

innova patika spring bootcamp 4. hafta 2. ödevi. SOLID prensipleri

öğrenci: ozan aydoğan

öğretmen: hamit mızrak

github:

SOLID Nedir

Geliştirme yaparken Java, C# gibi Object Oriented Programming (Nesneye Yönelimli Programlama) mimarisine sahip dilleri tercih eden şirketlerin iş ilanlarında sıkça aranan özellik olan ve mülakatlarda mutlaka “SOLID prensipleri nelerdir?” sorusu karşımıza çıkar.

Yazılımcılar büyük ölçekli projeler geliştirmeye başladığından beri birçok temel sorun ile karşılaşmıştır. Sorunların her birine farklı yaklaşımlar getirerek çözümler ortaya konulmuş ve bunların bir kısmı yazılım camiasından kabul görerek günümüzde aktif olarak kullanılan Design Patterns ve Prensipleri oluşturmuştur



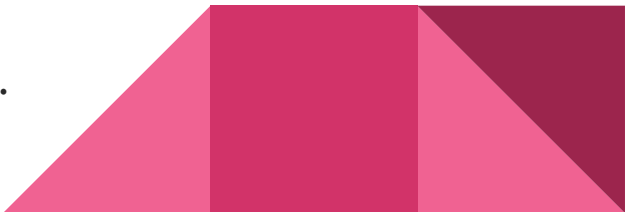
SOLID Nedir

Solid Prensipleri, yazılım geliştirme süreçlerinde karşılaşılan temel sorunlara getirilen ve içerisinde 5 temel prensibi barındıran prensipler bütünüdür.

Peki, nedir bu sıkça karşılaşılan temel sorunlar;



Yazılım geliştirme sürecinde karşılaşılan sorunlar

- **Esnemezlik:** Kullanılan tasarımın geliştirilememesi ve ekleme yapılamaması.
 - **Kırılganlık:** Bir yerde Yapılan değişikliğin başka bir yerde sorun çıkartması.
 - **Sabitlik:** Geliştirilmiş modülün başka yerde tekrar kullanılabilir (reusable) olmaması.
 - **Maliyet:** Geliştirme maliyetinin ve sürecinin giderek artması.
- 

SOLID Prensipleri

SOLID PRENSİPLERİ

1. Single Responsibility Principle
2. Open/Closed Principle
3. Liskov 's Substitution Principle
4. Interface Segregation Principle
5. Dependency Inversion Principle



1. Single Responsibility Principle (SRP)

“Her sınıfın veya metodun tek bir sorumluluğu olmalı”

Prensibin ana hedefini ben şu sözle özdeşleştiriyorum, “**Bir tek şeyi yap ve onu en iyi yap.**” Prensibimizin ana hedefi her sınıfın ve metodun tek bir sorumluluğu olması ve o sorumluluğu yerine getirmesi gerektiğini ve bir sınıf veya metod değişmesi için bir tek sebebi olması gerektiğini söyler.



1. Single Responsibility Principle (SRP)

Projelerde en önemli ama aslında en fazla ihmal edilen konulardan birisi sorumluluk ayrımıdır. Bu konuya dikkat edilmeyen projelere sonradan dâhil olduğumuzda veya kendimiz o an işimizi çözmesi için prensibe uygun olmayan bir şekilde kod yazdığımız zaman, bir süre sonra ortaya her değişikliğin birçok yeri etkilediği, okunması zor ve geliştirme maliyeti fazla olan “**Makarna kod**” diye tabir ettiğimiz bir çöp proje çıkar.



1. Single Responsibility Principle (SRP)

Her defasında kodlamaya başlamadan önce şu iki sorunu kendimize sormamız gerekir;

“Bu metod bu sınıfın içerisinde mi yer almalı?”

“Bu görevi yerine getirmek bu metodun veya sınıfın işimi?”



