SOLID design Principles

SOLID design principles

- Solid design principles are the coding standard guidelines / object oriented design concepts in software development

- These principles help reducing the coupling and complexity in the software programs

- Help improve the value and maintainability of the software

Single responsibility	Any class and object should be changed for only one reason
Open/ closed principle	Any code base should be open for extension and closed for modifications
Liskov substitution principle	Subclasses shouldn't be depending upon the client changes
Interface segregation	No class should be enforced to implement anything that's irrelevant
Dependency inversion	High level modules shouldn't depend on the low level modules

Single responsibility principle

- A class should intended to do only one job

- There can be multiple methods and properties but all should be related

```
class Employee {
public Pay calculatePay() {...}
public void save() {...}
public String describeEmployee() {...}
}
```

Open-closed principle

- Classes should be open for extension but closed for modification

- New functionality to be added as there are requirements

- No modifications to be done on existing items

- This helps in reducing the risk of introducing new bugs

Wrong implementation

```
public class VehicleCalculations {
    public double calculateValue(Vehicle v) {
        if (v instanceof Car) {
            return v.getValue() * 0.8;
        if (v instanceof Bike) {
            return v.getValue() * 0.5;

    }
}
```

Recommended implementation

TestLeaf
Always Ahead

```
public class Vehicle {
    public double calculateValue() {...}
}
public class Car extends Vehicle {
    public double calculateValue() {
        return this.getValue() * 0.8;
}
public class Truck extends Vehicle{
    public double calculateValue() {
        return this.getValue() * 0.9;
}
```

Liskov substitution principle

- LSP applies for inheritance hierarchies

- All the subclasses must operate in the same manner as of base class

- A subtype of a specific super type shouldn’t do anything that super type doesn’t expect

LSP violation

```
public class Rectangle {
  private double height;
  private double width;

  public double area();

  public void setHeight(double height);
  public void setWidth(double width);
}
```

```
public class Square extends Rectangle {
  public void setHeight(double height) {
    super.setHeight(height);
    super.setWidth(height);
  }

  public void setWidth(double width) {
    setHeight(width);
  }
}
```

Interface segregation principle (ISP)

- No class should get enforced to implement an interface that it doesn’t require

- Always implement multiple smaller cohesive interfaces

- This helps reducing the coupling as we minimize the unused dependencies

Wrong implementation

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

    // Can be implemented
    public void drive() {...}
    public void stop() {...}
    public void refuel() {...}

    // Can not be implemented
    public void openDoors() {...}
}
```

Recommended implementation

```
public interface LoginMessenger {
  askForCard();
  tellInvalidCard();
  askForPin();
  tellInvalidPin();
}

public interface WithdrawalMessenger {
  tellNotEnoughMoneyInAccount();
  askForFeeConfirmation();
}

publc class EnglishMessenger implements LoginMessenger, WithdrawalMessen
  ...
}
```

Dependency inversion principle

- This principle states that high level module shouldn’t depend on the low level modules

- The abstractions shouldn’t depend upon the details rather it should be other way around

- All the details are safe and hidden

- It helps in easier design change

Wrong Implementations

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

```
public interface Engine {
    public void start();
}
```

Recommended implementation

TestLeaf
Always Ahead

```
public class Car {
    private Engine engine;
    public Car(Engine e) {
        engine = e;
    }
    public void start() {
        engine.start();
    }
}
public class PetrolEngine implements Engine {
   public void start() {...}
}
public class DieselEngine implements Engine {
   public void start() {...}
}
```