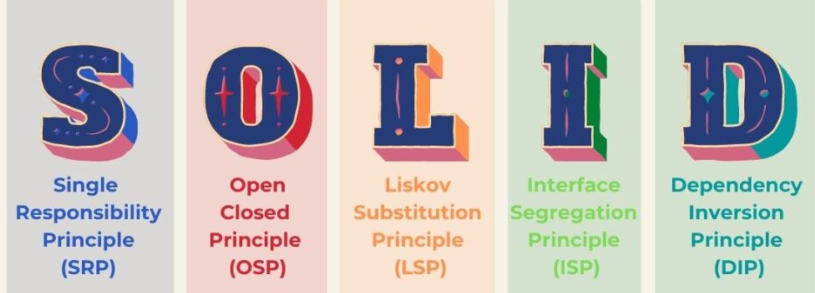
# SOLID PRENSİPLERİ



SOLID yazılım prensipleri; geliştirilen yazılımın **esnek**, **yeniden kullanılabilir**, **sürdürülebilir** ve **anlaşılır** olmasını sağlayan, **kod tekrarını önleyen** ve Robert C. Martin tarafından öne sürülen prensipler bütünüdür. Bu prensiplerin amacı;

* Geliştirdiğimiz yazılımın gelecekte gereksinimlere kolayca adapte olması,
* Yeni özellikleri kodda bir değişikliğe gerek kalmadan kolayca ekleyebileceğimiz
* 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.

## 1-S : Single Responsibility Principle (SRP)( Tek Sorumluluk İlkesi)

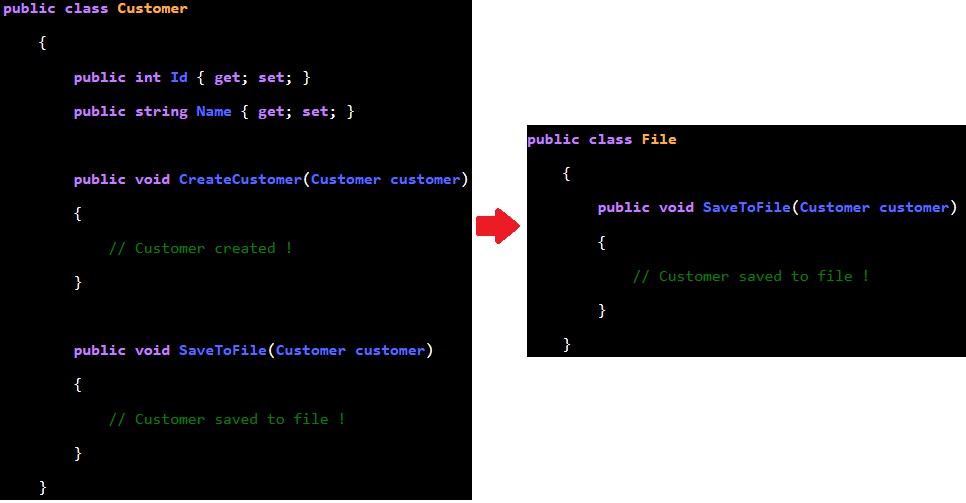
***Sorumluluğun azaltılması demek değişime daha kolay adapte olmak demektir.***

## 

***Single Responsibility Principle’a göre, her method ve class’ın tek bir görevi ve sorumluluğu vardır.***

*Yazılan programda güncelleştirmeler gerektiğinde kolayca entegre edilebilen bir yapı oluşturmak amaçlanmıştır.*

Aşağıdaki “Customer” class yapısını incelersek bu class yapısı içerisinde Customer oluşturan bir fonksiyon var. Ayrıca oluşturulan kullanıcıyı dosyaya kaydetmek için oluşturulmuş ayrı bir fonksiyon daha var. Bu class yapısı 2 farklı sorumluluk alıyor, biri Customer oluştururken, diğeri ise dosyaya yazma işlemi yapıyor. Customer sınıfı, dosyaya kaydetme sorumluluğunu üstlenmemelidir.



SRP’ye göre bir class sadece bir sorumluluk almalı. Bu yüzden SaveToFile fonksiyonunu File isminde bir class oluşturulur. Class oraya aktarılır. Böylelikle 2 ayrı işi birbirinden ayrılmış olur.



Sorumlulukları her bir sınıfa etkili bir şekilde ayırdık ve böylece bir yerdeki kod değiştirme şansını ve koddaki herhangi bir yerin diğerini bozma şansını azalttık.