1. Single Responsibility Principle (SRP) örneği

```
import java.util.Random;

public class BankService {

    public long deposit(long amount, String accountNo) {
        //deposit amount
        return 0;
    }

    public long withdraw(long amount, String accountNo) {
        //withdraw amount
        return 0;
    }
}
```

```
public class LoanService {

    public void getLoanInterestInfo(String loanType) {
        if (loanType.equals("homeLoan")) {
            //do some job
        }
        if (loanType.equals("personalLoan")) {
            //do some job
        }
        if (loanType.equals("car")) {
            //do some job
        }
    }
}
```

1. Single Responsibility Principle (SRP) örneği

```
public class NotificationService {  
    public void sendOTP(String medium) {  
        if (medium.equals("email")) {  
            //write email related logic  
            //use JavaMailSenderAPI  
        }  
        if (medium.equals("mobile")) {  
            //write logic using twilio API  
        }  
    }  
}
```

```
public class PrinterService {  
  
    public void printPassbook() {  
        //update transaction info in passbook  
    }  
}
```



1. Single Responsibility Principle (SRP) örneđi

görsellerde görüldüğü gibi her bir sınıfın kendine ait sorumluluđu bulunmaktadır. NotificationService sınıfının sadece bildirimlerle ilgileniyor. bunun dışında herhangi bir özelliđi bünyesinde barındırmıyor. PrintService print passbook ile alakalı bir özelliđi bünyesinde barındırıyor. BankService sınıfı, paraçekme ve para miktarını göstermeyle alakalı özellikleri var. Loanservice yapısı ise krediyle alakalı özellikleri barındırıyor. ev kredisi, araç kredisiyle alakalı durumları kısaca “kredi” ile alakalı durumları loanservice yapısını kullanarak değerdendirebiliriz. LoanService yapısına, NotificationService yapısında bulunan sendOTP özelliđini eklersek, mantık hatası yaparız çünkü LoanService bildirimlerle ilgilenmiyor. eđer bu özelliđi LoanService yapısında kullanırsak, projenin sürdürülebilirliđini baltamış oluruz.

2. Open / Closed Principle (OCP)

“Sınıflar değişikliğe kapalı ancak gelişime açık olmalıdır.”

Sınıflarımız veya metodlarımızı oluştururken ileride olabilecek yeni istekler ve gelişmeleri de öngörerek tasarlamamız gerekir. Projemizde oluşabilecek yeni istek ve ihtiyaçlar sonucunda yapacağımız geliştirmeler, projemizdeki diğer sistemleri etkilememeli ve herhangi bir değişikliğe sebebiyet vermemelidir.



2. Open / Closed Principle (OCP)

```
public class EmailNotificationService implements NotificationService {  
    public void sendOTP(String medium) {  
        //write logic to integrate with email api  
    }  
  
    public void sendTransactionReport(String medium) {  
        //write logic to integrate with email api  
    }  
}
```

```
public class WhatsAppNotificationService implements NotificationService {  
  
    public void sendOTP(String medium) {  
        //logic to integrate whatsapp api  
    }  
  
    public void sendTransactionReport(String medium) {  
        //logic to integrate whatsapp api  
    }  
}
```

2. Open / Closed Principle (OCP)

```
public class MobileNotificationService implements Notificationservice {  
    public void sendOTP(String medium) {  
        //write the logic to send otp to mobile  
        //twillo api  
    }  
  
    public void sendTransactionReport(String medium) {  
        //write the logic to send otp to mobile  
        //twillo api  
    }  
}
```

```
public interface Notificationservice {  
  
    public void sendOTP(String medium);  
  
    public void sendTransactionReport(String medium);  
}
```

2. Open / Closed Principle (OCP)

Sınıflarımız, NotificationService interface yapısını implement etmektedir. doğal olarak bu interface'de olan özellikleri implement etmektedir. EmailNotificationService yapısı, implement ettiği özellikleri kendi kullanım özelliklerine doldurup, kendisine has bir servis yapısı oluşturur. aynı durumlar diğer servis yapıları içinde geçerlidir. Bu kısımda şuna dikkat etmeliyiz. interface'ye eklediğimiz yeni bir özellik, bu interface'yi implement eden tüm methodlarda yeniden doldurulmalıdır. yani prensibimizdeki “Open” kısmıyla uyumlu hale getirmiş olduk, kodumuz gelişime açık olmuş oldu ve notification classlarımızda varolan özelliklerde herhangi bir değişiklik yapmadan bu gelişimi sağlamış olduk. yani varolan özelliklerini korudular ve yeni birer özellik kazandılar. interface yapımız “Notification”, adındanda anlaşılacağı gibi bildirimlerle ilgileniyor. doğal olarak, bu interface'yi implement edecek olan yapılar notification ile ilgilenmelidir. interface

