SOLID Prensipleri

Doç. Dr. Üyesi Fatih ÖZYURT

Fırat Üniversitesi Yazılım Mühendisliği Bölümü

ISO 9126 Kalite Faktörleri

Prensipler

SOLID yazılım prensipleri;

- geliştirilen yazılımın esnek,
- yeniden kullanılabilir,
- sürdürülebilir ve anlaşılır olmasını sağlayan,
- kod tekrarını önleyen

prensipler bütünüdür.

Amaç

Kısaltması Michael Feathers tarafından tanımlanan bu prensiplerin amacı;

- Geliştirdiğimiz yazılımın gelecekte gereksinimlere kolayca adapte olması,
- Yeni özellikleri kodda bir değişikliğe gerek kalmadan kolayca eklenebilmesi,
- Yeni gereksinimlere karşın kodun üzerinde en az değişimi sağlaması,
- Kod üzerinde sürekli düzeltme hatta yeniden yazma gibi sorunların yol açtığı zaman kaybını da minimuma indirmektir.

Bu prensipler uygulanarak uygulamalarımızın büyürken, karmaşıklığın da büyümesinin önüne geçmiş oluruz. “**kaliteli kod**” yazmak için bu prensiplere uygun yazılım geliştirmeliyiz.

SOLID

S — Single-responsibility principle

O — Open-closed principle

L — Liskov substitution principle

I — Interface segregation principle

D — Dependency Inversion Principle

S — Single-responsibility principle (Tek Sorumluluk Prensibi)

Single Responsibility; Tek işi, tek sorumlulukta yapma sanatı…

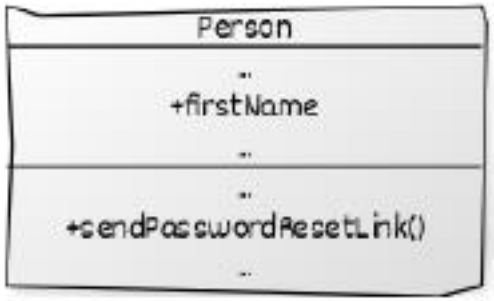
Her sınıf, metot, fonksiyon tek bir sorumluluğa sahip olmalıdır.

Şayet bu kurala uymazsak ilerleyen süreçte bir değişikliğe gidildiğinde bunun etkisini birçok yerde görmüş oluruz.

Nedeni ise bir yapıya birden fazla sorumluluk yüklenmesinden dolayıdır.

Eğer değişikliklerden etkilenen yerler arasında sistemin birçok yerinde kullanılan bir yapımız da varsa maliyet gittikçe artacaktır.

S — Single-responsibility principle (Tek Sorumluluk Prensibi)




```java
public class Person {
    public String firstName;

    public void sendPasswordResetLink() {
        ...
    }
}
```

Yukarıdaki diyagrama ve koda baktığımızda `Person` sınıfı içerisinde `sendPasswordResetLink()` diye bir metot bulunmaktadır. Bu sınıfın asıl amacı kişilere ait bilgileri tutmaktır, şifre sıfırlama bağlantısı göndermek değil.

Birden fazla sorumluluk yüklendiği için olası bir mail gönderme değişikliğinde bu sınıf da etkilenecektir.

S — Single-responsibility principle (Tek Sorumluluk Prensibi)

Yukarıdaki UML diyagramını biraz daha düzenlersek aşağıdaki gibi bir yapı elde edilir.

JAVA Kod Orneği:

```
class Person {
     public String firstName;
}

class EmailService {
    public void sendPasswordResetLink(Person person) {
        ...
    }
}
```

O — Open/Closed Principle (Açık Kapalı Prensibi)

Sınıflarımız/fonksiyonlarımız değişikliğe kapalı ancak yeni davranışların eklenmesine açık olmalıdır.

