



# SOLID

Software Development is not a Jenga game

## SOLID Principles

**Mehmet Aydın Ünlü**

[aydinunlu85@gmail.com](mailto:aydinunlu85@gmail.com)

<http://www.aydinunlu.blogspot.com>

Kaynak : <http://www.oodeesign.com>

# İçindekiler

1. Solid Principles Nedir ?
  - a. Single Responsibility Principle
  - b. Open/Closed Principle
  - c. Liskov 's Substitution Principle
  - d. Interface Segregation Principle
  - e. Dependency Inversion Principle

# 1. SOLID PRINCIPLES

**Design Principles kavramı**, kötü tasarımdan uzak durmak için belirlenmiş belli kurallardan oluşan bir kurallar zinciridir.

**Principles of Class Design** kavramı ise sınıflar üzerinde ki modellemede uymamız gereken prensipleri belirlemiştir. Bunun dışında ayrıca paketlerin nasıl tasarlanması gerektiğini belirten prensipler de vardır. Biz bu yazı dizimizde sınıflar için belirlenmiş prensipleri tanıyacağız.

Sınıflar için uyulması gereken bu prensipler, kötü tasarıma neden olabilecek 3 ana unsur ortadan kaldırmak amacıyla geliştirilmiştir. Peki nedir bu unsurlar ? Robert Martin bunları şöyle sıralamıştır;

**Rigidity (Esnemezlik)** : Kullanılan tasarımın esnek olmadığını gösterir. Yani kullanılan tasarımın geliştirmeye ve plug-in mimarisine uygun olmadığını gösterir.

**Fragility (Kırılganlık)** : Sistemin bir yerinde yaptığınız bir değişikliğin, sistemin bir başka yerinde sorun çıkarmasıdır.

**Immobility (Sabitlik)** : Geliştirdiğiniz bir modülün tekrar kullanılabilir olmadığını gösterir.

Bu 3 ana unsurun aslında ortak bir paydada buluştuğunu ve buna çözüm bulmak için prensipler geliştirildiğini söyleyebiliriz. Şöyle ki; bu durumların her birinin ortaya çıktığı nokta bağımlılık seviyesi yüksek sınıflarda görülür. Sınıf tasarımları eğer birbirlerine sıkı sıkıya bağlı ise, bu tasarım ne esnek, ne sağlam ne de taşınabilir olur. Buradan şunu söyleyebiliriz ki tüm prensiplerin çözüm ararken yapmaya çalıştığı şey, mümkün olduğu kadar sınıfların birbirlerine olan bağımlılıklarını azaltmaktır. Bunların nasıl yapılacağını yazı dizimizin sonraki bölümlerinde tek tek göreceğiz zaten. Fakat bundan önce şimdi derseniz bu sorunları ortadan kaldırmak için uymamız gereken prensiplere çok kısa bir şekilde bakalım.

**Single Responsibility Principle** : Her modülün bir tek sorumluluğa sahip olmasını ve ilerde olası bir değişikliğin de tek bir nedene dayandırılması gerektiğini belirtir.

**Open/Closed Principle** : Genişlemelere açık, modifikasyonlara kapalı bir tasarım kullanılması gerektiğini belirtir.

**Liskov 's Substitution Principle** : Türeyen sınıfın üyeleri, temel sınıfın üyeleri ile tamamen yer değiştirebilir olmalıdır.

**Interface Segregation Principle** : Interface' lerin mümkün olduğu kadar birbirlerinden ayrıştırılması gerektiğini belirtir.

**Dependency Inversion Principle** : Yüksek seviye sınıfların, düşük seviye sınıflara direkt olarak bağımlı olmaması gerektiğini belirtir. Bunu da araya bir abstract sınıf veya Interface koyarak yaparız.

## A- SINGLE RESPONSIBILITY PRINCIPLE



### SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

**Bir class 'ın sadece ve sadece bir tek sorumluluğa sahip olması gerektiğini belirtir. Buradan yola çıkarak şunu diyebiliriz ki; bir class 'ın ileride herhangi bir değişikliğe uğraması için öne sürülebilecek sadece bir tane sebep olmalıdır.**

Diğer bir deyişle, bir class 'a verilebilecek görev tekil olmalıdır. Örnek vermek gerekirse; bir sınıf, temel kayıt işlemlerinden (CRUD) sorumluyken aynı zamanda bir raporlama işlemi de yerine getirecek işlevleri yönetmemelidir. Veya rapor almaktan sorumlu bir sınıf aynı zamanda bu raporları ilgili kişilere mail atabilecek yeteneğe sahip olmamalıdır. Kısaca her sınıf, bir tek amaca hizmet edecek şekilde geliştirilmelidir.

Bunun dışında tek bir amaca hizmet edecek olan sınıfların da, kendi içinde ileride çıkabilecek değişiklik sebepleri de tekil olmalıdır.

Dilerseniz bu söylediklerimi daha rahat anlayabilmek için aşağıdaki örneği inceleyelim. Elimizde mesaj göndermekten sorumlu, **IMessageManager** interface 'ini implemente etmiş **MessageManager** adında bir sınıf var.

```
Public Interface IMessageManager
{
    void Send(string message, string number);
}
Public Class MessageManager : IMessageManager
{
    void Send(string message, string number)
    {
        // Send...
    }
}
```

Koda ilk baktığımızda herşey son derece düzgün görünüyor. **Send** adında iki parametre alan bir method var. Bu method aldığı string tipindeki mesajı, ikinci parametrede aldığı numaraya gönderiyor. Fakat sorun tam olarak bu noktada başlıyor. Siz programı bu şekilde piyasaya sürdünüz diyelim. 3 ay sonra MMS diye bir teknoloji çıktığı zaman, sizin string tipinde parametre alan methodunuz bu ihtiyacı karşılamayacak. Tamam programı yayınladığınız zaman böyle bir teknoloji yoktu. Zaten beklenti asla MMS diye bir teknolojiyi önceden tahmin edip ona göre bir sınıf tasarlamamız değildir. **Fakat beklenen şey; ileride değişiklik olabileceğini göz önüne alarak buna uygun esnek bir yapı tasarlamamızdır.** Tabi olası bu yeniliklere mümkün olduğu kadar modüler çözümler sunmak gerekiyor. Şöyleki; bu şekilde yazılan bir kodun yeni bir MMS teknolojisine sunacağı çözüm, ancak MMS tipinde parametre alan ikinci bir Send methodu yazmaktan geçer. Diğer bir deyişle Send methodunu overload etmekten geçer. Bu da benim mesaj göndermekten sorumlu olan MessageManager isimli sınıfımın, mesaj göndermek ile tamamen alakasız olarak, bir mesaj tipi yüzünden değişime uğraması demektir. Bu da SIR' a tamamen aykırı bir durumdur. Benzer şekilde 5 ay sonra da sesli mesaj teknolojisi çıktığı zaman aynı süreci maalesef tekrar yaşamak gerekecektir...