3. Liskov's Substitution Principle (LSP)

Alt sınıflar miras aldığı üst sınıfın bütün özelliklerini kullanmalı, alt sınıflarda oluşturulan nesneler üst sınıfların nesneleriyle yer değiştirdiklerinde aynı davranışı göstermeli ve herhangi bir kullanılmayan özellik olmamalı.



3. Liskov's Substitution Principle (LSP)

Sağda görüldüğü gibi SocialMedia adında bir abstract classımız olsun, bu classtan türeyen 3 farklı classımız olsun. whatsapp, facebook, instagram. SocialMedia sınıfını extend ettikleri için bu sınıfın özelliklerini kullanabiliyor olmalılardır. eğer abstract classta tanımlı olan bir özellik alt classta kullanılmıyorsa, bu durumda LSP prensibine aykırı bir durum yapmış oluruz

```
public abstract class SocialMedia {  
  
    //@support WhatsApp,Facebook,Instagram  
    public abstract void chatWithFriend();  
  
    //@support Facebook,Instagram  
    public abstract void publishPost(Object post);  
  
    //@support WhatsApp,Facebook,Instagram  
    public abstract void sendPhotosAndVideos();  
  
    //@support WhatsApp,Facebook  
    public abstract void groupVideoCall(String... users);  
}
```

3. Liskov's Substitution Principle (LSP)

```
public class Facebook extends SocialMedia {  
  
    public void chatWithFriend() {  
  
    }  
  
    public void publishPost(Object post) {  
  
    }  
  
    public void sendPhotosAndVideos() {  
  
    }  
  
    public void groupVideoCall(String... users) {  
  
    }  
}
```

```
public class Instagram extends SocialMedia {  
  
    public void chatWithFriend() {  
  
    }  
  
    public void publishPost(Object post) {  
  
    }  
  
    public void sendPhotosAndVideos() {  
  
    }  
  
    public void groupVideoCall(String... users) {  
        //not applicable  
    }  
}
```

```
public class Whatsapp extends SocialMedia {  
    public void chatWithFriend() {  
  
    }  
  
    public void publishPost(Object post) {  
        //Not applicable  
    }  
  
    public void sendPhotosAndVideos() {  
  
    }  
  
    public void groupVideoCall(String... users) {  
    }  
}
```

3. Liskov's Substitution Principle (LSP)

Yukarda belirtilen classlarda, whatsapp sınıfı socialmedia sınıfını extends etse bile publishpost özelliği desteklemiyor. doğal olarak bu özelliğin bu sınıfta bulunması uygun olmaz. Bu durum LSP'ye aykırıdır. aynı durum instagram sınıfında bulunan groupvideocall özelliğinde de vardır. bu özelliği base classtan alsa bile, instagramda groupvideocall özelliği bulunmamaktadır. Şimdi bu durumlara yapılabilecek en güzel çözümü inceleyelim,



3. Liskov's Substitution Principle (LSP)

tanımladığımız interfacerler
ve abstract classlar,

```
public abstract class SocialMedia {  
  
    public void chatWithFriend() {  
  
    }  
  
    public void sendPhotosAndVideos() {  
  
    }  
  
}
```

```
public interface PostMediaManager {  
  
    public void publishPost(Object post);  
  
}
```

```
public interface SocialVideoCallManager {  
    public void groupVideoCall(String... users);  
}
```