Bu prensip; sürdürülebilir ve tekrar kullanılabilir yapıda kod yazmanın temelini oluşturur.
Robert C. Martin

Open Sınıf için yeni davranışlar eklenebilmesini sağlar. Gereksinimler değiştğinde, yeni gereksinimlerin karşılanabilmesi için bir sınıfa yeni veya farklı davranışlar eklenebilir olmasıdır.
Closed Bir sınıf temel özelliklerinin değişimi ise mümkün olmamalıdır.

O — Open/Closed Principle (Açık Kapalı Prensibi)

- Geliştirdiğimiz yazılıma/sınıfa var olan kodu değiştirmeden, yeni kod yazılarak yeni özellikler eklenebilmelidir.

- Yeni bir gereksinim geldiğinde mevcut kod üzerinde herhangi bir değişiklik yapıyorsanız, open/closed prensibine ters düşüp düşmediğinizi kontrol etmenizde yarar var.

- Yazılımı geliştirirken gelecekte oluşabilecek özellikler ve geliştirmeleri her şeyiyle öngöremeyiz. O yüzden oluşabileceğini düşündüğümüz kodları da şimdiden geliştirmemeliyiz.

- Yeni gelecek özellikler için var olan kodu değiştirmeden, var olan yapıyı bozmadan esnek bir geliştirme modeli uygulayarak, önü açık ve gelecekten gereksinimlere kolayca adapte olup, ayak uydurabilen bir model uygulamalıyız.

O — Open/Closed Principle (Açık Kapalı Prensibi)

Single Responsibility'e sahip bir Alan hesaplayıcı program yazmaya başladığımızı varsayalım ancak sadece dikdörtgen hesabına ihtiyacımız oldugunu düşünelim.

```
public class Rectangle {


 private double length;
 private double height;
 // getters/setters ...
}
public class AreaService {

    public double calculateArea(List<Rectangle>... shapes) {
        double area = 0;
        for (Rectangle rect : shapes) {
            area += (rect.getLength() * rect.getHeight());
        }
        return area;
    }
}
```

Area servisimiz dikdörtgen hesabını yapacak seviyede, ancak bizim ihtiyaçlarımız daire hesabının da yapılmasını da gerektirmeye başladı diyelim.

O — Open/Closed Principle (Açık Kapalı Prensibi)

```
public class Circle {
 private double radius;
 // getters/setters …
}
```

Daire alan hesabı yapabilmek için AreaService'i içindeki calculateArea methodumuzda değişiklik yapmamız gerekti.

```
public class AreaService {
    public double calculateArea(List<Object>... shapes) {
        double area = 0;
        for (Object shape : shapes) {
            if (shape instanceof Rectangle) {
                Rectangle rect = (Rectangle) shape;
                area += (rect.getLength() * rect.getHeight());
            } else if (shape instanceof Circle) {
                Circle circle = (Circle) shape;
                area += circle.getRadius() * cirlce.getRadius() *
Math.PI;
            } else {
                throw new RuntimeException("Shape not supported");
            }
        } return area;
    }
}
```

O — Open/Closed Principle (Açık Kapalı Prensibi)

Yeni bir şekil eklemek istediğimizde **üçgen** gibi sürekli bu metot üzerinde değişiklikler yapacağız ve durum giderek kötüleşecek ve oluşan durum **open/closed** prensibine **uymadığımızı** gösteriyor.

Bu durum için sınıfımız/metodumuz değişikliğe kapalı değil aksine değişiklik zorunlu hale gelmiştir.

Genişleme ise seçenekler arasında değildir. Her yeni şekil eklenmesi için AreaService üzerinde **değişikliğe** gitmemiz gerekiyor.

O — Open/Closed Principle (Açık Kapalı Prensibi)

AreaService tüm şekil tiplerinin alan hesabını yapmakla yükümlü ancak her alanın da kendine özgü bir hesaplama yöntemi mevcut, bu cümleden de anlaşılacağı üzere her şekil için farklı hesaplama yöntemi, her şekil için kendi içlerinde hesaplama gerekliliğini doğurmaktadır.

