YZM 3017
Yazılım Tasarımı ve Mimarisi

Prof. Dr. Hamdi Tolga KAHRAMAN
Arş. Gör. M. Hakan BOZKURT
Arş. Gör. Sefa ARAS

TASARIM DESENLERİ

Genel olarak tasarım kalıpları:

- Creational Patterns (Oluşturucu Kalıplar)
- Structural Patterns (Yapısal Kalıplar)
- Behavioral Patterns (Davranişsal Kalıplar)

olarak ayrılır.

Yapısal Tasarım Desenleri

FACADE

Yapısal Tasarım Desenleri

Facade kelime anlamı olarak "cephe, yeni yüz" anlamlarına gelmektedir. Yani var olan bir nesneye yeni bir yüz katma yeni bir cephe katma anlamında kullanılmaktadır.

Yapısal Tasarım Desenleri

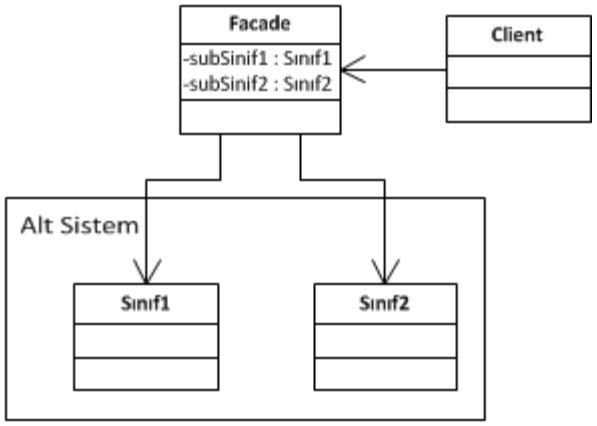
Facade tasarım deseni de GoF(Gangs of Four) olarak adlandırılan desenlerden olduğu için "Elements of Reusable Object-Oriented Software, AW,1995" kitabındaki tanım ile başlayalım.

"Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use."

Yapısal Tasarım Desenleri

- Facade tasarım deseni uygulanması en basit tasarım desenlerinden birisidir. Örneğin uygulamamızda bazı modüller var ve bazı işlemlerin gerçekleştirilmesi için bu modüllerin kullanılması gerekiyor.
- Facade tasarım deseni, istemcinin yani bu operasyonları gerçekleştirecek nesnenin kod karmaşasına bulaşmamasını sağlar ve farklı istemcilerin olduğu uygulamalarda bu kodların tekrarlanmasını engellemiş olur.
- Sonuç olarak da anlaşılması daha kolay bir kodlama yapılmış olunur.

Yapısal Tasarım Desenleri



Facade tasarım deseni için uml diyagramı

- Görüldüğü gibi client alt sisteme direkt erişmeyip, bu alt sistemi kullanan facade nesnesi üzerinden işlem yapıyor.
- Katman mimarisi olarak düşündüğümüzde clientın olduğu katmanda alt sistem referansı eklemeye gerek yoktur.
- Facade nin olduğu katmanda alt sistem referansları eklenmelidir.

FACADE Tasarım Deseni

- Büyük bir binaya dışardan bakıldığı düşünülsün, binanın içsel detayları dışarıdan görülmez ancak binanın dış cephesi yani facade'ı görülebilir.

- Mimarlar bu detaylarla oluşacak çirkinliği gizlemek ve bir anlamda algı kolaylığı oluşturmak için binalarına şık cepheler yani facade'ler tasarlama yoluna gitmektedir.

- Benzer şekilde bir kütüphane ya da framework'ün tasarımını yapan programcılar ise o kütüphaneyi kullanacak olan kişilere algı kolaylığı sağlamak amacı ile kütüphaneye genellikle bir facede sınıfı eklemektedir.

- Kullanıcılar ise bu facade nesnesi üzerinden kütüphanedeki diğer nesneleri kullanırlar.

FACADE Tasarım Deseni

- Büyük bir kütüphane ya da framework çeşitli alt sistemlerden (subsystem) oluşur.
- Genellikle facade olan sınıflar, bu alt sistemlerin kütüphanenin kullanıcısı tarafından yüksek seviyeli soyutlama sağlayarak daha kolay kullanımına imkan sunar. Facade bir başka bakış açısıyla kütüphane için ortak bir giriş çıkış arabirimidir.
- Facade patern'i özellikle çok katmanlı mimariye sahip uygulamaların geliştirilmesinde kullanılır.
- Zira facade sınıfları katmanların birbirinden soyutlanmasına böylece katmanların birbirine gevşek bağlı olmasına hizmet eder.
- Böylece katmanlar birbirinden bağımsız olarak geliştirilebilirler.

Facede Tasarım Deseni Nasıl Çalışır?

FACADE Tasarım Deseni

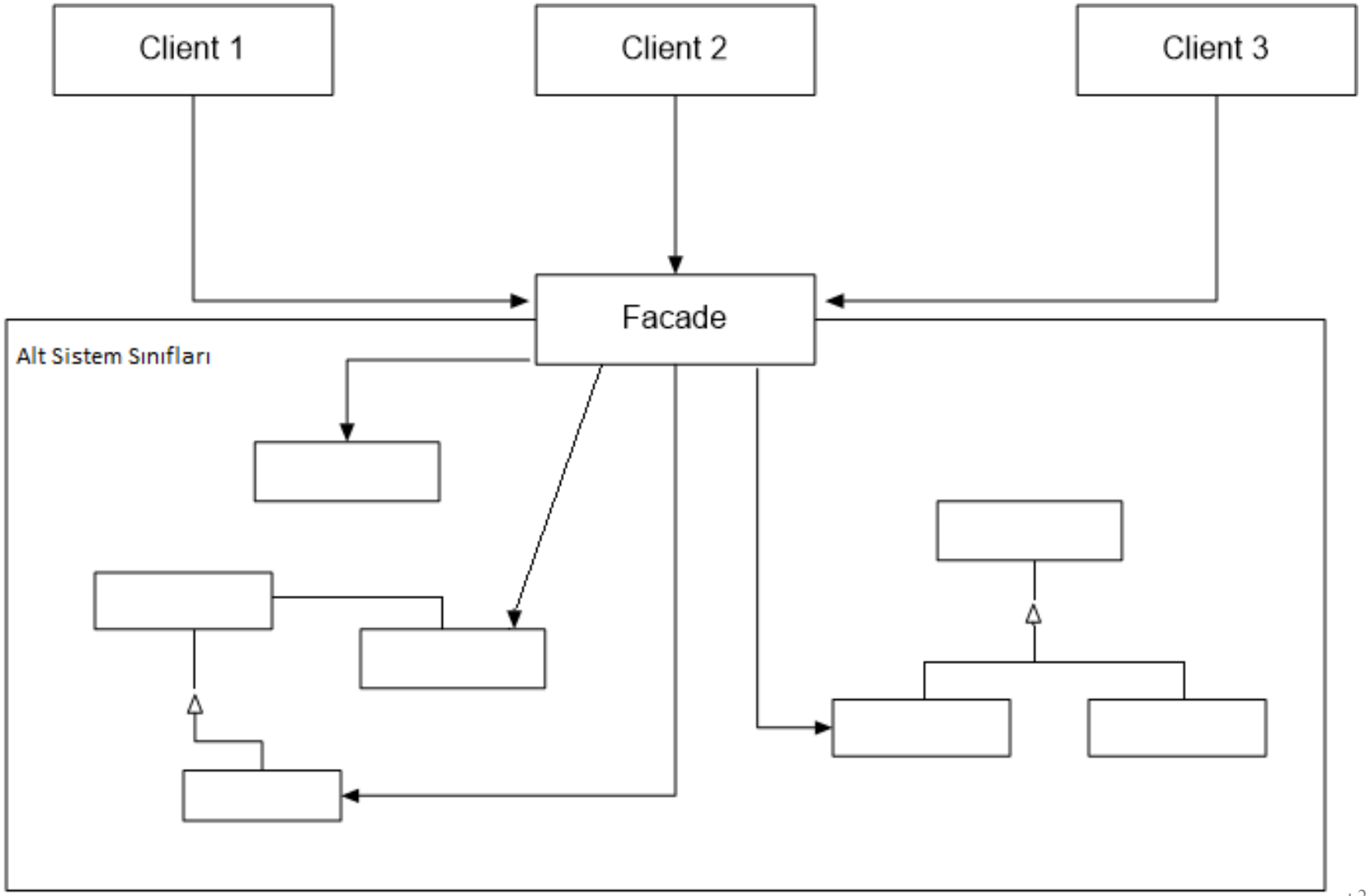
- Alt sistemlerin direkt kullanılması yerine, alt sistemlerin kullanılmasını sağlayan arayüzler yazılarak kullanım kolaylaştırılır.

- Bu arayüzleri gerçekleyen sınıflara facade sınıfları denir.

- Bu tasarım deseninde, alt sistemde birden fazla sınıf ve bu sınıflar arasında ilişkiler bulunur ve herhangi bir işlemi gerçekleştirmek için bu sınıflardaki metotları belli bir sırayla çağırmak gerekir.

- Buna daha yüksek seviyeli bir işlem yapmak gerekir de diyebiliriz. Bu durumda bu metotları kendi içerisinde sarmalayan yeni sınıflar yazılır.

FACADE Tasarım Deseni



FACADE Tasarım Deseni

- Görüldüğü üzere Client kodu sisteme direkt erişmiyor, belli facade sınıfı üzerinden sisteme erişiyor.

- Karmaşık alt sistem sınıfları, şekilden de görüleceği üzere **sarmalanmış**.

- Bu sistemdeki client kodu, sadece arayüz tarafından gelen bir istek olarak düşünülmemelidir; bir web servisi için facade yazılmış ve siz kendi kütüphanenizde o facade'i kullanıyor olabilirsiniz.

- Yani kim kullanmak istiyorsa o client.

FACADE Tasarım Deseni

Ne Zaman Kullanılır?

FACADE Tasarım Deseni

- Uygulamada birden fazla modül olduğunda herhangi bir işlemi gerçekleştirmek için birden fazla sayıda sınıftan örnek oluşturulur, metotlar çağırılır.

- Uygulama en baştan yapılırken tek sorumluluk prensibine uygun olarak en küçük modüllere ayırarak tasarlanacağı için, yüksek seviyeli işlemlerde birden fazla referans ekleyip kod yazmak gerekir.

- Bu durumda ilerleyen süreçte kopyala-yapıştır işlemlerinin fazlasıyla artması anlamına gelir.

FACADE Tasarım Deseni

- Tam bu noktada imdadımıza facade tasarım deseni yetişir. Çoğu yazılımcının da zaten kullandığı bu deseni kullanmak, kod karmaşıklığını engelleyip sadeliği sağlar. Alt sistemin karmaşık olduğu durumlarda kullanılması büyük avantaj sağlar.

Eğer facade deseni kullanılıyorsa amacı unutulmamalıdır:

- Karmaşık ve küçük parçalardan oluşan sistemlere basit bir arayüz sunmak.
- Yani zaten anlaşılması basit bir işlem için facade yazmak gereksiz olur.