3. Liskov's Substitution Principle (LSP)

tanımladığımız classlar

```
public class Instagram extends SocialMedia implements PostMediaManager {  
    public void publishPost(Object post) {  
    }  
  
    public void chatWithFriend() {  
    }  
  
    public void sendPhotosAndVideos() {  
    }  
}
```

```
public class WhatsApp extends SocialMedia implements SocialVideoCallManager {  
    public void chatWithFriend() {  
    }  
  
    public void sendPhotosAndVideos() {  
    }  
  
    public void groupVideoCall(String... users) {  
    }  
}
```

3. Liskov's Substitution Principle (LSP)

tanımladığımız socialmedia abstract yapısında chatWithFriend() ve sendPhotosAndVideos() özellikleri tanımlıdır. Bu sınıfı extends eden sınıflar, bu özellikleri kullanmalıdır. instagram, whatsapp ve facebook bu sınıfı extends etmektedir. doğal olarak bu özellikleri kullanmaktadır. instagramda groupvideocall özelliği bulunmamaktadır. doğal olarak SocialVideoCallManager interface yapısını implement edemez, postmediamanager interface'inde publishpost özelliğini kullanabilir. çünkü instagramda publishpost özelliği bulunmaktadır. whatsapp ise publishpost özelliği olmadığı için postmediamanager yapısını implement etmez fakat SocialVideoCallManager içindeki özellikleri barındırdığı için, bu interface'i implement eder. böylece LSP prensibine sadık kalmış oluruz.

4. Interface Segregation Principle

Bir ara yüz bir sınıfa implemente edildiği zaman, ara yüz'ün barındırdığı metotları barındırmak veya oluşturmak zorundadır. Zaten bu durumun aksi olduğundan hata alırız.

İşte bu durumda prensibimiz devreye girer ve derki “eğer class içerisinde gerçekten ihtiyaç duyulmayan ve kullanılmayan metotlar ara yüz aracılığı ile implemente edilmiş ise bu kodlar dummy kod olur, bu yüzden ara yüzler ayrılmalı ve classlar açısından işlevsel olmayan metotlar barındırması engellenmelidir.”



4. Interface Segregation Principle

```
public interface IBaseApi {  
    void Put(int id);  
    void Post(int id);  
    void Get(int id);  
    void Delete(int id);  
}
```

```
public class Video implements IBaseApi{  
    public void Put(int id) {  
    }  
    public void Post(int id) {  
    }  
    public void Get(int id) {  
    }  
    public void Delete(int id) {  
    }  
}
```

```
public class News implements IBaseApi{  
    public void Put(int id) {  
    }  
    public void Post(int id) {  
    }  
    public void Get(int id) {  
    }  
    public void Delete(int id) {  
    }  
}
```



4. Interface Segregation Principle

BaseAPI ara yüzümüz ve bunun içerisinde get, put, post ve delete metodlarımız olsun. Ardından bunu service katmanlarındaki news, video gibi katmanlara implemente edelim.

Şimdilik bir sıkıntı yok peki sadece Get yaptığımız bir servisimiz olursa örneğin bildirimleri çektiğimiz Notification servisimiz olduğu durumda buna BaseApi implemente edilir ise gereksiz bir şekilde Hiç kullanılmayacak olan put, pot ve delete metodları da implemente edilecek. Aşağıdaki kod bloğunda görüldüğü gibi.



4. Interface Segregation Principle

```
public class Notifications implements IBaseApi{  
    public void Put(int id) {  
    }  
    public void Post(int id) {  
    }  
    public void Get(int id) {  
    }  
    public void Delete(int id) {  
    }  
}
```

4. Interface Segregation Principle

Peki, bu durumda doğru yaklaşım nasıl olmalıdır? Doğru yaklaşım Sadece içerisinde Get metodunu barındıran bir ara yüz oluşturulması ve Notification servisi bundan kalıtılması olmalıdır. Aynı zamanda IBaseApi içerisinde Get metodu çıkartılarak yeni oluşturulan ara yüz IBaseApi tarafından kalıtılmalıdır.