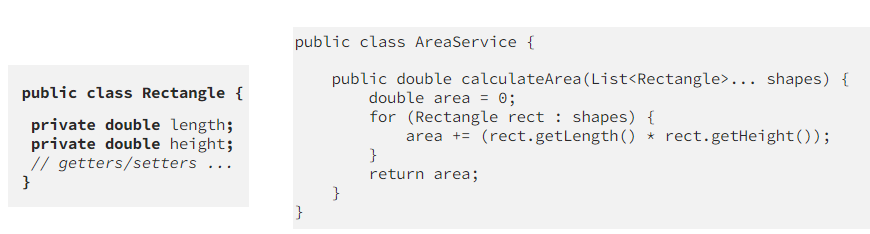
## ****2) O : Open Closed Principle (OSP)(**** ****Açık Kapalı İlke)****

## *Bir sınıf ya da fonksiyon halihazırda var olan özellikleri korumalı ve değişikliğe izin vermemelidir. Yani davranışını değiştirmiyor olmalı ve yeni özellikler kazanabiliyor olmalıdır.*

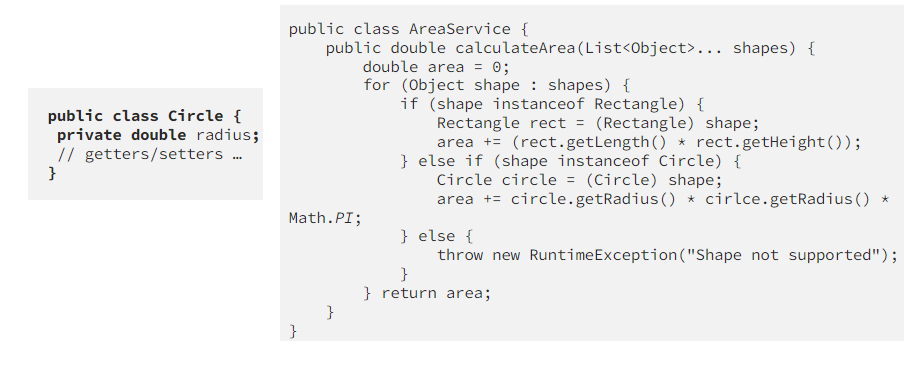
**Open**sınıf için yeni davranışlar eklenebilmesini sağlar. Gereksinimler değiştiğinde, yeni gereksinimlerin karşılanabilmesi için bir sınıfa yeni veya farklı davranışlar eklenebilir olmasıdır.  
**Closed**bir sınıf temel özelliklerinin değişimi ise mümkün olmamalıdır.

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.

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

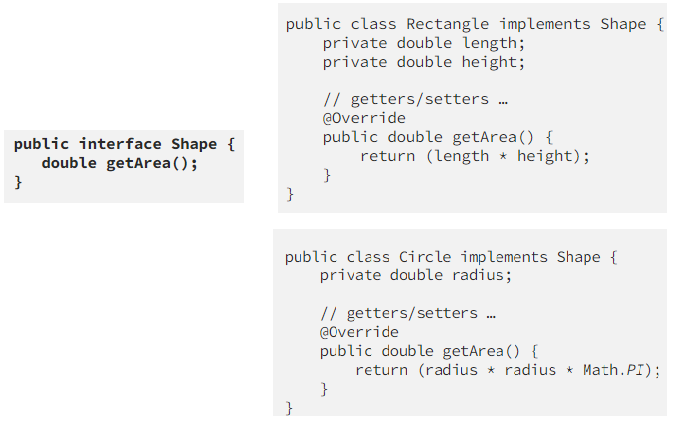


Area servisimiz dikdörtgen hesabını yapacak seviyede, ancak bizim ihtiyaçlarımız daire hesabının da yapılmasını da gerektirmeye başladı diyelim. Daire alan hesabı yapabilmek için AreaService’i içindeki calculateArea methodunda değişiklik yapmamız gerekir.

**

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.

Bunu çözmek için bir Shape interface’imiz olsa ve her bir şekil için hesaplanmış area’yı dönse nasıl olur ? Bunu yapmakla object tipinden de kendi bildiğimiz bir nesneye geçiş sağlamış olacağız.

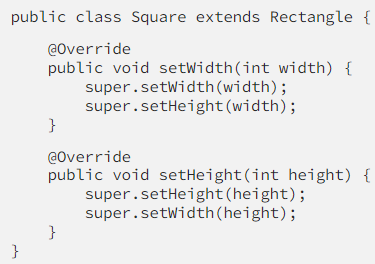
**

## 3)[Liskov substitution principle](https://gokhana.medium.com/liskov-substitution-prensibi-nedir-kod-%C3%B6rne%C4%9Fiyle-soli%CC%87d-3cfc1cd63c1a)

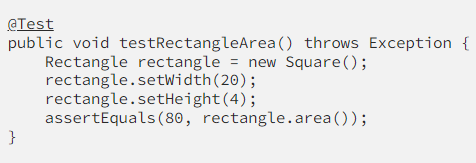
Kodlarımızda herhangi bir değişiklik yapmaya gerek duymadan alt sınıfları, türedikleri(üst) sınıfların yerine kullanabilmeliyiz.