FACADE Tasarım Deseni

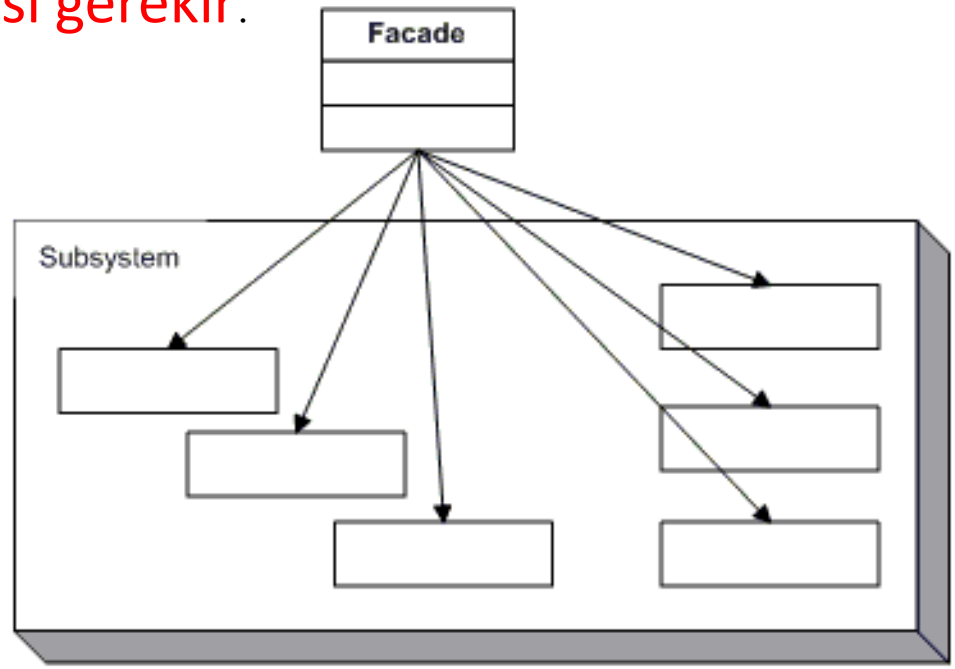
- Aynı alt sistemi veya sistemleri, birden fazla sayıda proje kullanıyorsa eğer alt sistemde olan bir değişiklik sonrası yine birçok yerin değiştirilmesi de gerekebilir.
- Facade deseni kullanıldığında sadece ilgili arayüz sağlayan sınıfı değiştirmek yeterlidir.
- Facade yapısı kurulduğunda arayüzünü sağlayan sınıf sık sık değişmeyecek bir sınıf olmalıdır.
- Yani değişiklik isteğinin iskelette değil de içerikte olabileceği durumlar için kullanılmalıdır.

FACADE Tasarım Deseni

- Bu tasarım desenini bir örnek ile açıklamaya başlayalım. Örneğin çok karmaşık bir sınıf kütüphaneniz veya çok karmaşık bir modülünüz var.
- İşlemleri bu sınıflarla yapıyorsunuz. Çoğu zaman belirli bir işlemi yapmak için çok fazla sayıda nesne üretmeniz ve onlar arasında çeşitli ilişkiler kurmanız gerekir. Bunu sıkça yaptığınız için çoğu zaman kopyala-yapıştır yapmanız olasıdır.
- Facade tasarım desenine göre ise belirli bir işi büyük bir kütüphane içerisinde sık sık yapıyorsak, bu iş için özelleştirilmiş yüksek seviyeli bir arayüz tanımlamak kısa vadede fayda getirecektir.

FACADE Tasarım Deseni

- Yani alt sınıflarla yapılan komplex işleri bir üst katmanda yaptırarak hem copy-paste'ten kurtulup hemde client tarafına sade kod sağlamış olmaktayız.
- Burada önemli noktalardan biri Facade sınıf ile alt sınıflar birbirine sıkı sıkıya bağlı olmaması gerekir.
- Yani Facede sınıfı ortadan kaldırıldığında program yine çalışır halde olması gerekir.



FACADE Tasarım Deseni

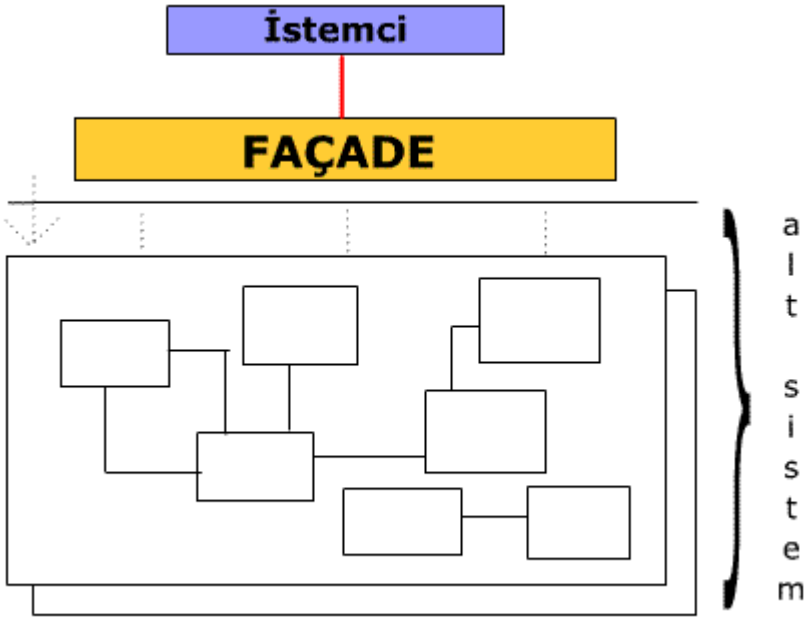
- Bir yazılım uygulaması için alt kütüphane (layer) geliştiriyorsunuz ve geliştirme aşamasında sıklıkla bu kütüphanenin arayüzünü değiştirmek zorunda kalıyorsunuz.
- Eğer kütüphane içerisinde bulunan spesifik nesneleri bu kütüphaneyi kullanan diğer modüllerde kullanırsanız yeni bir arayüz değişikliğinde çok fazla zaman kaybedeceksiniz.
- Ancak değişebilir arayüzleri bir katman daha yukarıya çıkarırsanız ana sınıfların arayüzünde yapacağınız değişiklikler sizi çok fazla zorlamayacaktır.
- En azından tek bir yerden değişiklik yaparak uygulamanızın yeni arayüze adapte olmasını sağlarsınız.

FACADE Tasarım Deseni

- O zaman şöyle bir soru akla gelebilir. Ya Facade(yeni yüz) olarak tasarladığım yeni arayüzde değişik olursa : bu noktada Facade olarak tasarlayacağımız arayüzde nelere dikkat etmemiz gerektiği önem kazanıyor.
- Yani her sınıf kütüphanesi facade tasarım desenine uygun olmayabilir.
- Facade tasarım deseni, daha çok nesne ilişkileri fazla olan ve çok sayıda sınıfın olduğu sınıf kütüphaneleri için kullanışlıdır.

FACADE Tasarım Deseni

- Aşağıdaki şekilde Facade tasarım deseninin görsel gösterimi bulunmakta. Şekilden de görüldüğü **üzere Facade tasarımındaki amaç yeni bir şey yaratmak değil var olan bir şeyi farklı bir şekilde daha kolay çözmektir.**



FACADE Tasarım Deseni

- Gelelim facade arayüzleri tasarlanırken dikkat edilmesi gereken noktalara.

- Her şeyden önce facade arayüzlerini yapmamızdaki amaç var olan kompleks bir sistemi daha kolay anlaşılabilir ve daha kolay kullanılabilir hale getirmektir.

- Eğer facade arayüzlerini kullanırken oluşturulması gereken nesne sayısı ve kurulması gereken ilişki sayısı mevcut sistemle aynı seviyede ise facade tasarımında bir yanlışlık var demektir. Bu durumda facade tasarımından sonra aynı işi daha az satır kodla yada daha kolay anlaşılabilir kod blokları yapabiliyor olmamız gerekir.

FACADE Tasarım Deseni

- Facade ile ilgili en önemli kural ise facade arayüzündeki kurulan kuralların asıl işi yapan sınıflarla sıkı sıkıya bağlı olmaması gerekmesidir.

- Yani facade arayüzünü sistemden komple çıkardığınızda mevcut sistemin aynen işini yapabiliyor olmasıdır.

- Facade varolan sisteme yeni bir yüz katmaktan öteye gitmemelidir. Aksi durumda yapılan şey facade değil sistemin olmazsa olmaz bir parçası haline gelir.

FACADE Tasarım Deseni

- Facade ile ilgili akıllara gelebilecek bir soru ise şudur: Acaba var olan bir sisteme yeni yüz katma sitemin idare edilebilirliğini (management) olumsuz yönde etkiler mi?
- Evet bir noktada doğru. Ancak bize sağlayacağı fayda zararından fazla olacaktır. Nitekim kompleks bir sistemde yaptığınız bir değişikliği sadece facade arayüzünde değiştireceksiniz.
- Buradan da şu sonuç çıkıyor , facade arayüzünde (sınıflar yada sınıfların metotları) değişime çok fazla açık olmaması gerekiyor.

FACADE Tasarım Deseni

- Örneğin facade arayüzünü (facade olarak geliştirdiğiniz sınıfların kendi aralarındaki ilişki ve metotların prototipinden bahsediyorum) de sıklıkla değiştiriyorsanız bu noktada bir sorun var demektir.

- Facade arayüzleri kesin olarak ihtiyaç duyulan ve sistem nasıl olursa olsun ona ihtiyacımız olacak durumlarda tasarlanmalıdır.

FACADE Tasarım Deseni

Sonuç

- Facade kompleks bir mevcut sistemin fonksiyonelitesini daha üst seviyede soyutlayarak kullanılmasını amaçlamaktadır.

- Facade tasarımının amacı tüm sistemi yeni bir alt sistem olarak açığa çıkarmak değil spesik bir işi örneğin veritabanında sorgu çalıştırıp sonucunu almayı daha basit ve anlaşılır kılmaktır.

