SOLID Prensibleri

Bugüne kadar yazılımcıların çalışarak tecrübe ettiği karşılaştığı sorunlara karşı ürettikleri çözümleri toplayarak hepsini bir araya getirip belli bir prensiplerimizin olduğuna dair bir fikir ürettiler ve bu prensipler üzerinden projelerimizi üretelim diye fikiri benimsediler. Bu kriterleri 5 başlık altında toplamışlar. Bu prensiplerin baş harfleriyle SOLID kısaltımı ortaya çıktı.

- ✓ **S**ingle Responsibility → Tek Sorumluluk
- ✓ **O**pen Closed → Açık / Kapalı
- ✓ **L**iskov Substitution → Yerine Geçme
- ✓ **I**nterface Segregation → Arayüz Ayrımı
- ✓ **D**ependency Inversion → Bağımlılıkları Tersine Çevirme

Single Responsibility (Tek Sorumluluk**)**

Bir sınıfın veya bir metotun sadece tek bir görevi olması gerektiğini söyler. Yani bir örnek vermek gerekirse diyelim ki bizim eposta diye bir sınıfımız var. Bu eposta sınıfının içinde gönder diye bir metotumuz var. Bizim bu eposta sınıfı **sadece gönderme işini yapmalı!** Yani eposta gönder metotu sadece epostayı gönderiyor olmalı. Yani eposta gönder sınıfı içinde başka bir işlem mesela epostaları oku işlemi olmamalıdır. Yani 1 sınıf sadece tek bir şeyden sorumlu olmalıdır. Yani yarın-öbür gün o mevcut işlemde bir sorun olduğunda onu bulmak çok daha kolay ve onu tamir etmek çok daha basit olur.

Java'daki her sınıfın yapacak tek bir işi olmalıdır. Kesin olmak gerekirse, bir sınıfı değiştirmek için yalnızca bir neden olmalıdır. Aşağıda, tek sorumluluk ilkesine (SRP) uymayan bir Java sınıfı örneği verilmiştir:

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

Sınıfın üç ayrı sorumluluğu vardır: raporlama, hesaplama ve veritabanı. SRP'yi uygulayarak, yukarıdaki sınıfı ayrı sorumlulukları olan üç sınıfa ayırabiliriz.

Open Closed (Açık / Kapalı**)**

Bir proje büyümeye-genişlemeye yani geliştirilmeye açık olmalı ama mevcuttaki kodların değiştirilmesine kapalı olmalı.

Yazılım varlıkları (örneğin, sınıflar, modüller, işlevler) bir uzantı için *açık* olmalı, ancak değişiklik için *kapalı* olmalıdır.

```
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 başka bir alt sınıf eklemek istediğimizi varsayalım. Yukarıdaki sınıfı, Açık-Kapalı İlkesine aykırı olan başka bir if ifadesi ekleyerek değiştirmemiz gerekir. Alt sınıflar için ve yöntemi geçersiz kılmak için 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 tür eklemek, başka bir alt sınıf oluşturmak ve sınıftan genişletmek kadar basittir.

Liskov Substitution (Yerine Geçme**)**

Bir türden başka bir tür türetiyorsak bu türettiğimiz türler arasında işlem bazında farklılıklar olmalı. Yani biri birinin yerine geçtiğinde hata oluşmamalı.

Türetilmiş sınıfların temel sınıfları için tamamen değiştirilebilir olması gerektiği şekilde devralma hiyerarşileri için geçerlidir. Türetilmiş bir sınıf ve temelin 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);
    }
}
```

Temel sınıfı türetilmiş sınıfıyla değiştiremediğiniz için yukarıdaki sınıflar lsp'ye uymaz. Sınıfın ek kısıtlamaları vardır, yani yükseklik ve genişlik aynı olmalıdır. Bu nedenle, sınıfla değiştirme beklenmeyen davranışlara neden olabilir.

Interface Segregation (Arayüz Ayrımı**)**

Bir interface içinde gereksiz metotların kullanılmaması. Yani bir interface'imiz var. Mesela eposta ile ilgili bir interface'imiz Varsa orada kullanıcıları getir diye bir metotumuzun olmaması gerekiyor. Kısacası alakasız işlerin olmaması gerekiyor.

İstemcilerin kullanmadıkları arabirim üyelerine bağımlı olmaya zorlanmaması gerektiğini belirtir. Başka bir deyişle, herhangi bir istemciyi kendileriyle alakasız bir arabirim uygulamaya zorlamayın.

```
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 bisikletin kapısı olmadığı için bir sınıfın yöntemi uygulaması mantıklı değil! Bunu düzeltmek için Interface Segregation, arabirimlerin birden çok küçük birleşik arabirime ayrılmasını önerir; böylece hiçbir sınıf, herhangi bir arabirimi ve dolayısıyla ihtiyaç duymadığı yöntemleri uygulamak zorunda kalmaz.

Dependency Inversion (Bağımlılıkları Tersine Çevirme**)**

Bir sınıf başka bir sınıfa bağımlı olmamalı. Yani üst interface'lerden implement edilerek interface'ler kullanılmalı. Yani bir sınıftan başka bir sınıfı new'lemeyin.

Bağımlılık Ters Çevirme İlkesi (DIP), somut uygulamalar (sınıflar) yerine soyutlamalara (arabirimler ve soyut sınıflar) bağımlı olmamız gerektiğini belirtir. Soyutlamalar ayrıntılara bağlı olmamalıdır; bunun yerine, ayrıntılar soyutlamalara bağlı olmalıdır.

Aşağıdaki örneği göz önünde bulundurun. Somut sınıfa bağlı bir sınıfımız var; bu nedenle Dip'e 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 başka bir motor tipi eklemek istiyorsak, buna dizel motor diyelim mi?
Bu, sınıfın yeniden düzenlenmesini gerektirecektir.
Ancak, bunu bir soyutlama katmanı getirerek çözebiliriz. Doğrudan bağlı olmak yerine, bir arayüz ekleyelim:

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

Artık Motor arabirimini uygulayan herhangi bir türü sınıfa 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() {...}
}
```