Çözüm son derece basittir. Mesajımızı string olarak göndermek yerine, **IMessage** gibi bir Interface 'ten implemente edilmiş bir sınıf tipinde göndermek işimizi görecektir.

Methodumuz artık IMessage referansına sahip bir parametre almaktadır. Yani aşağıdaki kod bloğunda görüldüğü gibi.

```

Public Interface IMessage
{
}

Public Class SMS : IMessage
{
    public string Message;
}

Public Class Picture
{
    public String Name;
    public int Size;
}

Public Class MMS : IMessage
{
    public Picture Message;
}

Public Interface IMessageManager
{
    void Send(IMessage message, string number);
}

Public Class MessageManager : IMessageManager
{
    void Send(IMessage message, string number)
    {
        // Send...
    }
}

```

Artık elimizde bir IMessage Interface' i var ve bunu implemente etmiş, SMS ve MMS sınıfları vardır. SMS' in sadece string bir mesajı vardır, fakat MMS' in Picture tipinde çok farklı bir mesajı vardır. Mesaj tipleri farklı dahi olsa biz bu kodu artık şu şekilde kullanabiliriz.

```
SMS sms = new SMS();
sms.Message = "mesaj";
MessageManager manager = new MessageManager();
manager.Send(sms, "123");
//-----
Picture pic = new Picture();
pic.Name = "resim";
pic.Size = 130;

MMS mms = new MMS();
mms.Message = pic;
manager.Send(mms, "123");
```

Görüldüğü gibi ben MessageManager sınıfında en ufak bir değişiklik bile yapmadan MMS gibi farklı bir teknolojiyi desteklemiş oldum. Bunu da sadece yeni sınıflar ekleyerek yaptım. İşte modüler tasarım dediğimiz şey özünde böyle bir mimariye sahip olmalıdır.

SIR, modüler programlamanın temelinde yatan prensiplerden belkide en önemlisidir. Bu şekilde geliştirilen sınıfların birbirlerine olan bağımlılıkları son derece düşüktür. Bu da modüler programlamanın en önemli gereksinimidir. Zaten ileride göreceğimiz diğer prensiplerin de hedeflediği en önemli amaç, sınıfları mümkün olduğu kadar birbirinden soyutlamak ve bağımsız tutmaktır.

Mehmet Aydın Ünlü

[aydinunlu85@gmail.com](mailto:aydinunlu85@gmail.com)

<http://www.aydinunlu.blogspot.com>

## B- OPEN / CLOSED PRINCIPLE

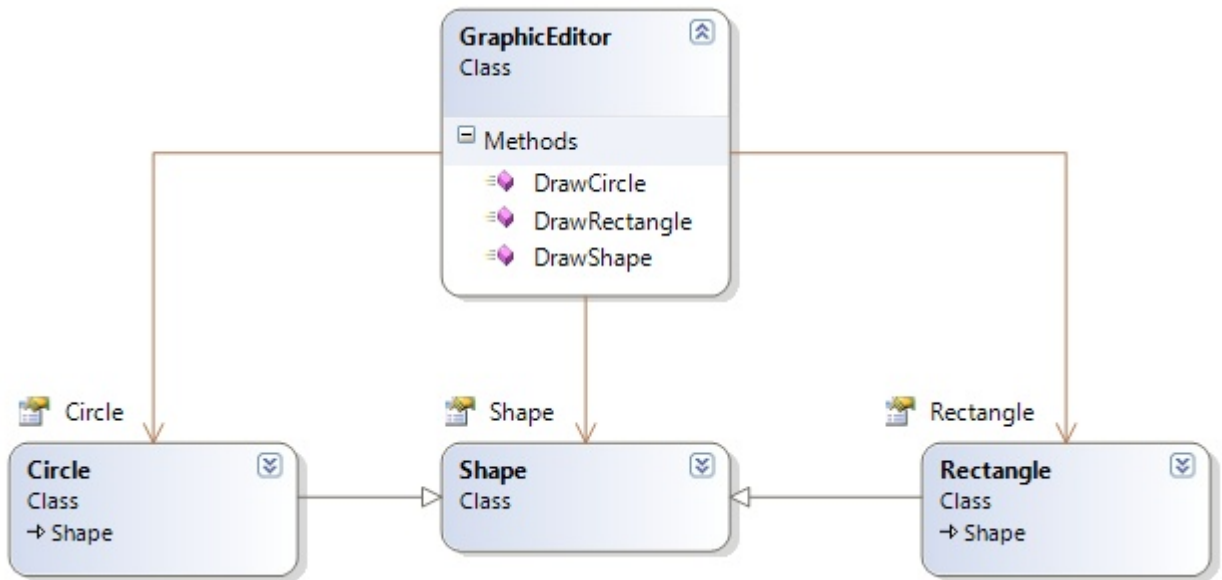


# OPEN CLOSED PRINCIPLE

Open Chest Surgery Is Not Needed When Putting On A Coat

Bu prensibin ana fikri; geliştirilecek olan uygulamanın **genişlemelere açık fakat modifikasyonlara kapalı olması gerektiğidir**. Diğer bir deyişle var olan uygulama üzerine sürekli yeni modüller ve işlevler ekleyebilmelisiniz. **Fakat bunu yaparken var olan kodlar üzerinde asla bir değişiklik yapmamalısınız.**

Bu prensibe olan ihtiyaç, genelde bir yönetici sınıfın, yönettiği diğer sınıflar ile iletişim kurduğu durumlarda ortaya çıkıyor. Şöyle ki; bizim elimizde **GraphicEditor** adında bir yönetici sınıf olsun. Bu sınıf ekrana çeşitli şekillerin çizilmesinden sorumlu olsun ve yapısı aşağıdaki şekildeki gibi olsun.



```

public class GraphicEditor
{
    public void DrawShape(Shape s)
    {
        if (s._type == 1)
        {
            DrawRectangle((Rectangle)s);
        }
        else if (s._type == 2)
        {
            DrawCircle((Circle)s);
        }
    }

    public void DrawRectangle(Rectangle s)
    {
        Console.WriteLine("Rectangle");
    }

    public void DrawCircle(Circle s)
    {
        Console.WriteLine("Circle");
    }
}
  
```

Şekilden ve kodlardan da görüldüğü gibi her bir şekil **direkt** olarak yönetici sınıf olan **GraphicEditor** sınıfına bağlı. İşte bu durum Robert Martin' in belirlediği kötü tasarım ilkelerinden her üçüne de aykırıdır.