Bunu çözmek için bir Shape interface'imiz olsa ve her bir şekil için hesaplanmış area'yı dönse nasıl olur ?

```
public interface Shape {
   double getArea();
}
```

Her şekil Shape üzerinden türetilmelidir. Burada açıkça görüyorum ki; Şekillerden biri olan Dikdörtgen alan hesabını getArea metodumla öğrenebilirim.

O — Open/Closed Principle (Açık Kapalı Prensibi)

```
public class Rectangle implements Shape {
    private double length;
    private double height;

    // getters/setters …
    @Override
    public double getArea() {
        return (length * height);
    }
}

public class Circle implements Shape {
    private double radius;

    // getters/setters …
    @Override
    public double getArea() {
        return (radius * radius * Math.PI);
    }
}

public class AreaManager {
    public double calculateArea(List<Shape> shapes) {
        double area = 0;
        for (Shape shape : shapes) {
            area += shape.getArea();
        }
        return area;
    }
}
```

Artık programımız Open/Closed prensibine uygun hale gelmiştir.
Herhangi bir yeni şekil alanı hesaplamamız gerektiğinde yapmamız gereken **AreaService** üzerinde değişiklik değil, ki değişikliğe kapalı olmalıyız.
Shape nesnemizden yeni şekli türetmemiz ve alan hesabını kendi içinde yapmamızdır.
Böylece genişlemeye açık oluyoruz ve hiç bi yerde değişikllik yapmamıza gerek kalmıyor.

L — Liskov Substitution (Yerine Geçebilme)

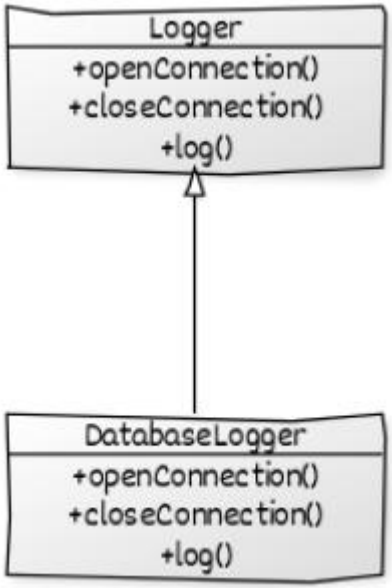
Kodumuzda herhangi bir değişiklik yapmaya gerek kalmadan türetilmiş sınıfları (sub class) türedikleri ata sınıfın (base class) yerine kullanabilmeliyiz.

"Alt seviye sınıflardan oluşan nesnelerin/sınıfların, ana(üst) sınıfın nesneleri ile yer değiştirdikleri zaman, aynı davranışı sergilemesi gerekmektedir. Türetilen sınıflar, türeyen sınıfların tüm özelliklerini kullanabilmelidir."

Alt sınıflar, üst sınıflardan türediği için onların davranışlarını devralırlar. Eğer üst sınflara ait davranışları gerçekleştirmiyorlarsa davranışı yapan metotu muhtemelen boş bırakır ya da bir hata fırlatırız fakat bu işlemler kod kirliliğine ve gereksiz kod kalabalığına neden olmaktadır.

L — Liskov Substitution (Yerine Geçebilme)

Bunların yanı sıra projeye daha sonradan dahil olacak geliştiriciler için de sorun oluşturmaktadır. Geliştirici, sistemin sağlıklı yürüdüğünü düşünerek gerçekleştirilmeyen bir davranışı kullanmaya çalışabilir.

