Java Solid Prensipleri

SOLID ilkeleri , yazılım geliştirmeyle ilgili nesne yönelimli tasarım kavramlarıdır.

SOLID diğer 5 tasarım ilkesinin kısatlamasdır.

- Single Responsibility Principle (Tek sorumluluk ilkesi)
- Open-Closed Principle (Açık – Kapalı İlkesi)
- Liskov Substitution Principle (Liskov Değiştirme İlkesi)
- Interface Segregation Principle(Arayüz Ayrıştırma İlkesi)
- Dependency Inversion Principle(Bağımlılığı Tersine Çevirme İlkesi)

Single Responsibility Principle

Her sınıf, sistemin tek bir parçasından veya işlevselliğinden sorumlu olmalıdır.

Open-Closed Principle

Yazılım bileşenleri, genişletmeye açık olmalı, ancak değiştirilmeye açık olmamalıdır.

Liskov Substitution Principle

Bir üst sınıfın nesneleri, sistemi bozmadan alt sınıflarının nesneleri ile değiştirilebilir olmalıdır.

Interface Segregation Principle

Hiçbir müşteri, kullanmadığı yöntemlere bağımlı olmaya zorlanmamalıdır.

Dependency Inversion Principle

Yüksek seviyeli modüller düşük seviyeli modüllere bağlı olmamalı, her ikisi de soyutlamalara bağlı olmalıdır.

Neden SOLID ?

SOLID, yazılımınızın modüler ve bakımı, anlaşılması, hata ayıklaması ve yeniden düzenlemesi kolay olmasını sağlayan yapılandırılmış bir tasarım yaklaşımıdır.

SOLID'i takip etmek ayrıca hem geliştirme hem de bakımda zamandan ve emekten tasarruf etmenize yardımcı olur.

SOLID, kodunuzun katı ve kırılgan hale gelmesini önleyerek uzun ömürlü yazılımlar oluşturmanıza yardımcı olur.

Examples

1. Single responsibility principle(Tek sorumluluk ilkesi)

Java'daki her sınıfın yapacak tek bir işi olmalıdır. Kesin olmak gerekirse, bir sınıfı değiştirmek için yalnızca bir neden olmalıdır.

İşte tek sorumluluk ilkesine (SRP) uymayan bir Java sınıfı örneği:

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

Araç sınıfının üç ayrı sorumluluğu vardır: raporlama, hesaplama ve veritabanı.

SRP uygulayarak yukarıdaki sınıfı ayrı sorumluluklarla üç sınıfa ayırabiliriz.

2. Open-closed principle(Açık-kapalı prensibi)

Yazılım varlıkları(Software Entities) (örn. sınıflar, modüller, işlevler) bir uzantı için açık, ancak değişiklik için kapalı olmalıdır.

VehicleCalculations sınıfının aşağıdaki yöntemini göz önünde bulundurun:

```
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;

    }
}
```

Şimdi Truck adında başka bir alt sınıf eklemek istediğimizi varsayalım. Açık-Kapalı İlkesine aykırı olan başka bir if ifadesi ekleyerek yukarıdaki sınıfı değiştirmemiz gerekir.

Car ve Truck alt sınıflarının calculateValue metodunu geçersiz kılması(override) daha iyi bir yaklaşım olacaktır:

```
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;
}
```

Başka bir Vehicle türü eklemek, başka bir alt sınıf(subclass) oluşturmak ve Vehicle sınıfından genişletmek kadar basittir.

3. Liskov substitution principle(Liskov Değiştirme İlkesi)

Liskov Değiştirme İlkesi (LSP), türetilmiş sınıfların temel sınıfları için tamamen değiştirilebilir olması gerektiği şekilde kalıtım hiyerarşileri için geçerlidir.

Square'den türetilmiş bir sınıf ve Rectangle temel sınıfının tipik bir örneğini düşünün:

```
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);
    }
}
```

Yukarıdaki sınıflar LSP'ye uymaz çünkü Rectangle temel sınıfını türetilmiş Square sınıfıyla değiştiremezsiniz. Square sınıfının ekstra kısıtlamaları vardır, yani yükseklik ve genişlik aynı olmalıdır.

Bu nedenle, Rectangle sınıfının Square sınıfıyla değiştirilmesi beklenmeyen davranışlara neden olabilir.

4. Interface segregation principle(Arayüz ayırma ilkesi)

Arayüz Ayrımı İlkesi (ISP), istemcilerin kullanmadıkları arabirim üyelerine(members) bağımlı olmaya zorlanmaması gerektiğini belirtir.

Başka bir deyişle, herhangi bir istemciyi(Client), kendileriyle ilgisi olmayan bir arabirimi uygulamaya zorlamayın.

Vehicle ve Bike sınıfı için bir arayüz(interface) olduğunu varsayalım:

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

Gördüğünüz gibi bir Bike sınıfının openDoors() yöntemini uygulaması bir bisikletin kapısı olmadığı için mantıklı değil!

Bunu düzeltmek için ISP, arayüzlerin(interface) çoklu, küçük uyumlu arayüzlere(interface) ayrılmasını önerir, böylece hiçbir sınıf herhangi bir arayüzü(interface) ve dolayısıyla ihtiyaç duymadığı yöntemleri(metod) uygulamak zorunda kalmaz.

5. Dependency inversion principle

Bağımlılık Tersine Çevirme İlkesi (DIP), somut uygulamalar (sınıflar) yerine soyutlamalara (arayüzler ve soyut sınıflar) bağlı olmamız gerektiğini belirtir.

Soyutlamalar ayrıntılara bağlı olmamalıdır; bunun yerine, ayrıntılar soyutlamalara dayanmalıdır.

Aşağıdaki örneği düşünün. Somut Engine sınıfına bağlı olan bir Car sınıfımız var; bu nedenle, DIP'ye uymuyor.

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

Kod şimdilik işe yarayacak ama ya dizel motor diyelim başka bir motor tipi eklemek istesek?

Bu, Car sınıfının yeniden düzenlenmesini gerektirecektir. Ancak, bunu bir soyutlama katmanı ekleyerek çözebiliriz.

Doğrudan Engine(motor) bağlı Car yerine bir arayüz(interface) ekleyelim:

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

Artık Engine arabirimini(interface) uygulayan(implement eden) herhangi bir Engine türünü Car sınıfına bağlayabiliriz:

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

Do not depend on concretions, depend on abstractions.

Program to an interface, not an implementation.

Kaynak :

https://www.educative.io

---

https://github.com/f4kanzyCoder