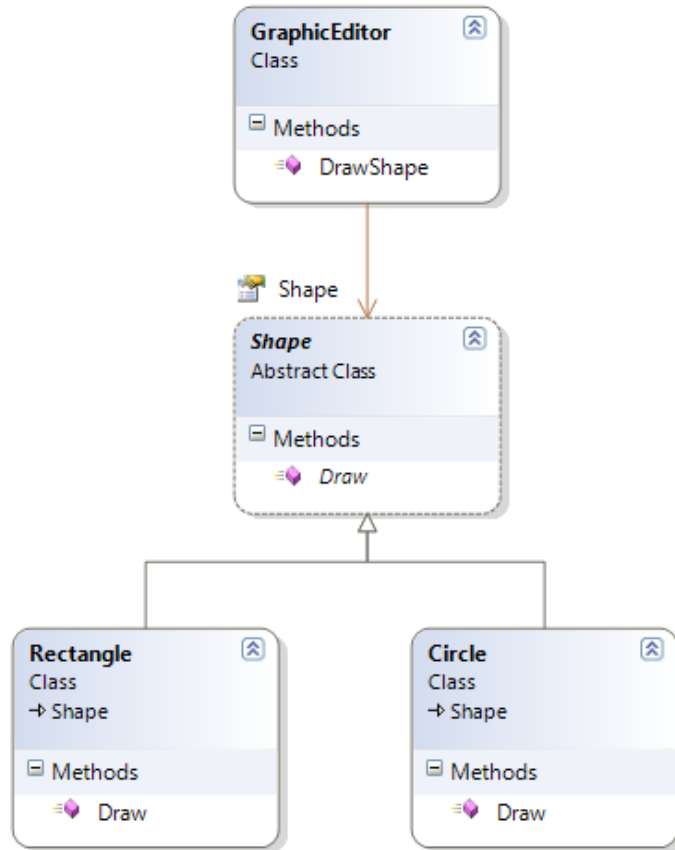
Öncelikle bu kadar birbirine bağlı sınıflardan esnek davranmalarını beklemek pek mümkün değil. Bu durum da **Rigidity** ilkesine tamamen aykırıdır. If koşulu ile hangi şeklin çizileceğinin belirlenmesine değinmiyorum bile.

Bunun dışında sisteme eklemeye çalışacağım herhangi bir yeni şekil, sistemin bir çok yerinde değişikliğe neden olacaktır. Kompleks yapılarda iş akışı hataları ve exceptionların çıkması çok muhtemel bir durumdur. Bu durum da **Fragility** ilkesine aykırıdır.

Son olarak bu kadar birbirine bağımlı bir mimaride siz kalkıp GraphicEditor sınıfını **Shape**, **Circle** ve **Rectangle** sınıfları olmadan tek başına bir yere taşıyamamazsınız. Çünkü GraphicEditor sınıfının bu sınıflara doğrudan bir bağımlılığı vardır. Bu durumda **Immobility** ilkesine ters düşer.

Peki çözüm nedir diye sorarsanız, devam edelim...

Böyle bir durumdan kurtulmak için tek yapmamız gereken şey; şekilleri ortak bir **abstract** sınıftan türetip, bu ortak sınıf içine yazacağımız **abstract bir method** ile Draw işlemini her sınıfın kendine has bir şekilde ele almasını sağlamamız okadar. Bunu da tabiki Draw methodunu her sınıf kendi içinde **override** edecek şekilde geliştirerek yapabiliriz. Doğru tasarımın diagramı ve kodları aşağıdaki gibidir.



```

public abstract class Shape
{
    public abstract void Draw();
}

public class Rectangle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Rectangle");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Circle");
    }
}

public class GraphicEditor
{
    public void DrawShape(Shape s)
    {
        s.Draw();
    }
}

```

Bu yapıyı bir örnekte kullanmaya başladığımız zaman, polymorphism 'in en güzel halini aşağıdaki gibi görebiliyoruz.

```

class Program
{
    static void Main(string[] args)
    {
        GraphicEditor editor = new GraphicEditor();

        Rectangle r = new Rectangle();
        editor.DrawShape(r);

        Circle c = new Circle();
        editor.DrawShape(c);
    }
}

```

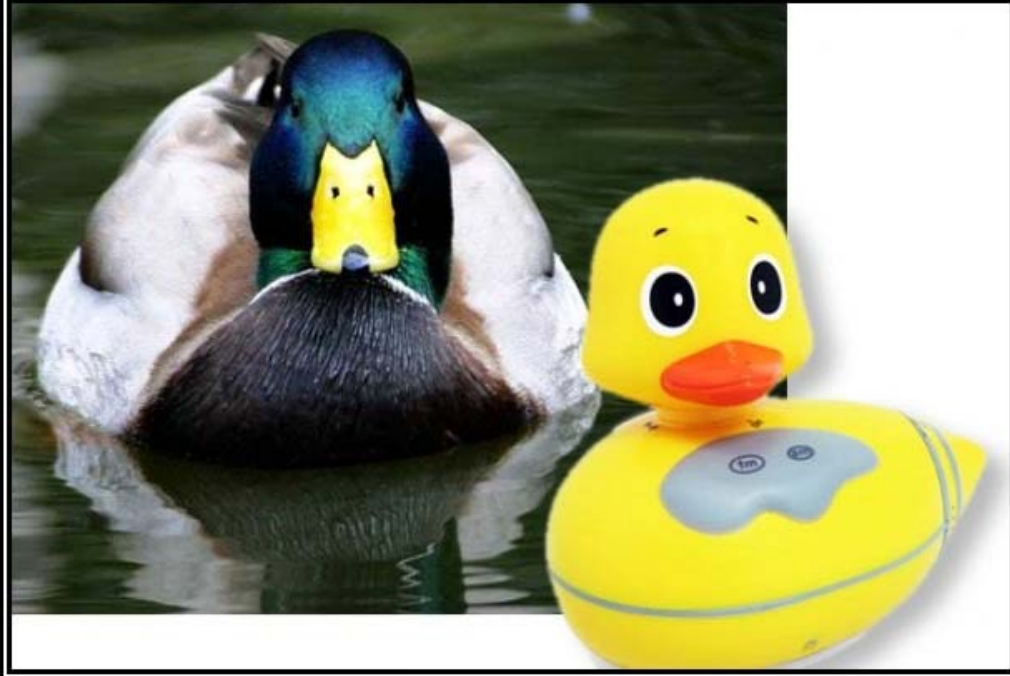
Koda dikkat ederseniz, hi bir kořula bakılmadan, benim gnderdiđim tre uygun bir izim iřlemi gerekleřmiř oluyor. Artık uygulamaya yeni bir řekil eklemek istersem tek yapmam gereken onu **Shape** isimli **abstract** sınıftan tretmek ve **Draw** methodunu override etmek okadar.