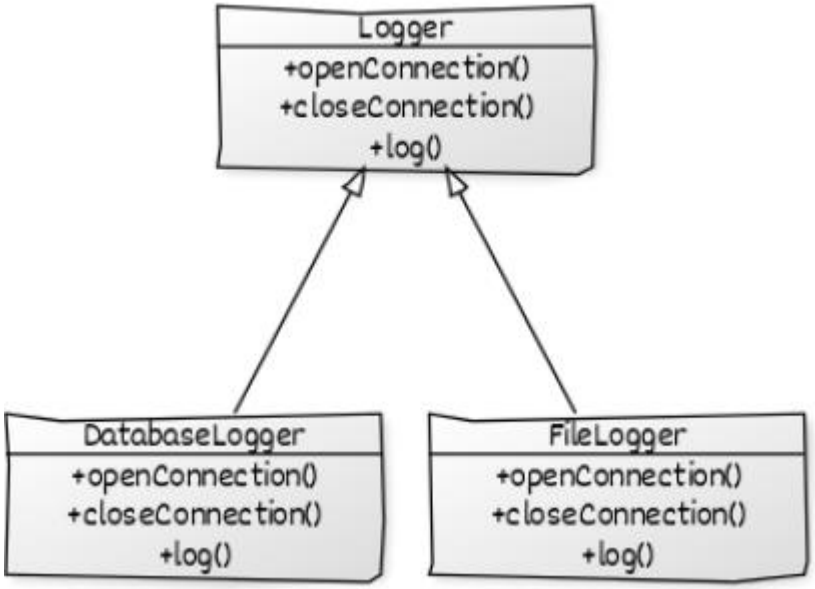



```java
abstract class Logger {
    public abstract void openConnection();
    public abstract void closeConnection();
    public abstract void log();
}
```

```java
class DatabaseLogger extends Logger {
    @Override
    public void openConnection() {
        ...
    }

    @Override
    public void closeConnection() {
        ...
    }

    @Override
    public void log() {
        openConnection();
        // LOG
        closeConnection();
    }
}
```

L — Liskov Substitution (Yerine Geçebilme)

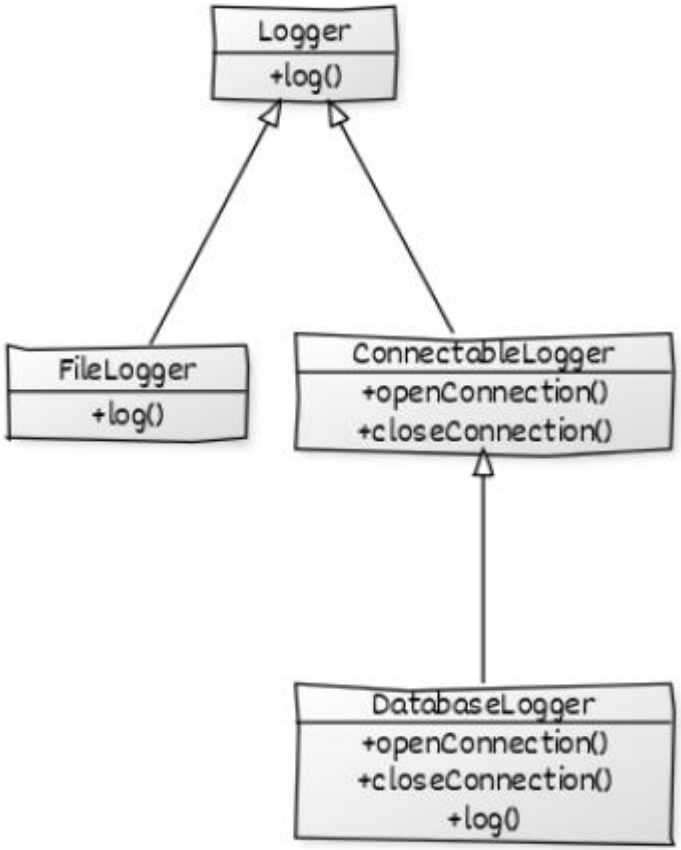
koda baktığımız zaman DatabaseLogger sınıfımız, Logger adlı sınıftan türemektedir. Başlangıç aşaması için bir problem görünmezken ilerleyen zamanlarda veri tabanı değil de bir dosyaya kayıt işlemi alınacağı zaman aşağıdaki gibi bir görünüm meydana gelecektir.




```java
class FileLogger extends Logger {
    @Override
    public void openConnection() {
        new Exception("Not implemented!");
    }

    @Override
    public void closeConnection() {
        new Exception("Not implemented!");
    }

    @Override
    public void log() {
        // LOG
    }
}
```

