SOLID Principles

(Sursă info*: https://www.educative.io/answers/what-are-the-solid-principles-in-java*)

- **Single Responsibility Principle**

Fiecare clasă ar trebui să fie responsabilă pentru o singură parte sau funcționalitate a sistemului.

- **Principiul Open-Closed**

Componentele software ar trebui să fie deschise pentru extindere, dar nu pentru modificare.

- **Liskov Substitution Principle**

Obiectele unei superclase ar trebui să fie înlocuibile cu obiecte din subclasele sale, fără a modifica sistemul/aplicația.

- **Interface Segregation Principle**

Niciun client nu trebuie să fie forțat să depindă de metodele pe care nu le folosește.

- **Dependency Inversion Principle**

Modulele de nivel înalt nu ar trebui să depindă de modulele de nivel scăzut, ambele ar trebui să depindă de abstracții.

Exemple:

1. **Single Responsibility Principle**

Fiecare clasă din Java ar trebui să aibă o singură responsabilitate. Pentru a fi precis, ar trebui să existe un singur motiv pentru a schimba o clasă. Iată un exemplu de clasă Java care nu urmează principiul responsabilității unice (SRP):

```
public class Vehicle {
    public void printDetails() {}
    public double calculateValue() {}
    public void addVehicleToDB() {}
}
```

Clasa *Vehicle* are trei responsabilități separate: raportare, calcul și bază de date. Prin aplicarea SRP, putem separa clasa de mai sus în trei clase cu responsabilități separate.

2. Open-closed principle

Entitățile software (de exemplu, clase, module, funcții) ar trebui să fie deschise pentru o extensie, dar închise pentru modificare.

```
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;

    }
}
```

Să presupunem că acum dorim să adăugăm o altă subclasă numită *Truck*. Ar trebui să modificăm clasa de mai sus adăugând o altă declarație if, care contravine principiului Open-Closed.

O abordare mai bună ar fi ca subclasele Car și Camion să suprascrie metoda calculateValue:

```
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;
}
```

Adăugarea unui alt tip de vehicul este la fel de simplă ca a face o altă subclasă și extinderea de la clasa de vehicule.

3. Liskov substitution principle

Principiul de substituție Liskov (LSP) se aplică ierarhiilor de moștenire, astfel încât clasele derivate trebuie să fie complet substituibile pentru clasele lor de bază.

Un exemplu tipic de clasă derivată Square și clasă de bază Rectangle:

```
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);
    }
}
```

Clasele de mai sus nu se supun LSP deoarece nu se poate înlocui clasa de bază *Rectangle* cu clasa sa derivată *Square*. Clasa Square are constrângeri suplimentare, adică înălțimea și lățimea trebuie să fie aceleași. Prin urmare, înlocuirea clasei Rectangle cu Square poate duce la un comportament neașteptat.

4. Interface segregation principle

Principiul de segregare a interfeței (ISP) prevede că clienții nu ar trebui să fie forțați să depindă de membrii interfeței pe care nu îi folosesc. Cu alte cuvinte, nu forțați niciun client să implementeze o interfață care este irelevantă pentru el.

Să presupunem că există o interfață pentru vehicul și o clasă *Bike*:

```
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() {...}
}
```

După cum se poate vedea, nu are sens ca o clasă *Bike* să implementeze metoda *openDoors*() deoarece o bicicletă nu are uși. Pentru a remedia acest lucru, ISP propune ca interfețele să fie împărțite în interfețe multiple, mici, coerente, astfel încât nicio clasă să nu fie forțată să implementeze vreo interfață și, prin urmare, metode de care nu are nevoie.

5. Dependency inversion principle

Principiul inversării dependenței (DIP) afirmă că ar trebui să depindem de abstracții (interfețe și clase abstracte) în loc de implementări concrete (clase). Abstracțiile nu ar trebui să depindă de detalii; în schimb, detaliile ar trebui să depindă de abstracții.

În exemplul de mai jos este prezentată o clasă *Car* care depinde de clasa concretă *Engine*; prin urmare, nu se supune DIP.

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

Codul va funcționa, deocamdată, dar dacă am vrea să adăugăm un alt tip de motor, să spunem un motor diesel? Acest lucru va necesita efectuarea „refactoring” asupra clasei Car.

Cu toate acestea, putem rezolva acest lucru prin introducerea unui strat de abstractizare. În loc ca mașina să depindă direct de motor, să adăugăm o interfață:

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

Acum putem conecta orice tip de motor care implementează interfața *Engine* la clasa *Car*:

```
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() {...}
}
```