Mehmet Aydın nl

[aydinunlu85@gmail.com](mailto:aydinunlu85@gmail.com)

<http://www.aydinunlu.blogspot.com>

## C- LISKOV' S SUBSTITUTION PRINCIPLE



### LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Bu prensibin ana fikri; **türeyen sınıfların üyeleri, temel sınıfın üyeleri ile tamamen yer değiştirebilir ilkesine dayanır.**

Bu dediğimizi bir örnek üzerinde görmek herşeyin daha net anlaşılmasını sağlar elbette. Ozaman hemen örneğimize geçelim.

Elimizde **Rectangle** adında bir sınıf olsun. Bu sınıf, **Width** ve **Height** adında iki tane özellik barındırsın ve bu özelliklerin çarpımını hesaplayıp geri döndüren **GetArea** adında bir method barındırsın. Bu methodun görevi sadece alan hesabı yapmak okadar. Birde bu sınıftan türeyen **Square** adında bir sınıf olsun.

```

public class Rectangle
{
    public int Width { get; set; }
    public int Height { get; set; }

    public int GetArea()
    {
        return Width * Height;
    }
}

public class Square : Rectangle
{
    private int width;
    public int Width
    {
        get { return width; }
        set
        {
            width = value;
            height = value;
        }
    }

    private int height;
    public int Height
    {
        get { return height; }
        set
        {
            height = value;
            width = value;
        }
    }
}

```

Şimdi yukarıdaki kodlarda bulunan, **Square** sınıfı içindeki özelliklerin set bloklarına dikkat etmenizi istiyorum. Gördüğünüz gibi **Width** veya **Height** özelliklerinden herhangi birine bir değer atadığınız zaman diğerinde aynı değer atanıyor. Bunu neden yaptık dersiniz, bir karenin 4 kenar uzunluğuda birbirine eşit olmalıdır diyebiliriz. Böylece bu sınıfı kullanıp bu özelliklere değer atarken olası bir yanlış hesabın önüne geçmiş oluyoruz...

Sınıflarımızı yazdıktan sonra test kodlarımızı yazmaya geçebiliriz. Şimdi sizden şöyle düşünmenizi istiyorum. Biz bu sınıfları kullanmak istediğimizde, direkt bir nesne oluşturma işlemi yapmayalım da, gerekli işlemleri yapan bir factory sınıfımız olduğunu varsayalım. Bu factory sınıfının içinde de **GetRectangle** adında geriye **Rectangle** sınıfının referansını döndüren bir method olsun. Şu durumda

Factory' nin nasıl çalıştığı ve methodun içeriği çokta önemli değil. Şimdilik önemli olan sadece, geriye bir **Rectangle** referansı döndüren **method imzasına** sahip bir method olması. Fakat koda dikkat ederseniz geriye dönen sınıf Rectangle tipinde değil de, **Square** tipindedir. Yani aşağıdaki gibi;

```
public static class Factory
{
    public static Rectangle GetRectangle()
    {
        // ...
        Return new Square();
    }
}
```

Şimdi bu factory sınıfımızı kullanarak örneğimizi test edelim.

```
class Program
{
    static void Main(string[] args)
    {
        Rectangle rect = Factory.GetRectangle();

        rect.Width = 5;
        rect.Height = 10;

        Console.WriteLine(rect.GetArea());
    }
}
```

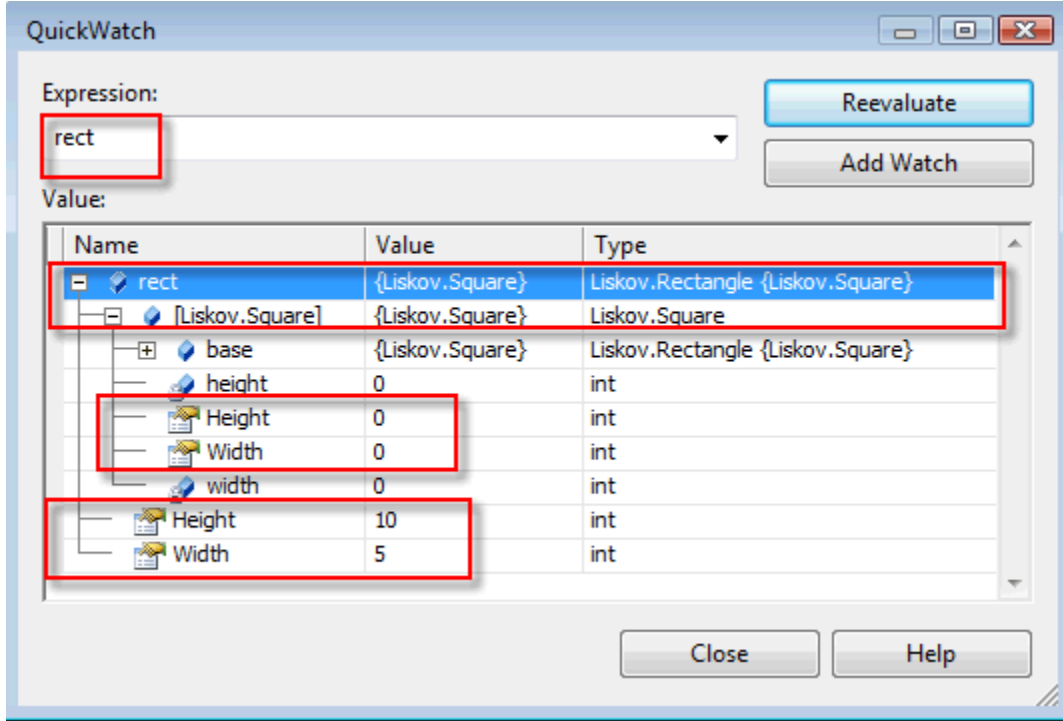
Şimdi herşeyden önce programı çalıştırıp çıktıya bakarsanız eğer, ekrana 50 yazdığını görürsünüz. Fakat 100 olması gerekmiyor mu sizce de ? Çünkü az önce Square sınıfı içinde özelliklere değer atanırken, en son atanan değer diğer özelliğe de atandığını söylemiştim. O zaman burada **Height** özelliğine atadığım değer, otomatik olarak **Width** özelliğine de atanması gerekmiyor mu ? Eğer böyle olsaydı  $10*10 = 100$  sonucuna ulaşmam gerekirdi. İşte işin püf noktasıda tam olarak burası. Eğer bunu gördüyseniz olayı biraz kavramış olmanız gerekiyor. Şimdi işin ispatına geçelim.

```
Rectangle rect = Factory.GetRectangle();
```