Türeyen sınıf yani alt sınıflar ana(üst) sınıfın tüm özelliklerini ve metotlarını aynı işlevi gösterecek şekilde kullanabilme ve kendine ait yeni özellikler barındırabilmelidir.

LSP’ye uymayan yapı örneği;



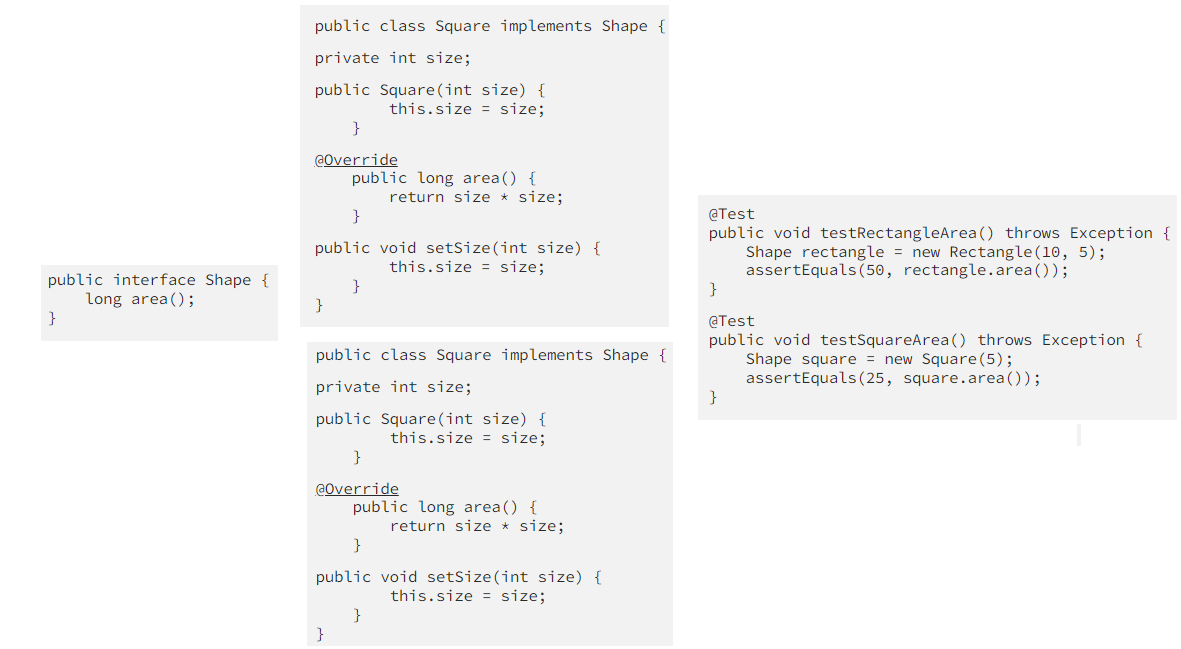
Matematiksel olarak bir kareyi de bir dikdörtgen olarak kabul edebiliriz.



Kareyi de beklendiği yerde dikdörtgen olarak kullanılabilir hale getirmiş olduk. Ancak böyle yaparak dikdörtgen davranışındaki beklentiyi bozuyoruz. Çünkü karenin sadece tek bir kenar bilgisi yeterlidir yada uzunluk ve en bilgisi aynı olmak durumundadır.

Matematiksel olarak karenin dikdörtgenden türediğini varsayabiliriz. Ama davranışsal olarak Kare Dikdörtgenin yerine geçmez, bu hiyerarşi Liskov prensibini (LSP) ihlal eder.

Open/Closed prensibine benzer ve open/closed prensibine uymak liskov prensibinin uygulanmasını kolaylaştırmaktadır.



Kare bir şekildir o halde Square adlı bir sınıf yaratarak Shape’den implement edebiliriz.

Dikdörtgen Kare ile farklı davranışlar gösterebiliyor o halde onu ayrı bir şekil olarak Rectangle adlı bir sınıf yaratıp Shape’den implement edebiliriz.

Artık Kare ve Dikdörtgen kendi davranışlarına sahip oldu. Ve her biri ayrı şekil olarak kabul ediliyor. Böylece alan hesaplama her bir şekle özgü matematiksel bir işlem içerebiliyor.