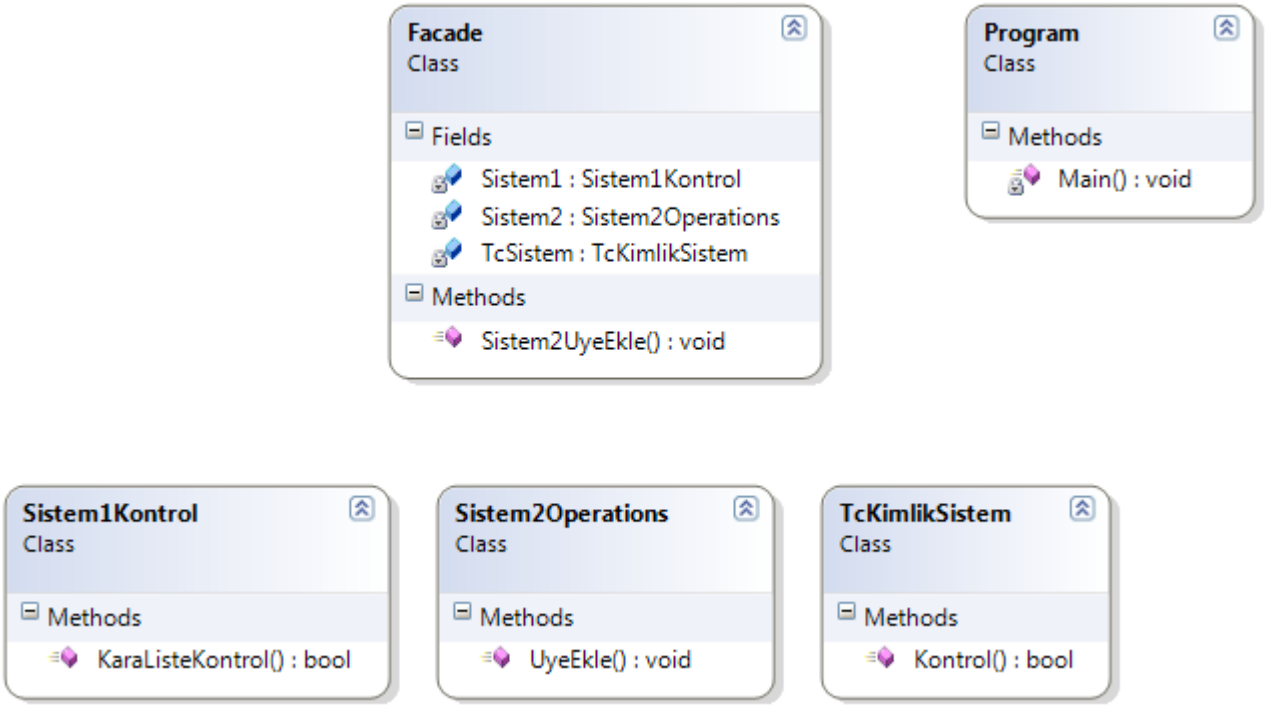
FACADE Tasarım Deseni

Şimdi facade tasarım deseni ile ilgili ufak bir uygulama yapalım. Örnek senaryomuz şu şekilde olsun.

- Sistem1 ve sistem2 adında üyelik modülünün olduğu 2 sistemimiz var.
- Sistem2 ye üye olunacağında Sistem1 de kara listede olup olmadığını ve kimlik numarasının doğru olup olmadığını kontrol edip üyeliği buna göre kabul ediyoruz.
- Bu senaryoyu facade tasarım deseni ile gerçekleştirelim. Uygulamamızın sınıf diyagramı aşağıdadır.

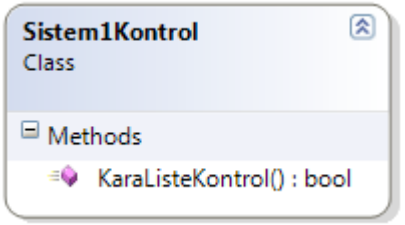
FACADE Tasarım Deseni

Uygulamamızda ki **sistem1Kontrol**, **Sistem2Operations** ve **TcKimlikSistem** sınıfları alt sistemlerdir. Program yani client bu sınıfları direkt olarak kullanmayıp oluşturulan Facade sınıfı üzerinden kullanır.

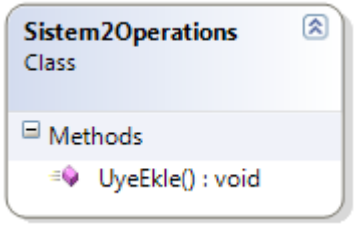


FACADE Tasarım Deseni

```
public class Sistem1Kontrol
{
    public bool KaraListeKontrol(string Tc)
    {
        //kontrol edildiğini varsayalım
        return false;
    }
}
```

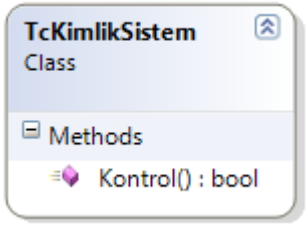



```
public class Sistem2Operations
{
    public void UyeEkle(string Tc)
    {
        Console.WriteLine("{0} Üye Eklendi",Tc);
    }
}
```



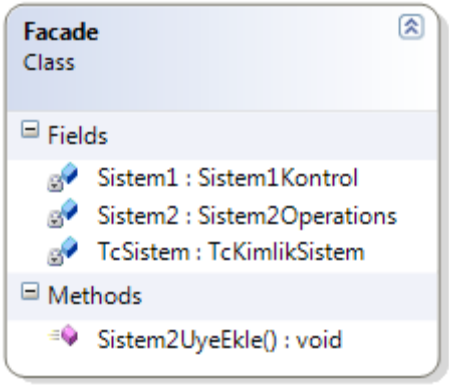
FACADE Tasarım Deseni

```
public class TcKimlikSistem
{
    public bool Kontrol(string Tc)
    {
        //kontrol edildiğini varsayalım
        return true;
    }
}
```



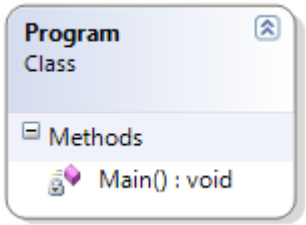
FACADE Tasarım Deseni

```
public class Facade
{
   //constructor da oluşturulabilir
   //singleton olarak tasarlanabilir
   TcKimlikSistem TcSistem = new TcKimlikSistem();
   Sistem1Kontrol  Sistem1 = new Sistem1Kontrol();
   Sistem2Operations  Sistem2 = new   Sistem2Operations();

   public void Sistem2UyeEkle(string Tc)
   {
      if (TcSistem.Kontrol(Tc) && !Sistem1.KaraListeKontrol(Tc))
      {
         Sistem2.UyeEkle(Tc);
      }
   }
}
```



FACADE Tasarım Deseni

```
class Program
{
    static void Main(string[] args)
    {
        Facade f = new Facade();
        f.Sistem2UyeEkle("123123");
        Console.ReadKey();
    }
}
```

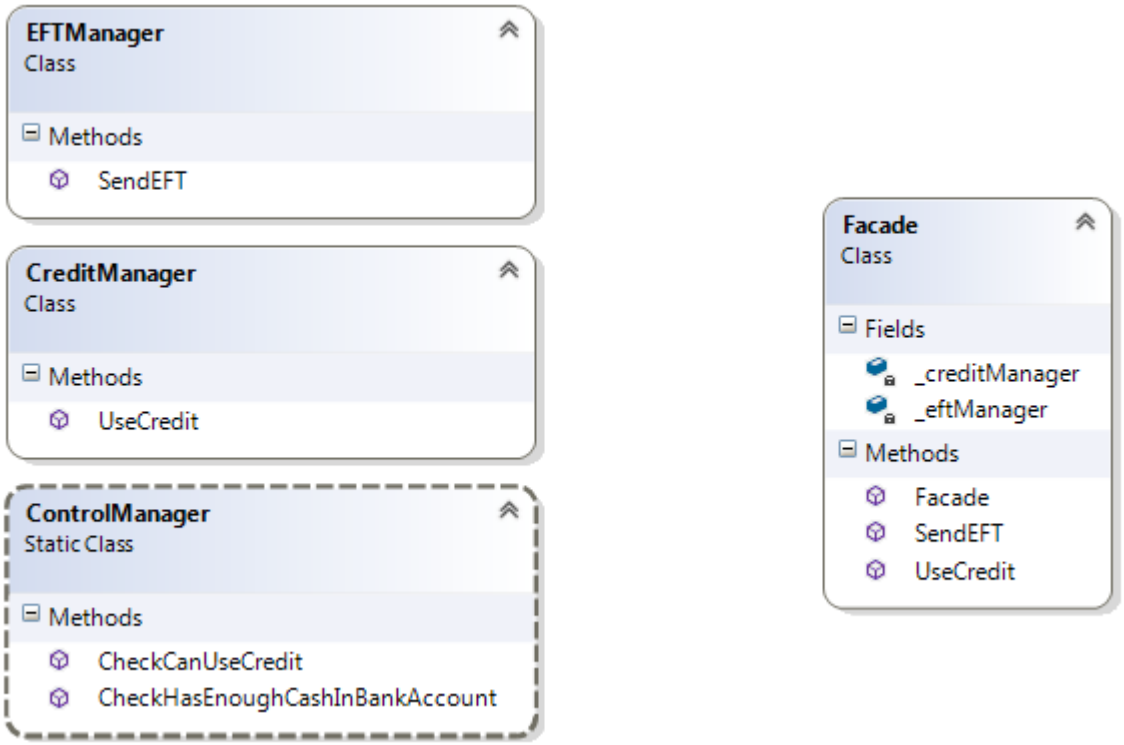


Uygulamamızda ki sistem1Kontrol, Sistem2Operation ve TcKimlikSistem sınıfları alt sistemlerdir. Program yani client bu sınıfları direkt olarak kullanmayıp oluşturulan Facade sınıfı üzerinden kullanır.

FACADE Tasarım Deseni

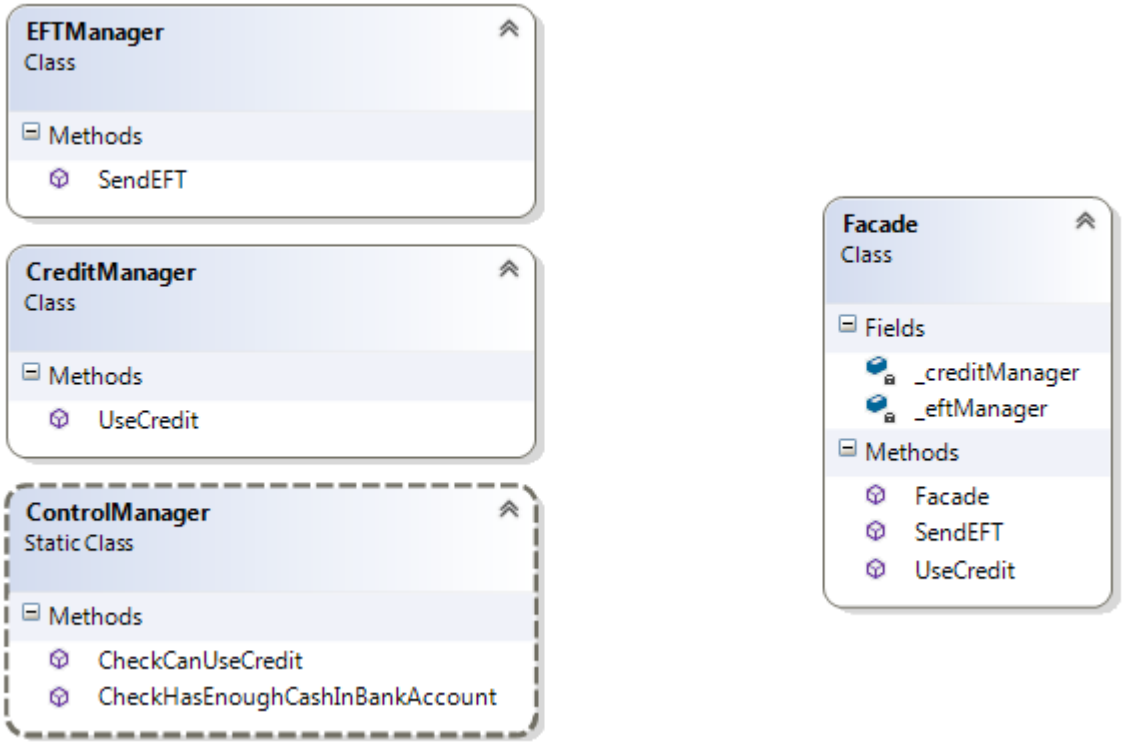
- Örnek bir uygulama olarak bankacılık uygulaması düşünebiliriz. Örneğin EFT yapan , kredi kullandıran ve bu işlemleri yaparken hesapta yeterli kredi/bakiye olup olmadığı, kişinin hesabında bloke bulunup bulunmadığı gibi çeşitli bankacılık kontrolleri yapıldıktan sonra işleme onay veren uygulamayı facade tasarım deseni kullanarak gerçekleştiriniz.