Bu satıra dikkat ederseniz eğer, ben **rect** adında bir **Rectangle** nesnesi örnekliyorum. **GetRectangle** 'ın method imzasına göre geriye bir Rectangle döndürmesi gerekse bile, bu method kendi içinde aslında geriye bir Square tipi döndürüyor. Fakat oluşan nesne bir Rectangle referansına atandığına göre bir Rectangle olması gerekmiyor mu ? Hayır tabiki de, çünkü zaten Square sınıfı, Rectangle sınıfından türediği için her Square aslında bir Rectangle 'dır. **Fakat burada önemli olan şey, Square sınıfının kendi içinde tanımlanan özelliklere göre değilde, türediği sınıf olan Rectangle 'ın**

özelliklerine göre davranıyor olması. Liskov 'un prensibi de şunu söyler zaten; türeyen sınıfın özellikleri, temel sınıfın özellikleri ile yer değiştirebilir.

Şimdi programı debug edip, Quick Watch' tan **rect** nesnesinin gerçek tipinin ne olduğunu ve hangi özelliklerine hangi değerlerin atandığına bakalım.



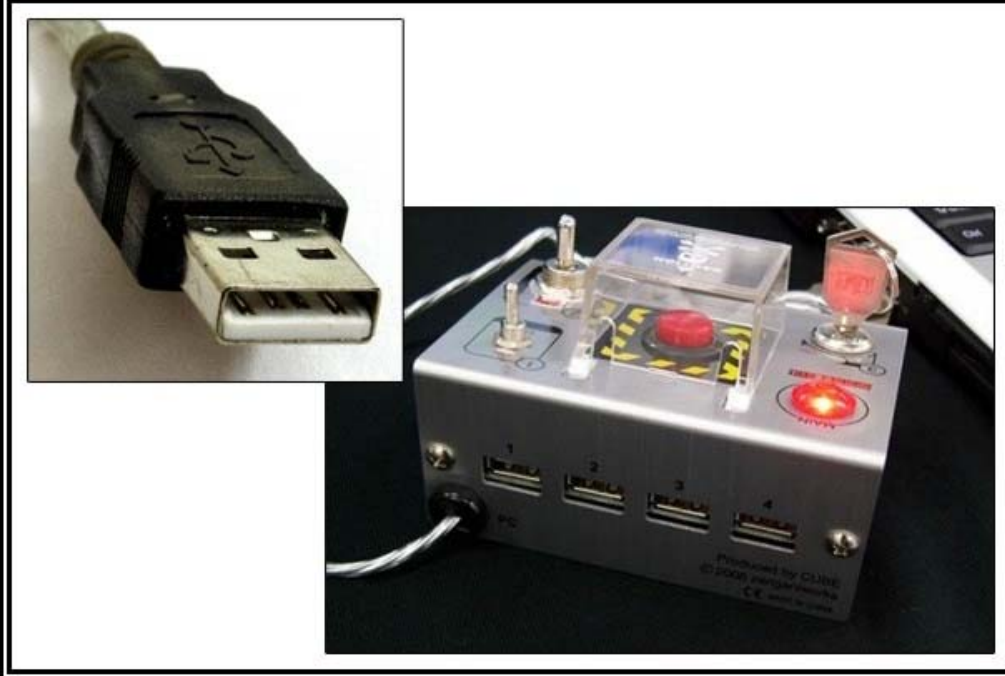
Resimden de anlaşılan şu ki, Square nesnesi Width ile Height değerlerini Rectangle içindeki özelliklere atamış. Bu durumda bizim Square sınıfı için yazdığımız Width ve Height özelliklerinin birbirlerine eşit olması gerektiğini belirtken kısıt ihlal edilmiş oluyor. Bu durumdan kurtulmanın yolu ise, Square sınıfını Rectangle' dan bağımsız bir şekilde tanımlamaktır.

Mehmet Aydın Ünlü

[aydinunlu85@gmail.com](mailto:aydinunlu85@gmail.com)

<http://www.aydinunlu.blogspot.com>

## C- INTERFACE SEGREGATION PRINCIPLE



### INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

Öncelikle bu prensibin dayandığı ana fikri genel çerçevede açıklamaya çalışıp, ardından bir örnek üzerinden giderek yazımıza devam edelim.

Elimizde ortak özellikler barındıran bir çok sınıf olduğunu düşünelim. Genelde böyle durumlarda bu sınıfları ortak olan tek bir interface 'i uygulayarak oluştururuz. Bu elbette ilk etapta mantıklı ve işimizi gören bir yaklaşımdır. Fakat bizim geliştirdiğimiz uygulama tabii ki genişletilebilir bir yapıya sahip olacak ve ileride uygulamamıza bu sınıflara benzer yeni sınıflar ekleme ihtiyacı duyacağız. Bunu yaparken tabii ki ilk yapacağımız şey, bu yeni sınıfları da aynı interface' i uygulayarak yaratmak olacaktır. Fakat yeni sınıfımızın yapısı gereği, bu interface içinde bulunan bazı üyeleri kullanması doğru ve mantıklı değildir. İşte bizim sorunumuz da tam bu noktada ortaya çıkıyor. **Sınıfların, interface içinde kullanmayacağı kesin olarak belli üyeler varsa ve bunlar sınıfın yapısına aykırı bir durum sergiliyorsa, sınıfın bu interface 'i uygulaması doğru değildir.**

Çözüm ilk etapta yeni bir interface yaratıp, yeni sınıfa onu uygulamak gibi görünse de bu durum da pek sağlıklı değildir. Çünkü ileride bu tarz sınıfların sayısının arttığı bir durumda, sınıf hiyerarşisi içinden çıkılmayacak kadar karışık bir hal almaya başlar. Bunu da kimse istemez herhalde. Peki ozaman

ne yapacağız? Çözüm okadar da zor değil aslında. **Tek yapmamız gereken şey, interface leri oluştururken barındırdığı üyeleri ortak olacak şekilde parçalayıp, bu üyeleri farklı interface 'ler altında toplayıp, ayrı ayrı interface' ler oluşturmak okadar.**

Bu genel açıklamadan sonra şimdi somut bir örnek ile ne demek istediğimizi çok daha net bir şekilde örnekleyelim. Kullanacağım örnek genelde bu prensibin anlatılırken kullanıldığı bir senaryoya sahip. Bende bu örneğin, sorunu ve çözümünü çok güzel açıkladığını düşündüğüm için aynen kullanacağım.