L — Liskov Substitution (Yerine Geçebilme)

bağlantı açma ve kapatma işlemleri veri tabanına aittir, bir dosyaya değil. Gereksiz hata fırlatmaları, kodun okunmasındaki zorluk, kod kalabalığı gibi birçok olaya neden olmaktadır. Burada bu işlemler bir ara sınıfa alınabilir.




```java
abstract class Logger {
    public abstract void log();
}
```

```java
abstract class ConnectableLogger extends Logger {
    public abstract void openConnection();
    public abstract void closeConnection();
}
```

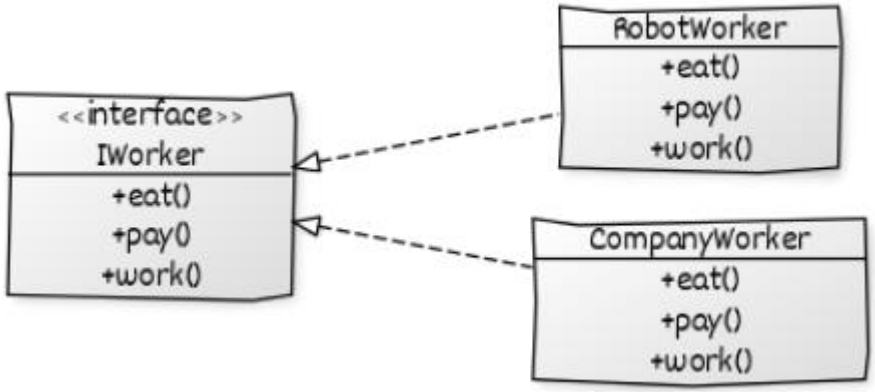
```java
class FileLogger extends Logger {
    @Override
    public void log() {
        // LOG
    }
}
```

```java
class DatabaseLogger extends ConnectableLogger {
    @Override
    public void openConnection() {
        ...
    }

    @Override
    public void closeConnection() {
        ...
    }

    @Override
    public void log() {
        openConnection();
        // LOG
        closeConnection();
    }
}
```

I— Interface Segregation Principle (Arayüz Ayrımı Prensibi)

Sınıflar, kullanmadığı metotları içeren arayüzleri uygulamaya zorlanmamalıdır.

Arayüzlerimizde genel olarak birçok operasyonel işlem barındırabiliriz fakat bu arayüzü uygulayan sınıfların, bazılarını kullanmama durumu olabilmektedir. Bir sınıf birden fazla arayüzü uygulaması özelliğiyle de birlikte bu prensip, bu tür durumlarda arayüzlerin ayrılmasını ve ihtiyaç halinde olanların kullanmasını söylemektedir.



I— Interface Segregation Principle (Arayüz Ayrımı Prensibi)

```java
interface IWorker {
    void eat() throws Exception;

    void work();

    void pay() throws Exception;
}
```

```java
class RobotWorker implements IWorker {

    @Override
    public void eat() throws Exception {
        throw new Exception();
    }

    @Override
    public void pay() throws Exception {
        throw new Exception();
    }

    @Override
    public void work() {
      ...
    }
}
```