FACADE Tasarım Deseni



- Facade kullanmazsak client tarafı bir EFT işlemi yaptırmak için EFT işlemi yapan sınıfı tanımlaması gerekir. Sonra Kredi kullandırmak için kredi sınıfı daha sonra başka bir işlem için o işlem sınıfını. Ama bu işlemi Facade sınıfına bırakırsak EFT göndermek için sadece Facade sınıfı üzerindeki SendEFT metodunu çağırırsak hem clientin işi kolaylaşır hem daha sadece tekrarı az kod yazmış oluruz.

FACADE Tasarım Deseni



- Görüldüğü gibi Facade sınıfımız nesne yaratımlarını Construtor'ında yapıyor ve altsınıflara ait clientin kullanacağı metodları kendi bunyesine alarak client ile alt sınıflar arasında köprü oluşturuyor.

FACADE Tasarım Deseni

```
public class EFTManager
    {
        public void SendEFT(Customer fromCustomer, Customer toCustomer, decimaleftAmount)
        {
            if (ControlManager.CheckHasEnoughCashInBankAccount(fromCustomer, eftAmount))
            {
                fromCustomer.CashAmount -= eftAmount;
                Console.WriteLine("EFT " + toCustomer.CustomerNumber + " nolu hesaba gönderildi..");
            }
            else
                Console.WriteLine("bakiye yetersiz EFT işleminiz gerçekleştirilemedi.");
        }
    }
```

FACADE Tasarım Deseni

```
public class CreditManager
    {
        public void UseCredit(Customer customer)
        {
            if (ControlManager.CheckCanUseCredit(customer.IDNo))
                Console.WriteLine("Kredi kullandırılmıştır.");
            else
                Console.WriteLine("Banka kredinizi onaylamamıştır.");
        }
    }
```

FACADE Tasarım Deseni

- Facade nin olduğu katmanda alt sistem referansları eklenmelidir.

```
public class Facade
    {
        private EFTManager _eftManager;
        private CreditManager _creditManager;

        public Facade()
        {
            _eftManager = new EFTManager();
            _creditManager = new CreditManager();
        }

        public void SendEFT(Customer fromCustomer, Customer toCustomer, decimaleftAmount)
        {
            _eftManager.SendEFT(fromCustomer, toCustomer, eftAmount);
        }

        public void UseCredit(Customer customer)
        {
            _creditManager.UseCredit(customer);
        }
    }
```

FACADE Tasarım Deseni

```
public class Customer
    {
        public int CustomerNumber { get; set; }
        public string Fullname { get; set; }
        public string IDNo { get; set; }
        public decimal CashAmount { get; set; }
    }

public static class ControlManager
  {
    public static bool CheckHasEnoughCashInBankAccount(Customer customer,decimal amount)
    {
        if (customer.CashAmount >= amount)
            return true;
        else
            return false;
    }

    public static bool CheckCanUseCredit(string IDNo)
    {
        //test için her zaman return true
        return true;
    }
  }
```

FACADE Tasarım Deseni

➢Görüldüğü gibi client alt sisteme direkt erişmeyip, bu alt sistemi kullanan facade nesnesi üzerinden işlem yapıyor.

➢Katman mimarisi olarak düşündüğümüzde clientın olduğu katmanda alt sistem referansı eklemeye gerek yoktur.

```
class Program
    {
        static void Main(string[] args)
        {
            Facade facade = new Facade();

            Customer customer1 = new Customer() { IDNo = "1245203836", Fullname ="Ahmet",
            CustomerNumber = 11243, CashAmount = 3456090.25M };
            Customer customer2 = new Customer() { IDNo = "2342342342", Fullname ="Mehmet",
            CustomerNumber = 123123, CashAmount = 34929272.36M };

            //send eft customer1 to customer2
            facade.SendEFT(customer1, customer2, 12345);

            //use credit to customer 2
            facade.UseCredit(customer2);

            Console.ReadLine();
        }

    }
```

FACADE Tasarım Deseni

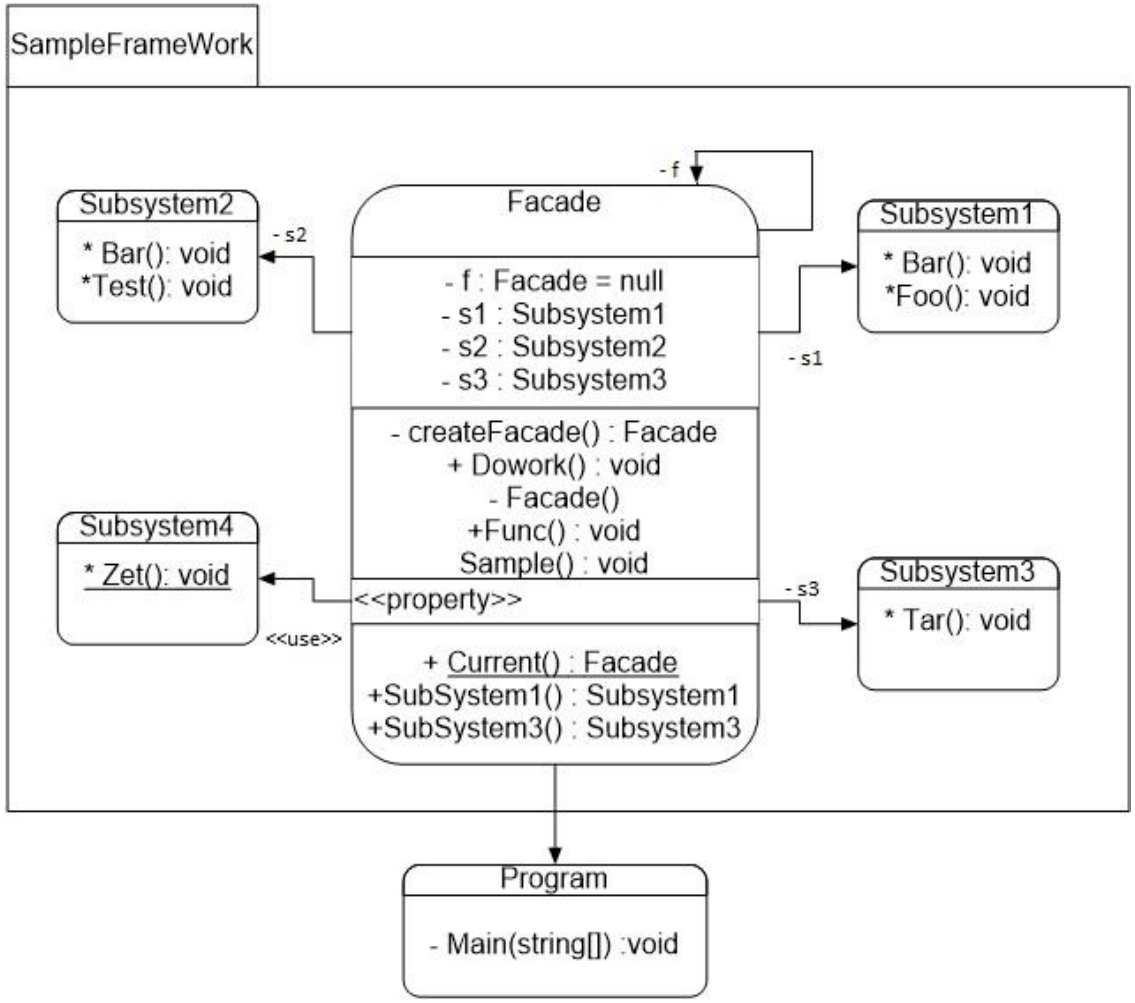
ÖRNEK:

1. Aşağıdaki sembolik örnekte dışarıdan Subsystem1 ve Subsystem3' erişilebilmekte ancak Subsystem2 ve Subsystem4 internal oldukları için erişilememektedir.
2. Facade isimli sınıf bu alt sistem nesnelerini kendi içinde barındırmakta ve onları kendi yüksek seviyeli fonksiyonlarıyla içsel olarak kullanmaktadır.
3. Facade sınıf, singleton tasarlanmış olduğu için Current isimli static property'si aracılığıyla kendi referansına ulaşılmakta ve o şekilde kullanılmaktadır.

FACADE Tasarım Deseni

Subsystem1 ve Subsystem3'e erişilebilmekte ancak Subsystem2 ve Subsystem4 internal oldukları için erişilememektedir.

Facade isimli sınıf bu alt sistem nesnelerini kendi içinde barındırmakta ve onları kendi yüksek seviyeli fonksiyonlarıyla içsel olarak kullanmaktadır.



Şekil 2. Örneğe ait UML diyagramı

FACADE Tasarım Deseni

```
namespace SampleFramework
{
    public class Facade
    {
        private Subsystem1 s1;
        private Subsystem2 s2;
        private Subsystem3 s3;
        private static Facade f = null;

        public Subsystem1 Subsystem1
        {
            get
            {
                return s1;
            }
        }
        public Subsystem3 Subsystem3
        {
            get
            {
                return s3;
            }
        }
        private Facade()
        {
            s1 = new Subsystem1();
            s2 = new Subsystem2();
            s3 = new Subsystem3();
        }
        public static Facade Current
        {
            get
            {
                return createFacade();
            }
        }
        private static Facade createFacade()
        {
            if ( f == null )
            {
                f = new Facade();
            }
            return f;
        }
        public void Sample()
        {
            s1.Foo();
            s2.Bar();
        }
        public void DoWork()
        {
            Subsystem4.Zet();
        }
        public void Func()
        {
            s3.Tar();
        }
    }

    public class Subsystem1
    {
        public void Foo()
        {
            Console.WriteLine("Subsystem1::Foo");
        }
        public void Bar()
        {
            Console.WriteLine("Subsystem1::Bar");
        }
    }
```

FACADE Tasarım Deseni

```
    internal class Subsystem2
    {
        public void Test()
        {
            Console.WriteLine("Subsystem2::Test");
        }
        public void Bar()
        {
            Console.WriteLine("Subsystem2::Bar");
        }
    }
    public class Subsystem3
    {
        public void Tar()
        {
            Console.WriteLine("Subsystem3::Tar");
        }
    }
    internal static class Subsystem4
    {
        public static void Zet()
        {
            Console.WriteLine("Subsystem4::Zet");
        }
    }
}
```

```
//////////////////////////////////////////
namespace UserApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Facade.Current.Func();
            Facade.Current.DoWork();
            Facade.Current.Sample();
            Facade.Current.Subsystem1.Foo();
            Facade.Current.Subsystem3.Tar();
        }
    }
}
```

Ekran Çıktısı:

```
Subsystem3::Tar
Subsystem4::Zet
Subsystem1::Foo
Subsystem2::Bar
Subsystem1::Foo
Subsystem3::Tar
```

Yapısal Tasarım Desenleri

ADAPTER TASARIM DESENİ

ADAPTER TASARIM DESENİ

Nedir?

