  String fuelType = "Petrol";
}
interface fuelTypeDiesel {
  String fuelType = "Diesel";
}
class PetrolCar implements fuelTypePetrol{
  private Engine engine;
  public Car() {
    this.engine = new Engine();
  }
  public void start() {
    engine.start();
  }
}

class DieselCar implements fuelTypeDiesel{
  private Engine engine;
  public Car() {
    this.engine = new Engine();
  }
  public void start() {
    engine.start();
  }
}
```