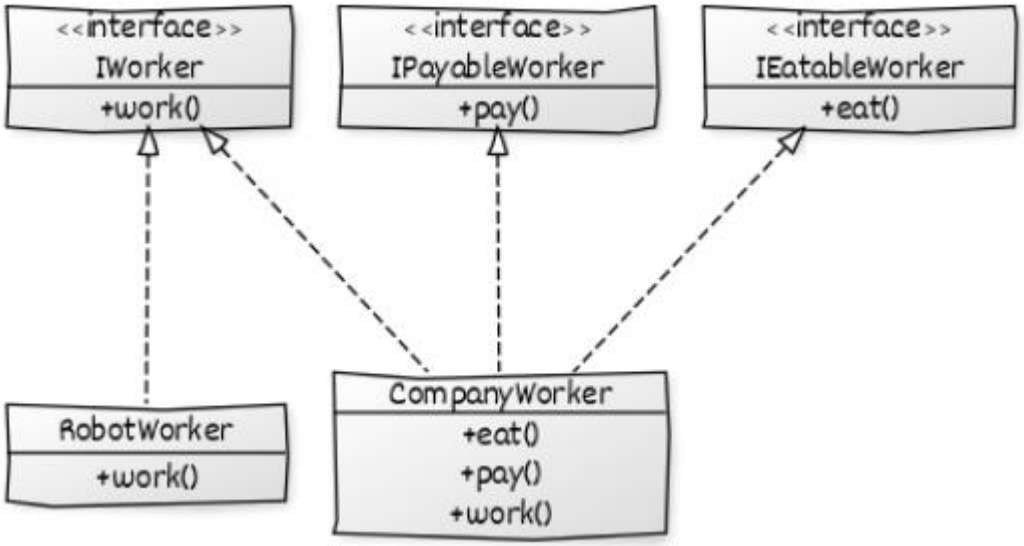
Yandaki kod incelendiğinde, şirket çalışanları `IWorker` arayüzünü uygulamaktadır; yemek yeme, ödeme alma, çalışma gibi davranışları gerçekleştirmektedir.
Fakat daha sonradan bazı işler robotlar tarafından yapılmaya başlandı ya da dış kaynaktan birileri(outsource) de çalışmaya başladı.
Bu durumda bazı davranışlar gerçekleşmeyecektir.
Örneğin robotların **yemek yeme** ya da **ödeme alma** davranışını gerçekleştirememesi gibi ya da dış kaynaktan gelenlere verilmeyen yemek imkanı. Bu gerçekleşmeyen davranışların içlerini ya boş bırakma ya da hata uyarı (throws exception) durumunda kalırız.

I— Interface Segregation Principle (Arayüz Ayrımı Prensibi)

Bu tür durumlarda bu prensip bizlere bu arayüzlerin ayrılmasını ve ihtiyaç halinde olanların kullanılmasını söylemektedir.

Önceki UML diyagramını biraz daha düzenlersek aşağıdaki gibi bir yapı elde edilir.

`work()` , `pay()` , `eat()` davranışları başka arayüzlere aktarıldı ve ihtiyaç halinde olanlar uygulandı.



D— Interface Segregation Principle (Arayüz Ayrımı Prensibi)

```java
interface IWorker {
    void work();
}
```

```java
interface IEatableWorker {
    void eat();
}
```

```java
interface IPayableWorker {
    void pay();
}
```

```java
class Worker implements IWorker, IEatableWorker, IPayableWorker {

    @Override
    public void eat() {
        ...
    }

    @Override
    public void work() {
        ...
    }

    @Override
    public void pay() {
        ...
    }
}
```

```java
class RobotWorker implements IWorker {
    @Override

    public void work() {
     ...
    }
}
```

D— Dependency Inversion Principle (Bağımlılıkların Tersine Çevrilmesi Prensibi)

Yüksek seviye sınıflar, düşük seviye sınıflara bağlı olmamalıdır. Her ikisi de soyutlamalara bağlı olmalıdır.
Soyutlamalar, detaylara bağlı olmamalıdır. Detaylar, soyutlamalara bağlı olmalıdır.