- Adapter tasarım deseni, structural tasarım desenlerinden biridir. Bu tasarım deseni, birbiriyle ilişkili olmayan interface'lerin birlikte çalışmasını sağlar. Bu işlemi ise, bir sınıfın interface'ini diğer bir interface'e dönüştürerek yapar.

Ne zaman Kullanılır?

Farklı interface'lere sahip sınıfların birbiriyle çalışabilmesini sağlamak amacıyla kullanılır.

ADAPTER TASARIM DESENİ

Nasıl Kullanılır?

- Var olan sistemin interface'i, target interface olarak adlandırılır. Bu interface'i implement edecek bir **Adapter** sınıfı yaratılır.

- **Adapter** sınıfında, **Adaptee** interface türünden bir **sınıf değişkeni** bulunur. Son olarak client sınıfı **Adapter** sınıfı nesnesi ve **Adaptee** nesnesini yaratır.

ADAPTER TASARIM DESENİ

Nasıl Kullanılır?

- Daha iyi anlamak için şöyle bir **örnek** verelim: Matematik işlemleri yapan bir modülümüz olsun. Bu modüle String işlemleri yapan bir modül eklemek istiyoruz.

- String modülü içerisinde spesifik işlemleri yapan sınıflar bulunacak. Bu işlemi yaparken var olan modülün değiştirilmemesi gerekir.

- Örneğimizdeki var olan modülü **Target** interface temsil etmektedir. String modülünü **Adaptee** interface'i, bu ikisi arasındaki ilişkiyi ise **Adapter** sınıfı sağlamaktadır. String modülü içerisindeki sınıflar ise **Adaptee** interface implement eden sınıflardır.

ADAPTER TASARIM DESENİ

Faydaları Nedir?

1. Birbiriyle ilişkili olmayan interface'lerin birlikte çalışmasını sağlar.

2. Kodların yeniden yazılmasını engeller.

3. Var olan modül(ler) değiştirilmeden sisteme yeni modüller eklenebilir

ADAPTER TASARIM DESENİ

- Adapter tasarım deseninde mevcut bulunan yapıya uymayan bir sınıfı entegre etme tekniği incelenecektir.
- Yani çalışan sisteme yeni bir özelliğin eklenmesini kolaylaştıran bir tekniktir.
- Bunu yapmak için ayrı bir adaptör sınıfı yazılır ve bu sınıftan adapte edilmeye çalışılan sınıfa bir referans vardır. Bu sayede yazılmış olan adaptör sınıfı mevcut yapıya entegre bir şekilde çalışabilir.
- Aslında yapılan iş arka tarafta sisteme yeni bir arayüz kazandırmaktan farklı bir şey değildir.

ADAPTER TASARIM DESENİ

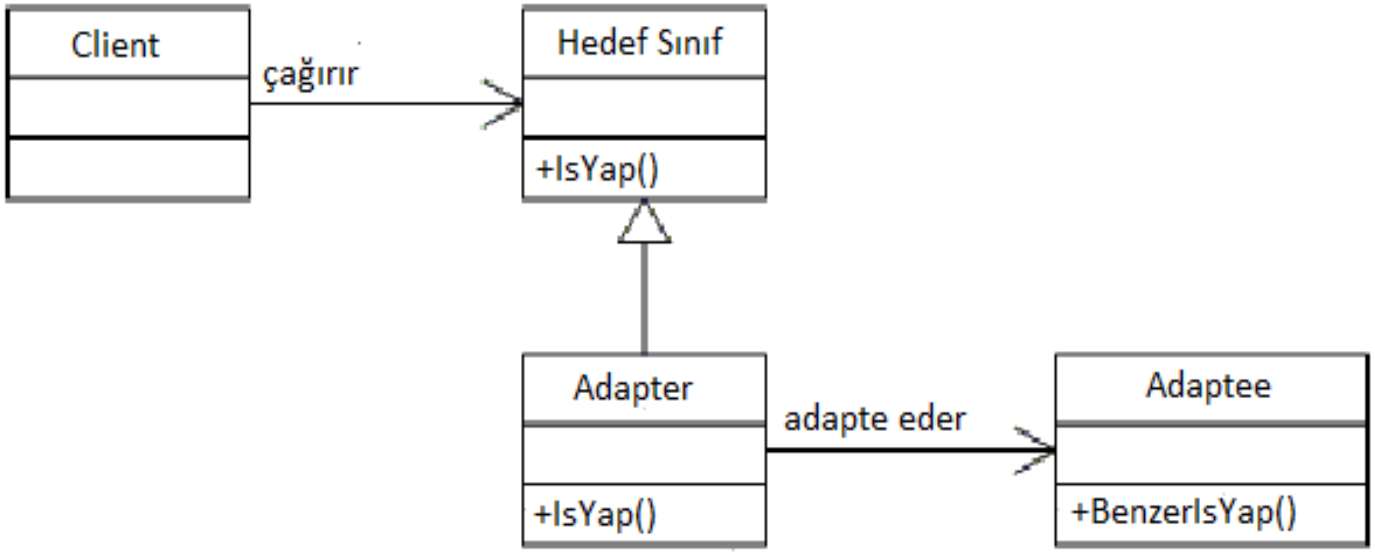
- Bu sebeple facade tasarım deseniyle ortak noktaları vardır. İkisinin ortak özelliği farklı bir arayüz sunmaktır.

- Ancak birinin amacı sistemi basitleştirmek, kod kalabalığını engellemek iken; adaptörün asıl hedefi sisteme yeni bileşenleri entegre edebilmektir.

- Entegre olunması hedeflenen bir arayüz vardır. Bu arayüzü gerçekleyen ve normalde gerçek adapte edilecek sınıfı kullanmaktan başka bir iş yapmayan bir adapter sınıf yazılır. Bu sınıf içerisinde sistemin içerisine dahil edilmesi istenen sınıf kullanılır.

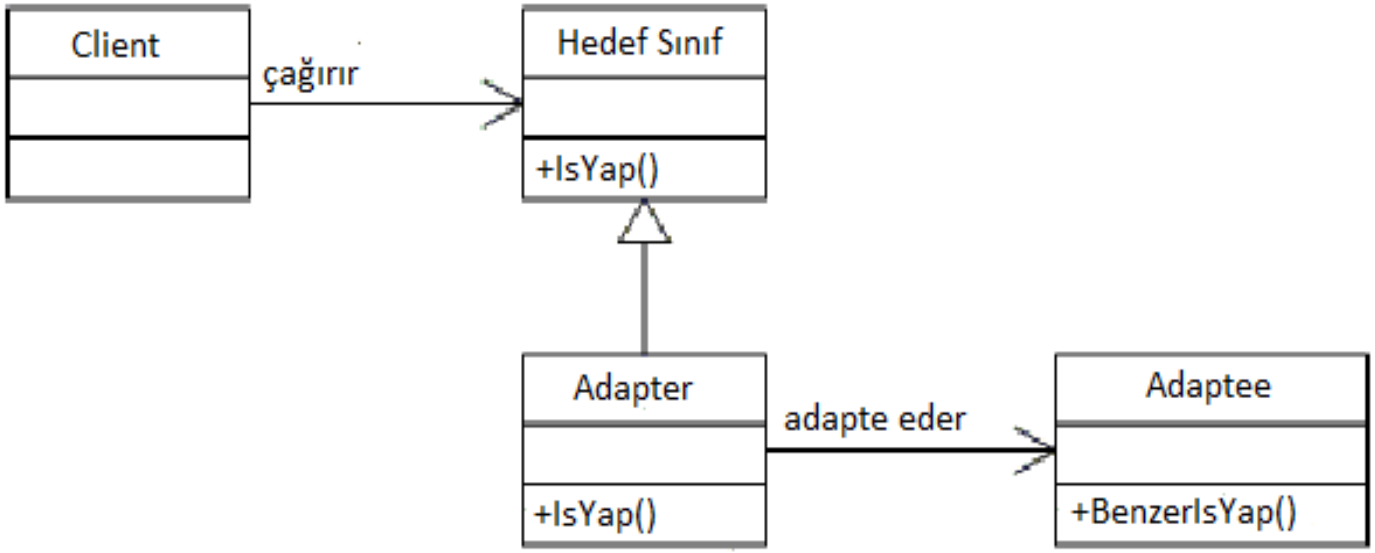
ADAPTER TASARIM DESENİ

- Entegre olunması hedeflenen bir arayüz vardır. Bu arayüzü gerçekleyen ve normalde gerçek adapte edilecek sınıfı kullanmaktan başka bir iş yapmayan bir adapter sınıf yazılır. Bu sınıf içerisinde sistemin içerisine dahil edilmesi istenen sınıf kullanılır.

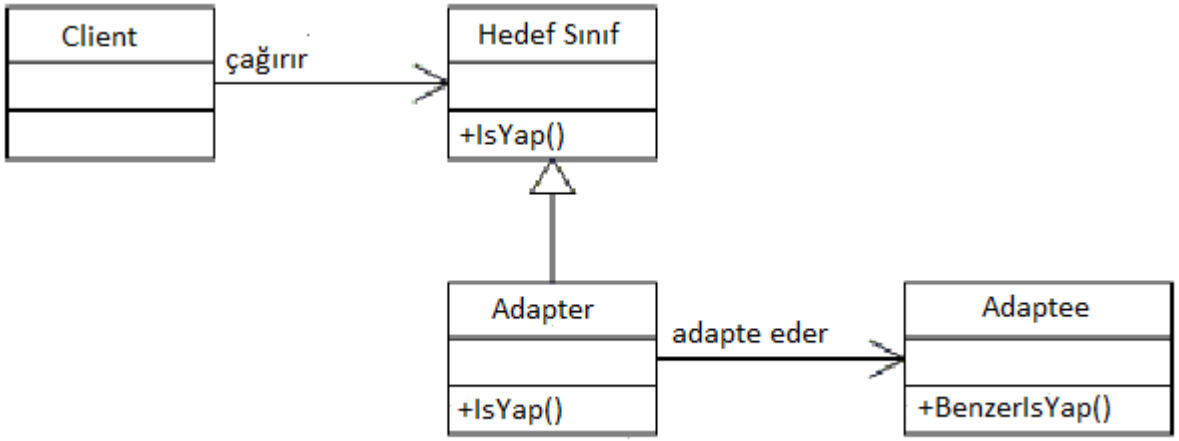


ADAPTER TASARIM DESENİ

- Adaptör kelimesinin günlük hayattaki karşılığı da benzer bir işlevi yerine getirmektedir. Bilgisayarlarımızın adaptörleri vardır. Bilgisayarın istemiş olduğu elektrik voltajı bellidir. Ancak priz o voltajdan çok daha yükseğini verir. İşte adaptörü kullanarak çıkış voltajını/gerilimini istenen seviyeye çekme işlemi gerçekleştirilir.