Örneğimiz de işçilerin ortak özelliklerini barındıran **IWorker** adından bir interface mevcut. Bu interface içinde işçilerin çalışma yeteneğini gösteren **Work** adında bir method ve yemek yeme yeteneğini gösteren **Eat** adında bir method var. Bunun dışında bu interface 'i uygulayan normal bir çalışan olan **Worker** ve daha çok çalışan **SuperWorker** adında 2 tane işçi sınıfı var. Bütün işçi tipleri de bu interface 'i uygulayarak yaratılıyor. Yani biz yeni bir işçi sınıfı tanımlamak istersek bunu da **IWorker** interface' ini uygulayarak yaparız. Böylece bizim bütün işçilerimiz hem çalışıp hem de yemek yiyebilme yeteneklerine sahip olmuş oluyor. Son olarak birde **Manager** adında, işçilerin ne yapmasını gerektiğini söyleyen bir sınıf mevcut.

```
public interface IWorker
{
    void Work();
    void Eat();
}

public class Worker : IWorker
{
    public void Work()
    {
        Console.WriteLine("Az is yapar");
    }

    public void Eat()
    {
        Console.WriteLine("Az yemek yer");
    }
}

public class SuperWorker : IWorker
{
    public void Work()
    {
        Console.WriteLine("Cok is yapar");
    }

    public void Eat()
    {
        Console.WriteLine("Cok yemek yer");
    }
}
```

```

    }
}

public class Manager
{
    IWorker worker;

    public void SetWorker(IWorker w)
    {
        worker = w;
    }

    public void Manage()
    {
        worker.Work();
    }
}

```

Buraya kadar herşey çok güzel ve sorunsuz görünüyor. Şimdi size herşeyin açıklanmasına neden olacak soruyu sormak istiyorum. Uygulamayı geliştirdiğiniz fabrika, insan işçilerin yanına bir de robot işçiler almaya karar verdi ! Robot olmasına rağmen sonuçta o da bir işçidir. Bu durumda robot sınıfının mutlaka IWorker interface' ini uygulaması gerekiyor. Fakat robotlar yemek yemez. Biz bu örnekte sadece yemek yeme olayı üzerinden gittik. Fakat bu kompleks bir sınıf olsaydı, robot yapısına aykırı bir çok olay olacaktı.Örneğin maaş gibi veya izin kullanma hakkı gibi.Bu örnekler çoğaltılabilir. Hal böyleyken bu kadar farklı yeteneklere sahip yeni bir robot sınıfının, bu özellikleri bünyesinde barındırması onu robot olmaktan çıkarır aslında. Bu da asla doğru bir tasarım şekli değildir.

Çözüm yukarda da dediğimiz gibi temel interface ' i parçalayıp, çalışabilen ve yemek yiyebilen gibi 2 farklı interface yaratmak. Sonra eklenecek olan işçi sınıflarına bu interfacerler 'den uygun olanlarını seçip, onları uygulamak. Kodun revize edilmiş hali aşağıdaki gibidir.

```

public interface IWorkable
{
    void Work();
}

public interface IFeedable
{
    void Eat();
}

public class Worker : IWorkable, IFeedable
{
    public void Work()
    {
        Console.WriteLine("Az is yapar");
    }
}

```

```

    }

    public void Eat()
    {
        Console.WriteLine("Az yemek yer");
    }
}

public class SuperWorker : IWorkable, IFeedable
{
    public void Work()
    {
        Console.WriteLine("Cok is yapar");
    }

    public void Eat()
    {
        Console.WriteLine("Cok yemek yer");
    }
}

public class Robot : IWorkable
{
    public void Work()
    {
        Console.WriteLine("Cok fazla is yapar, yemek yemez");
    }
}

public class Manager
{
    IWorkable worker;

    public void SetWorker(IWorkable w)
    {
        worker = w;
    }

    public void Manage()
    {
        worker.Work();
    }
}

```

Interface 'lerin parçalanıp ayrı ayrı ele alınması gerektiğini söyleyen tasarım prensibimiz işte bu kadar arkadaşlar. Aslında şunu söylemekte fayda var diye düşünüyorum. Diğer tüm prensiplerde olduğu gibi bu prensipte de, zor olan şey; **prensiplere uygun bir şekilde kod yazmak değil, prensiplere aykırı**

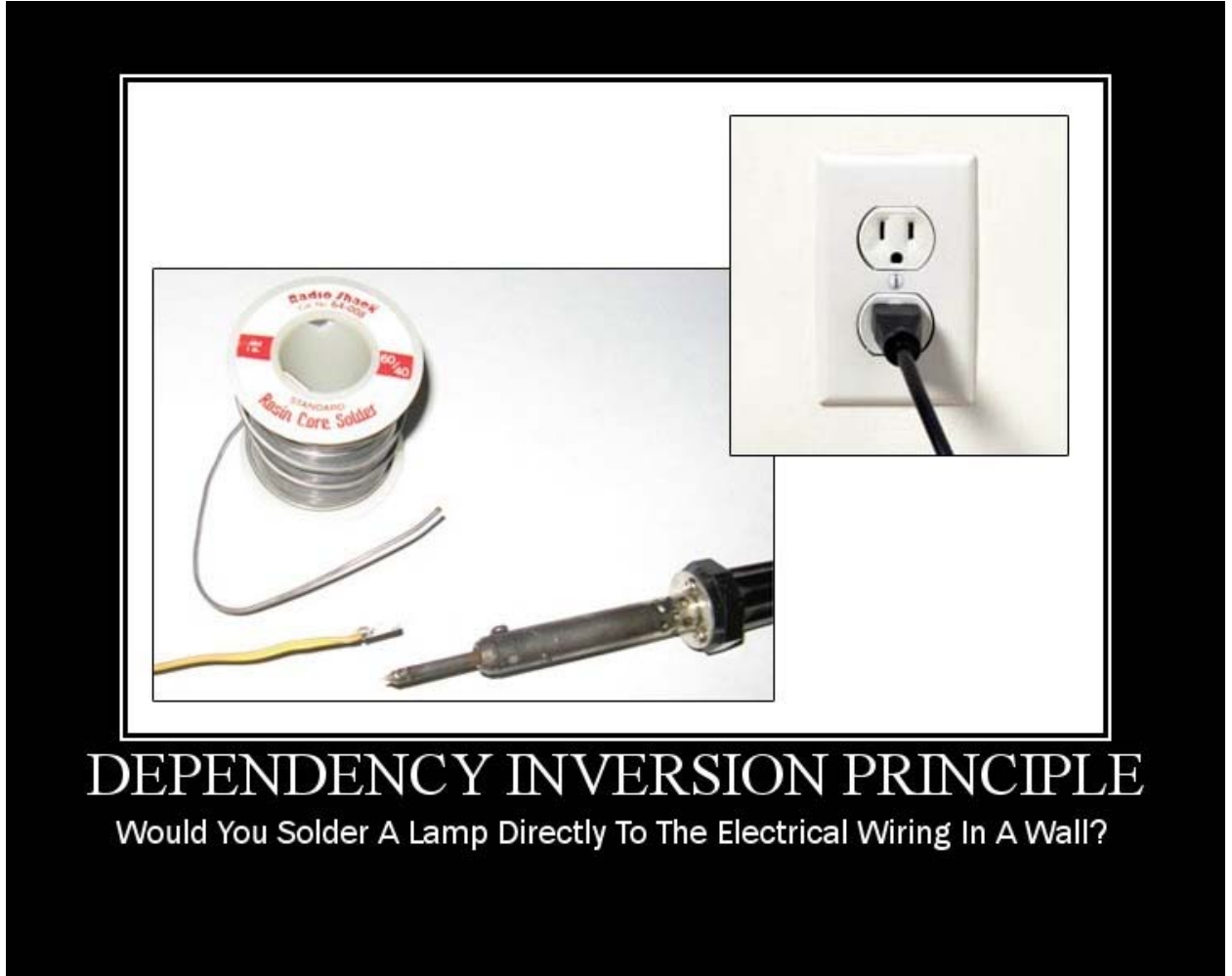
**olan durumları tespit etmektir.** Bu da aslında öyle çok zor bir şey değil ve zamanla aşılabilecek bir durumdur. Bu tarz tasarımsal problemler ile karşılaştıkça ve buna kafa yordukça, herhangi bir sınıf hiyerarşisine veya koda şöyle bir bakınca, bir çok eksikliğini ve güzelliğini görebilecek duruma geliyorsunuz zaten.