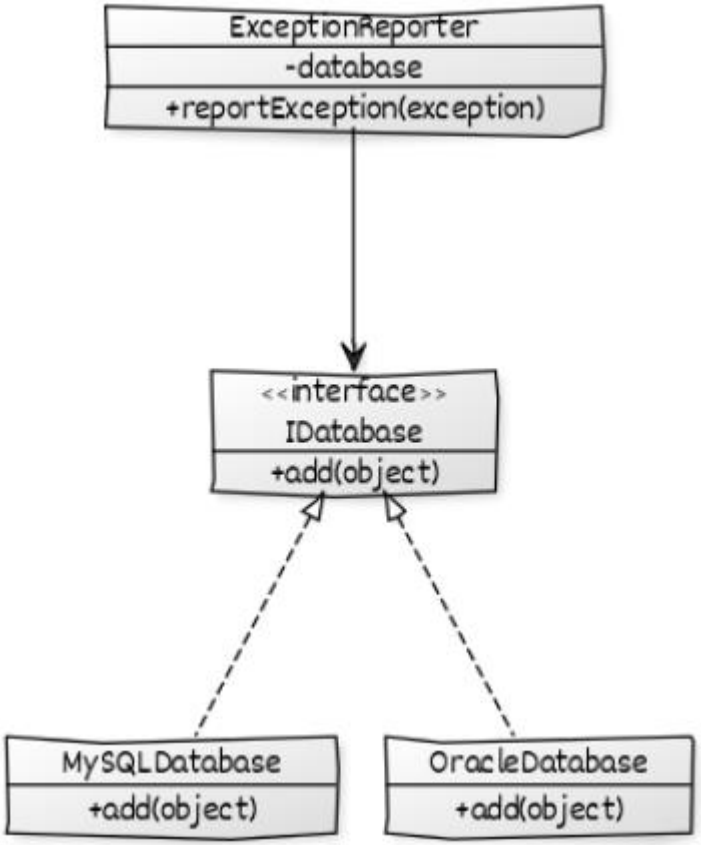
D— Dependency Inversion Principle (Bağımlılıkların Tersine Çevrilmesi Prensibi)

```java
class ExceptionReporter {
    private OracleDatabase oracleDatabase;

    public ExceptionReporter() {
        oracleDatabase = new OracleDatabase();
    }

    public void reportException(Exception exception) {
        oracleDatabase.add(exception);
    }
}

class OracleDatabase {
    public void add(Object object) {
        System.out.println("added :D");
    }
}
```

Yukarıdaki diyagram ve kod incelendiğinde `ExceptionReporter` sınıfının (yüksek seviyeli sınıf), `OracleDatabase` sınıfına (düşük seviyeli sınıf) direkt olarak bağımlı olduğu görülmektedir. İleride veri tabanı olarak Oracle değil de MySQL kullanmak istersek maalesef bu sınıfa müdahale etmek zorunda kalacağız. Bu istenmeyen bir davranıştır. Bunun çözümünü ise buradaki bağımlılıkları soyutlayarak sağlayacağız.

D— Dependency Inversion Principle (Bağımlılıkların Tersine Çevrilmesi Prensibi)

İlk baştaki UML diyagramını biraz daha düzenlersek yandaki gibi bir yapı elde edilir.



D— Dependency Inversion Principle (Bağımlılıkların Tersine Çevrilmesi Prensibi)

```
class ExceptionReporter {
    private IDatabase database;

    public ExceptionReporter(IDatabase database) {
        this.database = database;
    }

    public void reportException(Exception exception) {
        database.add(exception);
    }
}

interface IDatabase {
    void add(Object object);
}

class MySQLDatabase implements IDatabase {
    @Override
    public void add(Object object) {
        ...
    }
}

class OracleDatabase implements IDatabase {
    @Override

    public void add(Object object) {
        ...
    }
}
```

Referanslar

1. https://medium.com/@gokhana/solid-nedir-solid-yaz%C4%B1l%C4%B1m-prensipleri-nelerdir-40fb9450408e

2. SOLID PRENSİPLERİ İLE BAKIM İÇİN YAZILIMI YENİDEN YAPILANDIRMA YÖNTEMİ, Osman Turan, YL tezi.

3. Yazılım Desenleri Türkçe ders notları, Yusuf YILMAZ.

Sorularınız