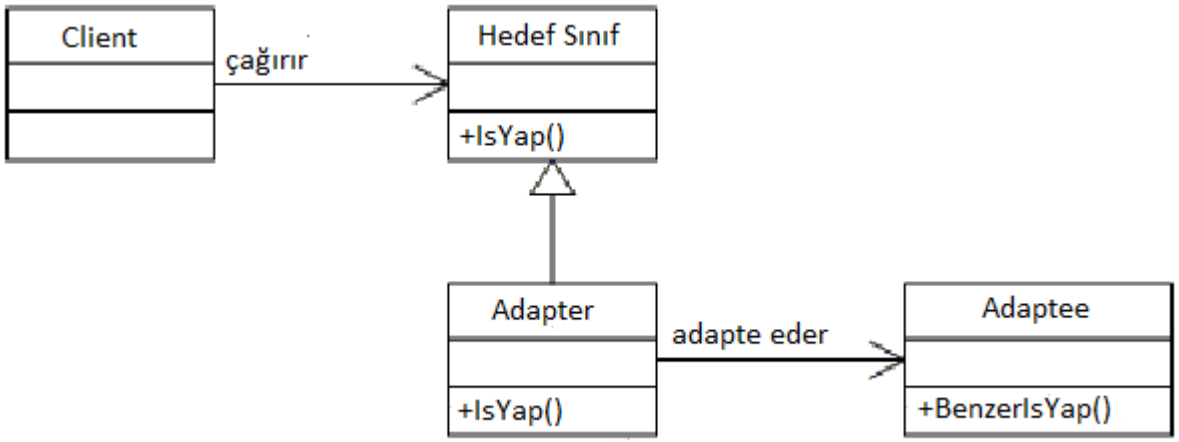


ADAPTER TASARIM DESENİ



- Hedef Sınıf(Target): Adapte olunması gereken sistem arayüzü.
- Adapter: Mevcut sistem arayüzünü gerçekleyen ara sınıf.
- Adaptee: Sisteme entegre çalışması istenen parça sınıf
- Client: Sistemi kullanan sınıf. Bu sınıf arkada olup biten bu işlemleri bilmez.

ADAPTER TASARIM DESENİ



- Yukarıdaki şemada "**Client**" tarafındaki kod biriminin **soyut olan** (interface veya abstract) "**Hedef**" kod birimini tanıması ve "Hedef" tipinden türetilen diğer tipleri tanımaması modülerliğin ve test edilebilirliğin sağlanabilmesi için gerekli esnekliği bize sunmaktadır.

- "Adapter" tarafındaki kod birimleri, "Hedef" tipinden türetilmektedir. Bu noktadaki implementasyonları N sayıda geliştirici yapabilmelidir.

ADAPTER TASARIM DESENİ

- Orta ve büyük ölçekli projeleri iş hayatında genelde tek bir kişi yapmaz. Bazı durumlarda birden fazla ekip bile aynı projede görev alabilir.
- Diğer yazılımcılar kendi sistemlerine çoğu zaman daha önceden yazılmış parçaları entegre etmek isterler.
- Bu tarz bir senaryoda adaptör tasarım desenine ihtiyaç duyarlar.
- Bu durum aynı şekilde parayla satın alınan veya açık kaynak kodlu bir üçüncü parti bileşen veya framework de olabilir.
- Bu tarz durumlarda da sisteme entegre etmek için adaptör tasarım deseni kullanılır.

ADAPTER TASARIM DESENİ

- Sisteme yeni bir log kütüphanesi eklendiğinde, sisteme yeni bir veri erişim kaynağı eklendiğinde sisteme yeni bir şifreleme algoritması entegre edildiğinde, sistemin genişletilmesi için yapılan yeni tedarikçi firma eklenmesi gibi senaryolarda eklemek istediğimiz sistem bizim sistemimizin arayüzüne çoğu zaman uymaz.

- Bu tarz bir durumda mevcut kodu değiştiremeyiz. Kendi projemizin kodlarını da değiştirmemiz zaten söz konusu olmaz.

- Bu desendeki teknik sayesinde sistemimizin arayüzünü bozmadan kolayca dış sistem kaynaklarını ekleyebiliriz.

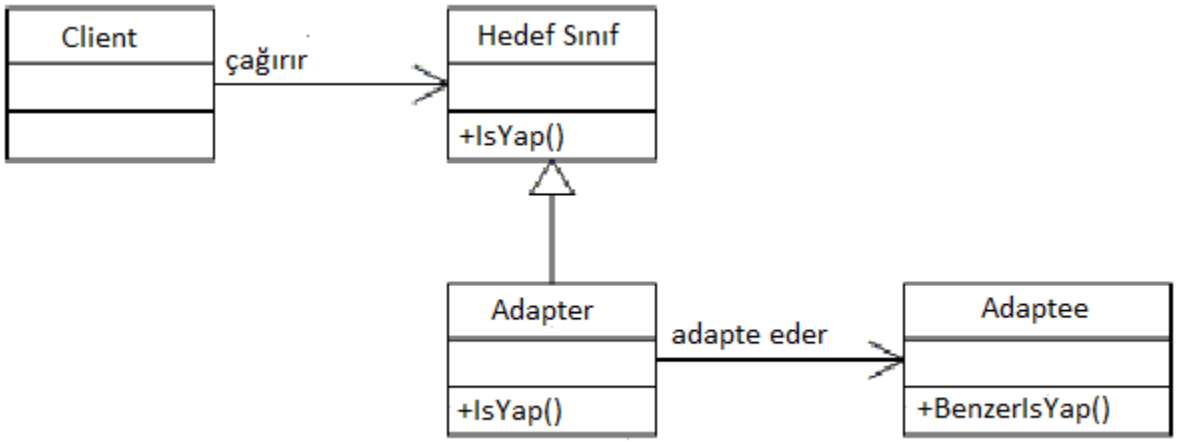
ADAPTER TASARIM DESENİ

- Adapter pattern adından da belli olduğu gibi var olan bir sistemi yeni sisteme adapte etme amacı güdülerek kullanılan bir patterndir.

- Yani bu pattern, hali hazırda yazılmış olup belli işlevleri gören kodların tekrar yazılmadan, yeni sisteme uyarlanıp entegre edilerek kullanılmasını sağlar.

ADAPTER TASARIM DESENİ

- Diyelim ki, önceden yazılmış iskontolu tutarı hesaplayan bir tane sınıfımız olsun. Bu sınıfımızı tutar hesaplama uygulamamıza entegre etmeye çalışalım.



ADAPTER TASARIM DESENİ

- Adaptee sınıfı içinde (yeni sisteme adapte edilecek sınıf) kullanıcıdan alınan fiyat, miktar ve iskonto bilgileri ile toplam tutarın hesaplanması yapılmaktadır.

Sisteme Adapte Edilecek Sınıf (Adaptee sınıfı)

```
1  public class Adaptee
2  {
3      public double IskontaliTutariHesapla(double fiyat, double adet)
4      {
5          Console.WriteLine("\nLütfen iskonto miktarını giriniz.\n");
6          double iskonto = Convert.ToDouble(Console.ReadLine());
7          return fiyat * adet * (1 - iskonto);
8      }
9  }
```

ADAPTER TASARIM DESENİ

- Adaptör nesnemizin implement edeceği ITutarHeaplayici interface'i data tipi double olan ve 2 tane parametre alan bir fonksiyon imzası taşımaktadır.

ITutarHesaplayici

```
1  public interface ITutarHesaplayici
2      {
3          double Hesapla(double fiyat, double adet);
4      }
```

ADAPTER TASARIM DESENİ

- Adaptör sınıfında private Adaptee tipinde değişken bulunmaktadır. Adapter sınıfının içinde ITutarHesaplayici interfacinin Hesapla fonksiyonunu implement ederken, bu fonksiyon içinde Adaptee tipindeki değişkenin IskontaliTutariHesapla fonksiyonunu çağırarak adapte edeceğimiz sınıfın örneğini kullanmış bulunmaktayız.

Adaptör Sınıfı

```
01 public class Adapter:ITutarHesaplayici
02 {
03     private Adaptee adaptee;
04
05     public Adapter()
06     {
07         adaptee = new Adaptee();
08     }
09
10     public double Hesapla(double fiyat, double adet)
11     {
12         return adaptee.IskontaliTutariHesapla(fiyat, adet);
13     }
14 }
```

ADAPTER TASARIM DESENİ

- Bu sınıfın içindeki OdenecekMeblayiHesapla fonksiyonu kullanıcıdan fiyat ve miktar bilgilerini aldıktan sonra ITutarHesaplayici interfacini implement etmiş sınıfların nesnelerini parametre olarak alarak hesaplama işlemini yapmaktadır.

Client Sınıfı

```
01 public class Client
02 {
03     public void OdenecekMeblayiHesapla(ITutarHesaplayici hesaplayici)
04     {
05         Console.WriteLine("\nLütfen fiyatı giriniz.\n");
06         double fiyat = Convert.ToDouble(Console.ReadLine());
07         Console.WriteLine("\nLütfen miktarı giriniz.\n");
08         double miktar = Convert.ToDouble(Console.ReadLine());
09         Console.WriteLine("\nTutar = " + hesaplayici.Hesapla(fiyat, miktar));
10         Console.WriteLine();
11     }
12 }
```

ADAPTER TASARIM DESENİ

Main fonksiyonu içinde kullanıcıdan hangi tür hesaplama işlemini yapacağı bilgisini aldıktan sonra, client nesnesi oluşturarak içine ilgili nesneleri parametre olarak geçiyoruz. Adapter ve MalzemeTipineGoreToplamTutariniHesapla nesnelerini tutar hesaplamak için kullanıyoruz. MalzemeTipineGoreToplamTutariniHesapla nesnesini kendimiz yazdığımız halde Adapter nesnesi sayesinde önceden yazılmış olan Adaptee nesnesini sisteme adapte ederek kullanıyoruz.

Program.cs

```
01 class Program
02 {
03     static void Main(string[] args)
04     {
05         Adapter adapter = new Adapter();
06         string tercih = null;
07         Client client = new Client();
08         try
09         {
10             while (true)
11             {
12                 Console.WriteLine("Malzemenin tutarını hesaplamak için 1 e," +
13                     "\niskontolu tutarını hesaplamak için 2 ye," +
14                     "\nuygulamadan çıkmak için 3'e basınız.\n");
15                 tercih = Console.ReadLine();
16                 int secenek = Convert.ToInt32(tercih);
17                 Console.WriteLine();
18 
19                 if (secenek == 1)
20                 {
21                     client.OdenecekMeblayiHesapla(new MalzemeTipineGoreToplamTutariniHesapla());
22                 }
23                 else if (secenek == 2)
24                 {
25                     client.OdenecekMeblayiHesapla(new Adapter());
26                 }
27                 else
28                 {
29                     return;
30                 }
31             }
32         }
33         catch
34         {
35             Console.WriteLine("Hata ile karşılaşıldı. Uygulama sonlanacaktır.");
36             Thread.Sleep(1500);
37         }
38     }
39 }
```

ADAPTER TASARIM DESENİ

ITutarHesapla arayüzünü uygulayan MalzemeTipineGoreToplamTutariniHesapla sınıfını oluşturunuz. Bu sınıf altında Hesapla metodunu normal tutarı hesaplayacak şekilde tasarlayınız.