Mehmet Aydın Ünlü

[aydinunlu85@gmail.com](mailto:aydinunlu85@gmail.com)

<http://www.aydinunlu.blogspot.com>

## D- DEPENDENCY INVERSION PRINCIPLE



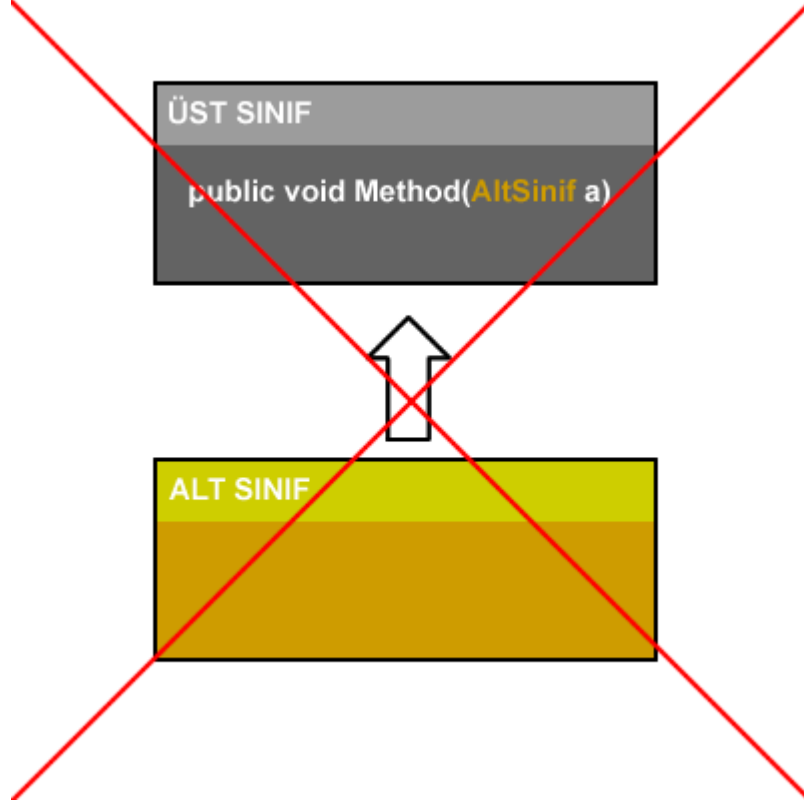
Bu prensipte anlatılmak istenen şeyi Türkçe'ye çevirince ortaya, bağımlılıkların **tersine çevrilmesi** gibi ilk bakışta anlaşılması zor olan bir şey çıkıyor. Fakat şimdilik bunun böyle isimlendirildiğini bilmeniz yeterli. Biz hemen konuyu daha net anlaşılması için açalım.

Öncelikle bu prensibin dayandığı ana fikri söylemekte fayda var. Şöyle ki; **herhangi bir sınıf ile herhangi bir başka sınıf arasında, doğrudan doğruya bir bağıın olması sakıncalıdır**. Bu durum genelde bir sınıf, diğer bir sınıfı kendi içinde örnekliyorsa ortaya çıkar. Böyle bir durumda, kendi bünyesinde başka bir sınıf örnekleyen sınıfa **üst sınıf**, örneklenen sınıfa da **alt sınıf** adını verelim. Şimdi demek istediğimizi şöyle basit bir şekilde örnekleyelim; üst sınıf, herhangi bir methodunda, alt sınıfı parametre olarak almamalıdır. Veya üst sınıf, alt sınıf tipinden bir property 'e sahip olmamalıdır. Ve bunlar gibi bağımlılığa neden olacak kullanım şekillerini örnek verebiliriz...