4. Interface Segregation Principle

```
public interface IBaseApi extends IGet {  
    void Put(int id);  
    void Post(int id);  
    void Delete(int id);  
}
```

```
public class Notifications implements IGet{  
  
    public void Get(int id) {  
  
    }  
}
```



4. Interface Segregation Principle

Böylelikle IBaseApi den kalıtılan News ve Video servicelerinde bir deęişiklik yapmak zorunda kalmadıđımız gibi, aynı zamanda Notification servicesinde IGet arayüzünden kalıtarak gereksiz metodlardan kurtulmuş oluruz aşıađıda olduđu gibi.




5. Dependency Inversion Principle

“Katmanlı mimarilerde üst seviye modüller alt seviyedeki modüllere doğrudan bağımlı olmamalıdır.” bağımlılıkların tersine çevrilmesi şeklindedir.

“üst seviyeli katmanlar kesinlikle alt seviyedeki katmanlara bağlı olmamalı, bağımlılıklar sadece abstract (soyut) kavramlara olmalıdır” şeklinde ifade edebiliriz.

Burada amaç üst seviyedeki modüllerin alt seviyelere bağımlı olmasından dolayı çıkabilecek sorunları ortadan kaldırmaktır. Yani alt seviyede yapılan herhangi bir değişikliğin üst seviyede kod değişikliğine veya onun bağımlılıklarının etkilenmesine engel olmaktır amaç.



5. Dependency Inversion Principle

Daha kaba bir anlatım ile üst seviyedeki modül alt seviyedeki işin nasıl yürüdüğünü bilmemeli ve ilgilenmemelidir. Ben üst seviyedeki modül olarak diyorum ki; alt taraftan x verisinin nasıl geldiği ile ilgilenmem, bana gelen x verisi ile ilgili işlemimi yapar, gerekli yerlere gönderirim. Ama x verisi bir veri tabanından mı geldi, bir API den mi geldi yoksa bir text dosyasından okunarak mı geldi burası beni ilgilendirmez. Zaten bu gibi isteklerinde beni etkilememesi gerekmektedir.



5. Dependency Inversion Principle

```
public interface BankCard {  
  
    public void doTransaction(long amount);  
}
```

```
public class CreditCard implements BankCard{  
  
    public void doTransaction(long amount){  
        System.out.println("payment using Credit card");  
    }  
}
```

```
public class ShoppingMall {  
  
    private BankCard bankCard;  
  
    public ShoppingMall(BankCard bankCard) { this.bankCard = bankCard; }  
  
    public void doPurchaseSomething(long amount) { bankCard.doTransaction(amount); }  
  
    public static void main(String[] args) {  
        // DebitCard debitCard=new DebitCard();  
        // CreditCard creditCard=new CreditCard();  
  
        BankCard bankCard=new CreditCard();  
        ShoppingMall shoppingMall=new ShoppingMall(bankCard);  
        shoppingMall.doPurchaseSomething( amount: 5000);  
    }  
}
```

```
public class DebitCard implements BankCard{  
  
    public void doTransaction(long amount){  
        System.out.println("payment using Debit card");  
    }  
}
```


5. Dependency Inversion Principle

Bir alışveriş merkezinde, bir ürün alırken ödeme yaptığımızı farkedelim. Bu ödemeyi credit card ya da debir card ile yapabileceğimizi farzedelim, eğer main classta bulunan ShoppingMall özelliğine parametre olarak, herhangi bir kart türünü direk verirsek (ShoppingMall(BankCard bankCard) yerine ShoppingMall(CreditCard bankCard)) bu özellik belirli bir parametreye bağlı kalmış olur. eğer biz method içerisinde CreditCard bankCard şeklinde tanım yaparsak ve ödemek için debitkart kullanmak istersek, buna izin verilmez. çünkü shopmall sadece creditcard'a izin verir. bu durumu önlemek için ödeme yöntemlerini bir BankCard interface yapısına aktarıp, shopmall içerisinde parametre olarak BankCard tipindeki değerleri alırsak, BankCard'ı implement eden herhangi bir ödeme yöntemini kullanabilmiş oluruz.

Ozan Aydođan

Kocaeli Üniversitesi Bilgisayar Mühendisliđi

İnnova-Patika spring bootcamp öğrencisi