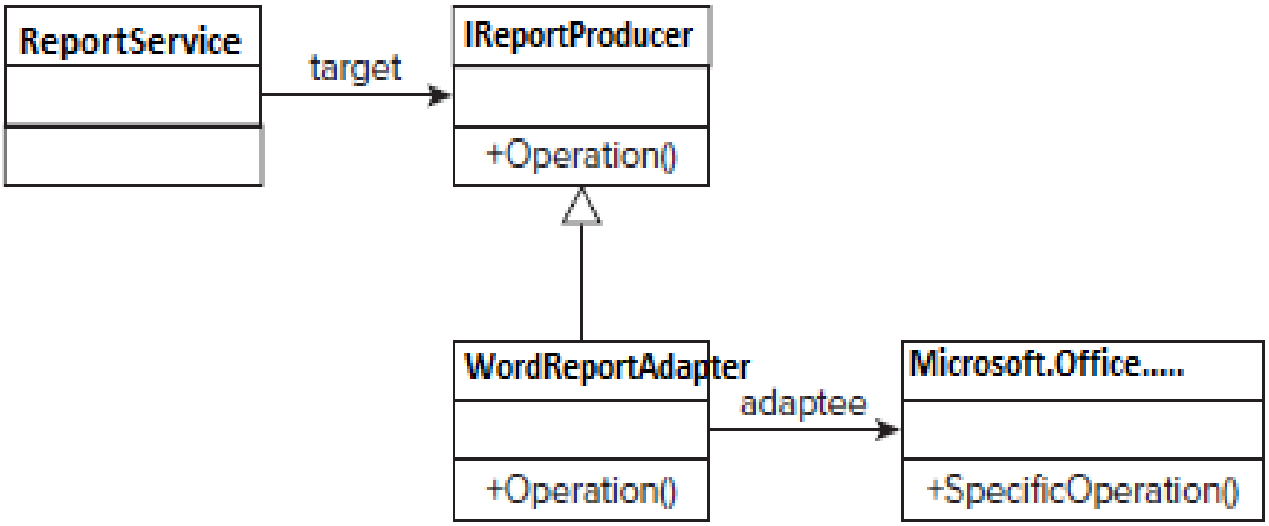
Program.cs

```
01 class Program
02 {
03     static void Main(string[] args)
04     {
05         Adapter adapter = new Adapter();
06         string tercih = null;
07         Client client = new Client();
08         try
09         {
10             while (true)
11             {
12                 Console.WriteLine("Malzemenin tutarını hesaplamak için 1 e," +
13                     "\niskontolu tutarını hesaplamak için 2 ye," +
14                     "\nuygulamadan çıkmak için 3'e basınız.\n");
15                 tercih = Console.ReadLine();
16                 int secenek = Convert.ToInt32(tercih);
17                 Console.WriteLine();
18
19                 if (secenek == 1)
20                 {
21                     client.OdenecekMeblayiHesapla(new MalzemeTipineGoreToplamTutariniHesapla());
22                 }
23                 else if (secenek == 2)
24                 {
25                     client.OdenecekMeblayiHesapla(new Adapter());
26                 }
27                 else
28                 {
29                     return;
30                 }
31             }
32         }
33         catch
34         {
35             Console.WriteLine("Hata ile karşılaşıldı. Uygulama sonlanacaktır.");
36             Thread.Sleep(1500);
37         }
38     }
39 }
```

ADAPTER TASARIM DESENİ

- Adapter tasarım deseni, kodun bağımlılığını azaltmak amacıyla uygulanan kalıplardan biridir.
- Özellikle kurumsal bazlı projelerde modüler yapıda geliştirme yapıldığı düşünüldüğünde uyumluluğun sağlanması, gerekli şartlardan sadece biridir.
- Proje üzerinde çalışan geliştiriciler farklı modülleri hazırlayıp ortak bir noktada yazılıma monte edebilmelidir.

ADAPTER TASARIM DESENİ



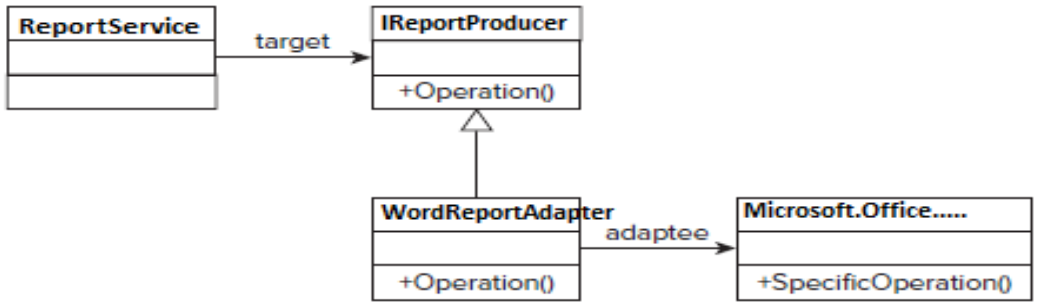
- Bu şemada "**Client**" kod olarak bir rapor servisi (**ReportService**) sunulmaktadır.
- **ReportService** servis tipimiz sadece soyut olan "**IReportProducer**" tipini tanır.
- **IReportService** tipinin uygulandığı **WordReportAdapter** tipini tanımaz.
- **WordReportAdapter** tipi sadece **Microsoft.Office** kütüphanesini kullanarak Word dökümanı üretir.

ADAPTER TASARIM DESENİ

- İki farklı geliştirici olduğu düşünülürse; birine Word dokümanı üreten adaptör, diğerine de PDF dokümanı üreten adaptör yazdırılabilir.
- Geliştiriciler de ReportService tipiyle ilgilenmezler. Sadece IreportProducer tipini tanırlar ve bu tipi implemente ederek PDF ve Word raporu üreten sınıfları geliştirirler.
- Eğer Word doküman raporunu üreten kod ReportService sınıfı içerisinde yapılsaydı;
  - Test edilebilirlikten uzaklaşılacaktı.
  - Birden fazla developer ile bir iş geliştirilemeyecekti.
  - Rapor servis tipi somut nesnelere bağımlı olacaktı.
  - Yeni bir rapor üretici geliştirilmek istendiğinde servis sınıfını bozmak zorunda kalınacaktı.

ADAPTER TASARIM DESENİ

```
public interface IReportProducer
{
    void CreateReport(Report report);
}
public class ReportService
{
      private readonly IReportProducer reportProducer;
      public ReportService(IReportProducer reportProducer)

          this.reportProducer = reportProducer;
      }
     public void CreateReport()
      {
          reportProducer.CreateReport(new Report());
      }
}
public class WordReportAdapter : IReportProducer
{
      public void CreateReport(Report report)
      {
          // TR: Office DLL nesneleri yardımıyla raporu oluştur.
          // EN: Get Office DLL objects and create Report
      }
}
```



ADAPTER TASARIM DESENİ

- Bu tasarım sayesinde ReportService sınıfı test edilebilir hale getirilmiş oldu.
- Eğer Word döküman raporunu üreten kodu ReportService sınıfı içerisinde yapsaydık;
  - Test edilebilirlikten uzaklaşacaktık.
  - Birden fazla developer ile bir işi geliştiremeyecektik.
  - Rapor servis tipi somut nesnelere bağımlı olacaktı.
  - Yeni bir rapor üretici geliştirilmek istediğimizde servis sınıfını bozmak zorunda kalacaktık.
- Adapter tasarım kalıbının uygulanması gayet basittir. Tek amacı ise birbirini tanımayan tipleri interface gibi soyut tipler kullanarak birbiri ile çalışabilir hale getirmektir.

ADAPTER TASARIM DESENİ

Soru:

- Varsayalım ki bir Müzik mağazamız var ve sadece enstruman olarak gitar satışı yapıyoruz.
- Ancak sonradan mağazamızda yeni bir enstrüman satmaya karar verdik.
- İşte bu noktada sadece gitar siparişi verebildiğimiz class'ımıza Kemence siparişimizi de adapte ediyoruz.

ADAPTER TASARIM DESENİ

ÖRNEK:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Desing_Pattern___Adapter
{

    // Varsayalım ki bir Müzik mağazamız var ve sadece enstruman olarak gitar
satışı yapıyoruz.

    class Target
    {
        public virtual void Siparis()
        {
            Console.WriteLine("Gitar Siparişi başarı ile verilmiştir...");
        }
    }
```

ADAPTER TASARIM DESENİ

```
    // Ancak sonradan mağazamızda yeni bir enstruman satmaya karar verdik.

    class Adaptee
    {
        public void KemenceSiparisi()
        {
            Console.WriteLine("Kemençe Siparişi başarı ile verilmiştir");
        }
    }
    // İşte bu noktada sadece gitar siparişi verebildiğimiz classımıza Kemence
siparişimizide adapte ediyoruz.Bu yüzden Target classını

    class Adapter : Target
    {
        private Adaptee _adaptee = new Adaptee();

        public override void Siparis()
        {
            _adaptee.KemenceSiparisi();
        }
    }
```

ADAPTER TASARIM DESENİ

```
    // Tanımladığımız Adapteri denemek için clientte testimizi
yapıyoruz.