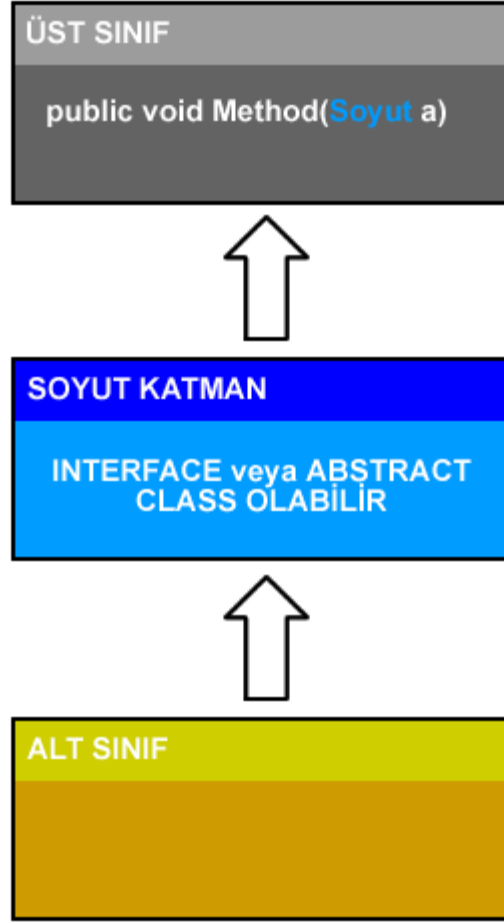
Peki böyle bir durum yaratmadan nasıl sınıf hiyerarşimizi kurabiliriz. **Çözüm; üst sınıf ile alt sınıf arasında, soyut bir katman koymak ile çözülebilir. Diğer bir deyişle, alt sınıflar ortak bir Interface 'den**

veya Abstract bir sınıftan türetilip, üst sınıfın da bu Interface 'e veya Abstact Class 'a bağımlı olması sağlanabilir.

Aşağıda yanlış ve doğru tasarım şekillerini görebilirsiniz.



Şekil 1. Yanlış tasarım



Şekil 2. Doğru Tasarım

İlk resimde, alt sınıf ile üst sınıf arasında doğrudan bir ilişki olduğu için kötü bir tasarımdır. Fakat ikinci resimde de gördüğümüz gibi bu iki sınıf arasına bir katman eklenince çok kolay bir şekilde doğru tasarımı elde ediyoruz ve alt sınıfları soyutlamış oluyoruz.

Aslında yapılmak istenen şey tamamen şudur; öyle bir yapı olmalıdır ki, üst sınıf ilerde sisteme eklenebilecek yeni tüm alt sınıfları kullanabilsin. Burada yapılan işlemde tamamen budur. Böylece üst sınıfın kullanmasını istediğimiz yeni bir sınıf eklemek istediğimiz zaman tek yapmamız gereken şey, alt sınıfı bu soyut katmandan türetmek okadar.

Şimdi bu örneği birde koda dökelim isterseniz. Örnek senaryoda üst ve alt sınıf mantığının anlaşılmasını kolaylaştırmak için alt sınıf olarak bir **Worker** sınıfı olacak ve üst sınıf olarak ta bir **Manager** sınıfımız olacak.

Önce kötü tasarıma bir bakalım.

```

public class Worker
{
    public void Work()
    {
        Console.WriteLine("Calisiyorum...");
    }
}

public class Manager
{
    Worker worker;

    public void SetWorker(Worker w)
    {
        worker = w;
    }

    public void Manage()
    {
        worker.Work();
    }
}

```

Kodda gördüğümüz gibi Manager sınıfı direk olarak Worker tipini kullandığı için Manager sınıfı, Worker sınıfına direk olarak bağımlıdır. Bu durumun zorluğunu görmek için kendimize şu soruyu soralım, **"biz bu Manager sınıfının, SuperWorker adında yeni bir sınıf ile de çalışmasını istersek ne yapacağız ?"** Yanlış çözümlerden birine örnek verecek olursak, önce SuperWorker 'ı tanımlarız. Daha sonra bunu kullanacak Manager sınıfı içindeki methodları SuperWorker' a özel olarak yazarız. Yani aşağıdaki gibi.

```

public class Worker
{
    public void Work()
    {
        Console.WriteLine("Calisiyorum...");
    }
}

public class SuperWorker
{
    public void Work()
    {
        Console.WriteLine("Daha cok calisiyorum");
    }
}

```

```

public class Manager
{
    Worker worker;

    public void SetWorker(Worker w)
    {
        worker = w;
    }

    public void Manage()
    {
        worker.Work();
    }

    SuperWorker superWorker;

    public void SetSuperWorker(SuperWorker s)
    {
        superWorker = s;
    }

    public void ManageSuperWorker()
    {
        superWorker.Work();
    }
}

```

Bu yapıya aslında bir tip kontrolü sayesinde **Manage** methodu içinde, bir if bloğu ile kontrol edebiliriz. Ya da Manage methodunu overload ederiz. Ve ya bir çok farklı senaryo ile destekleyebiliriz. Fakat sorun bu şekilde çözülmeye çalışıldığı her denemede, değişiklik yapılacak yer Manager sınıfıdır. Başka ihtimaliniz yok. Çünkü bu sınıflara bağımlısınız. Oysa DIP' in sağlamak istediği doğru tasarım bunun tam tersi olmakla beraber, şudur; **üst sınıf içinde hiçbir değişiklik yapmadan, genişletilebilir bir yapıyı geliştirebilmektir.** Ozaman çözüm yolu aşağıdaki gibi olacaktır. Yani alt sınıfları ortak bir interface olan IWorker 'ı uygulayarak geliştirmek. Sonrasında da manager sınıfını bu interface ile bağımlı hale getirmek.

```

public interface IWorker
{
    void Work();
}

public class Worker : IWorker
{
    public void Work()
    {
        Console.WriteLine("Calisiyorum...");
    }
}

```

```
    }  
}  
  
public class SuperWorker : IWorker  
{  
    public void Work()  
    {  
        Console.WriteLine("Daha cok calisiyorum...");  
    }  
}  
  
public class Manager  
{  
    IWorker worker;  
  
    public void SetWorker(IWorker w)  
    {  
        worker = w;  
    }  
    public void Manage()  
    {  
        worker.Work();  
    }  
}
```

Gördüğünüz gibi Manager sınıfı tamamen IWorker isimli interface ile bağımlıdır. Diğer bir deyişle, Manager sınıfı için önemli olan sadece IWorker interface'ini uygulamış bir şekilde gelen nesnelere. Worker veya SuperWorker olması onun için önemli değildir. Onları tanımaz bile. Bu şu demektir ki, Manager sınıfı IWorker'ı uygulayan bütün sınıflar ile çalışabilir. İşte extend edilebilir yani genişletilebilir kod yazmanın tekniklerinden biride budur.

Bu yazıyla birlikte Design Principles başlıklı yazı dizimin SOLID Principles bölümünü bitirmiş bulunuyorum. Sizde takdir edersiniz ki bu tarz fikirsel konuları anlatmak her zaman için bir teknolojinin nasıl kullanıldığını anlatmaktan daha zordur. Bende Türkçe kaynak sıkıntısı çektiğimiz bu alanda elimden geldiği kadar edindiğim bilgileri sizlerle paylaşmaya çalıştım...

Mehmet Aydın Ünlü

[aydinunlu85@gmail.com](mailto:aydinunlu85@gmail.com)

<http://www.aydinunlu.blogspot.com>