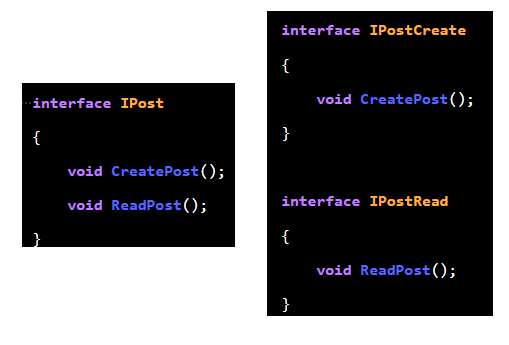
### ****4)Interface Segregation Principle(ISP)(**** ****Arayüz Ayırma Prensibi)****

### *Sorumlulukların hepsini tek bir arayüze toplamak yerine daha özelleştirilmiş birden fazla arayüz oluşturmalıyız.*

### Her farklı sorumluluğun kendine özgü bir arayüzü olması gerekmektedir. Böylece interface’i kullanan kişide sadece ihtiyacı olanlarla ilgilenmiş olur.

### İnterface birden fazla iş yapmaktadır. Bu arayüzden türetilen sınıflar tüm metodları kullanmak zorunda kalacaktır. Bunun yerine bu arayüzler daha küçük iş birimlerine ayrılmalıdır.

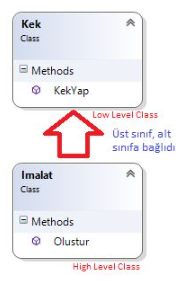
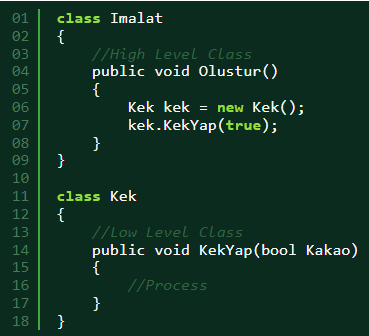
### Küçük parçalara ayrılmış interface’ler sınıflara daha kolay eklenirler. Bu sayede bu arayüzlerden türetilen sınıflar kullanmadıkları metodları almamış olurlar.



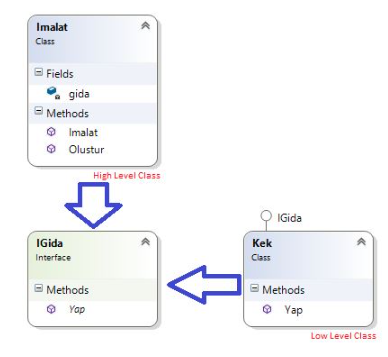
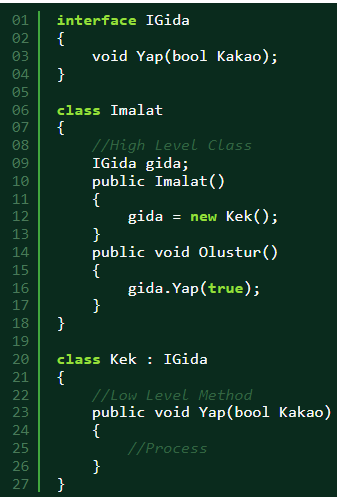
# 5) ****Dependency Inversion Principle****

***Sınıflar arası bağımlılıkların minimal seviyeye indirgenmesi ve bağımlılıkların sınıflar ile değil arayüzler*** *(****interface****)* ***ile kurulması gerektiğine dayanır.***

Sistem bu şekilde tasarlanmazsa yüksek seviyeli bileşenler, düşük seviyeli bileşenlere bağımlı kalacak ve düşük seviyeli bir bileşen içerisinde yapılacak olan değişikliğin zincirleme olarak bağımlı olan tüm yüksek seviye bileşenleri de değişikliğe zorlayacaktır. Dependency Inversion tam olarak da bu bağımlılığın tersine çevrilmesini amaçlamaktadır.



Yukarıdaki “Imalat” sınıfı içerisindeki “Olustur” metodu, “Kek” sınıfı içerisindeki “KekYap” isimli metoda bağımlıdır.



 Alt seviye sınıfı Interface sayesinde soyutlaştırarak, üst seviye sınıfta alt seviye sınıfına dair olan bağımlılığı tersine çevrilmiştir. Yani alt seviye sınıf, Interface’e bağımlı bir hale gelmiştir. İlgili Interface’de üst seviye sınıf tarafından refere edildiğinden dolaylı yoldan bağımlılık alt seviyeden üst seviyeye doğru olmuştur .