    // Client
    class MainApp
    {
        static void Main()
        {
            Target target = new Adapter();
            target.Siparis();
            Console.ReadKey();
        }
    }
  }
```

ADAPTER TASARIM DESENİ

ÖRNEK:

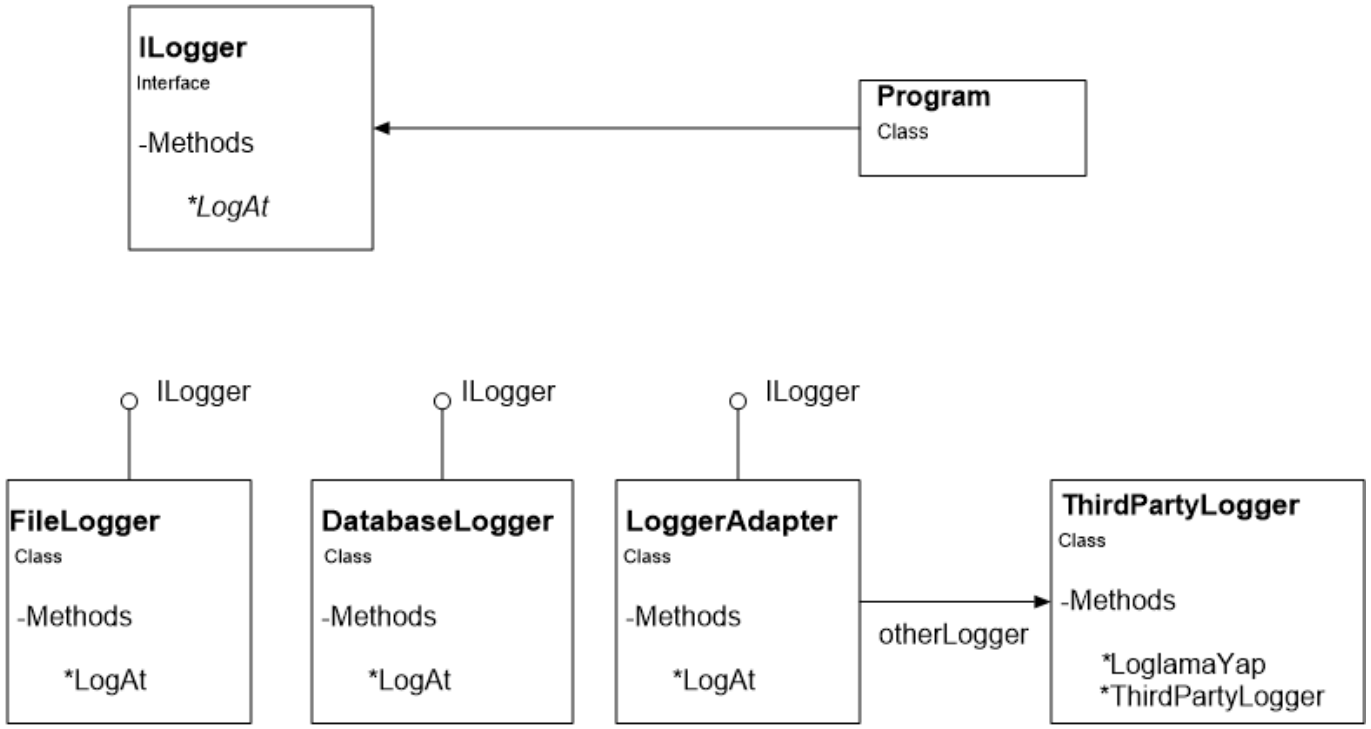
Bir uygulamada IO işlemleri için bir yapının var olduğu düşünülsün. IIO arayüzünü uygulayan sınıflar üzerinden IO işlemlerinin gerçekleştirildiği varsayılsın. Fakat bir gün IO işlemlerini daha hızlı yapan başka bir kod edinildi ve bu kodun mevcut IO yapısına adapte edilmesine karar verildi. Tek yapılması gereken bir IIO arayüzünden türeyen ve içinde adapte edilmek istenen kütüphaneyi kullanacak olan yeni bir Adapter sınıfı yazmak olacak.

Örnek uygulamanın class diyagramı ve kodları aşağıdadır.

ADAPTER TASARIM DESENİ

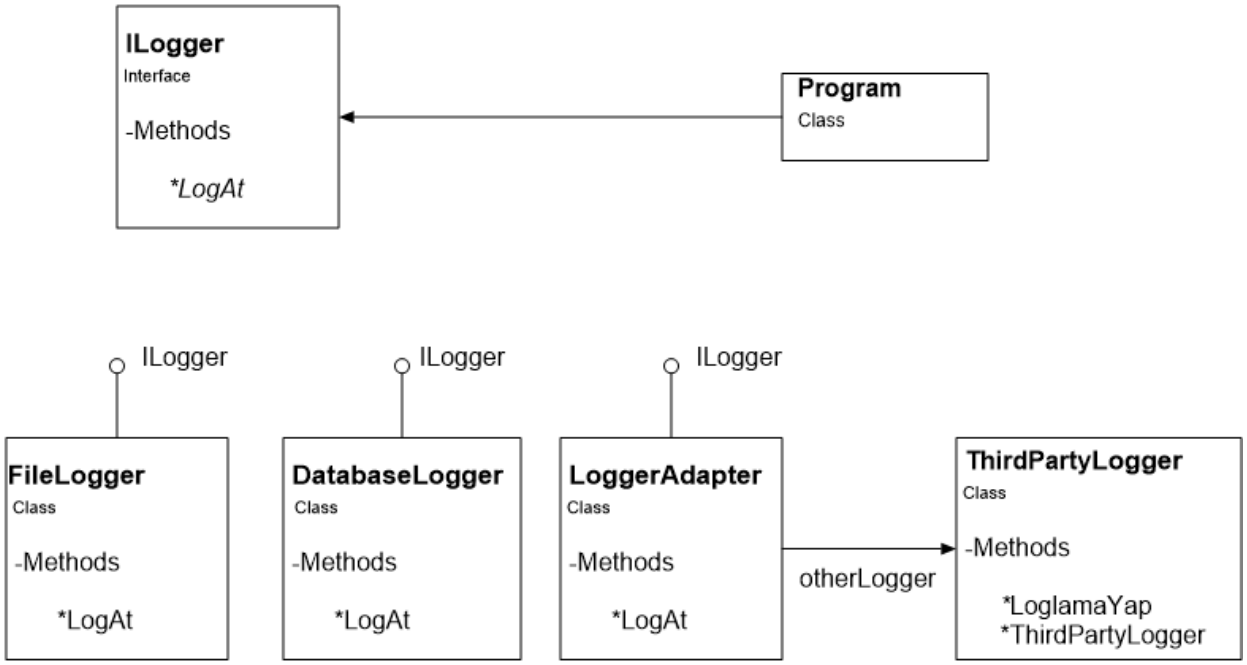
ÖRNEK:

Hali hazırda mevcut olan bir sistemde zaten bir log sisteminin var olduğu düşünülsün. Sisteme yeni satın alınan bir loglama mekanizmasının entegre edilmesi istenmektedir. Bu tarz bir durumu modelleyen diyagram aşağıdaki gibi olacaktır.



ADAPTER TASARIM DESENİ

- İlgili sistemde entegre edilmeye çalışılan arayüz, Ilogger sınıfıdır.
- Database ve File olmak üzere iki farklı log seçeneği vardır.
- Üçüncü parti bir loglama bileşeni satın alındığını düşünülürse. Onu sisteme entegre etmek için ayrı bir adapter sınıfı yazılır. Bu adapter sınıfı da kendi içerisinde ThirdPartyLogger sınıfını kullanır.



ADAPTER TASARIM DESENİ

```
namespace Adapter.OurLogSystem
{
    public interface ILogger
    {
        void LogAt(string loglanacakVeri);
    }
}
namespace Adapter.OurLogSystem
{
    public class FileLogger:ILogger
    {
        public void LogAt(string loglanacakVeri)
        {
            //burada dosyaya log atıldığını düşünün
        }
    }
}
namespace Adapter.OurLogSystem
{
    public class DatabaseLogger:ILogger
    {
        public void LogAt(string loglanacakVeri)
        {
            //burada ilgili verinin db ye kaydedildiğini düşünüyoruz.
        }
    }
}
```

ADAPTER TASARIM DESENİ

- Mevcut sistemde taslak olarak yapının yukarıdaki kod gibi olduğu düşünülsün. Şimdi de entegre edilmeye çalışılan sistemin log mekanizmasının kodlarını inceleyiniz.

```
namespace Adapter._3rdPartyLogSystem
{
    public class ThirdPartyLogger
    {
        public void LoglamaYap(LogItem logItem)
        {
            //burada herhangi ilgili sınıftaki veriler düzenlenerek
            //log atılıyor.
        }
    }
}
```

ADAPTER TASARIM DESENİ

- Görüldüğü gibi önceki arayüzü destekleyen bir yapıda değil. Bunu önceki sisteme entegre etmek için gerekli adapter sınıfı şöyledir:

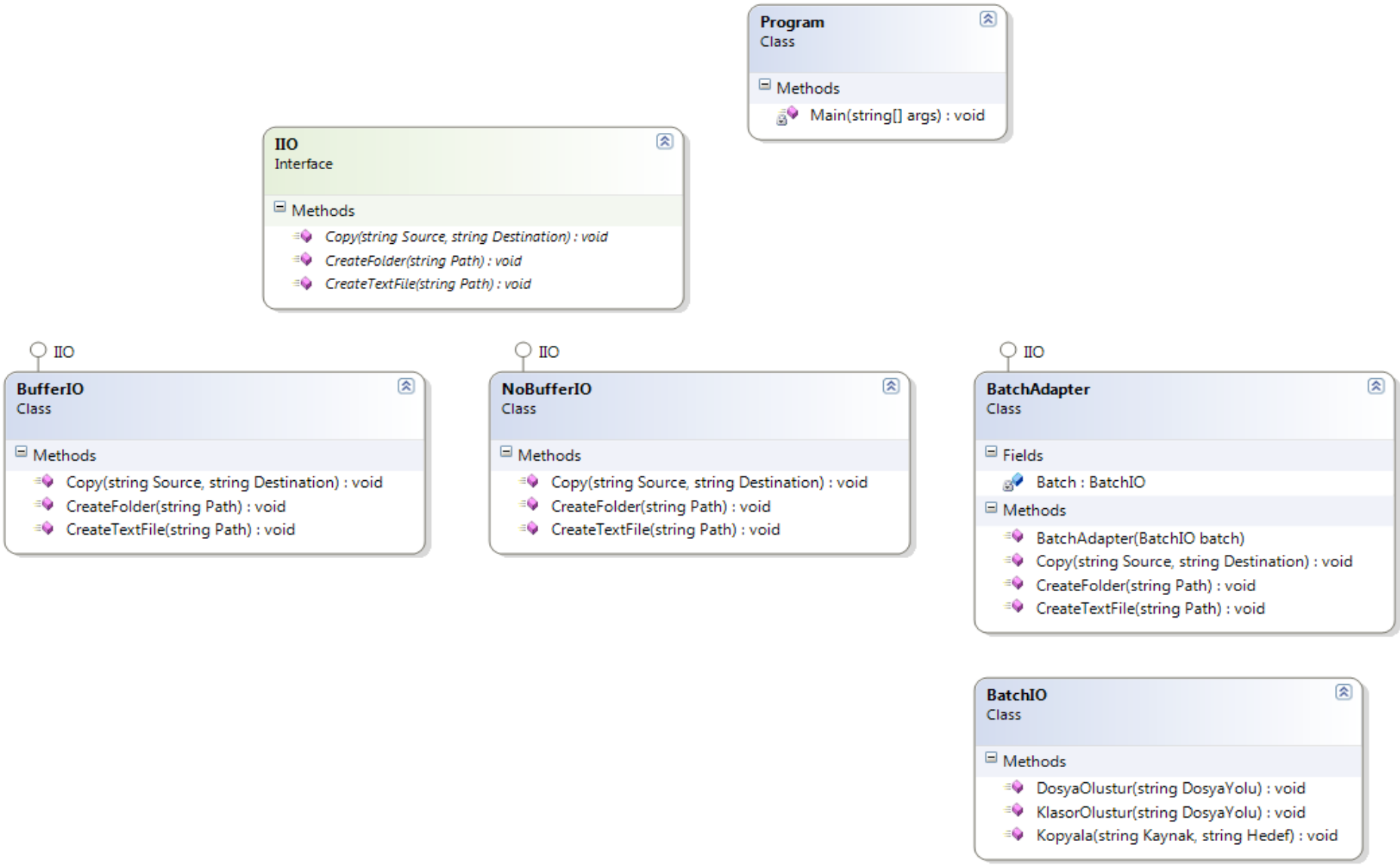
```
namespace Adapter.OurLogSystem
{
    public class LoggerAdapter:ILogger
    {
        private ThirdPartyLogger otherLogger;
        public void LogAt(string loglanacakVeri)
        {
            LogItem logItem = new LogItem();
            logItem.LogId = new Guid();
            logItem.LogDescription = loglanacakVeri;
            otherLogger = new ThirdPartyLogger();
            otherLogger.LoglamaYap(logItem);
        }
    }
}
```

ADAPTER TASARIM DESENİ

- Yazılan yeni sınıf, Ilogger arayüzünden türeyerek önceki sisteme dahil olur, ancak içerisindeki işleri aslında üçüncü parti log mekanizmasının sınıfıyla halleder. Bu sayede client kodunda ekstra referans eklemeye veya yapı güncellemesine gerek kalmaz. Zaten bu şekilde olsaydı, bu durum açık kapalı prensibine de uymayacaktı.

- Bu tarz problemlerde kod aynen kopyala yapıştır olarak da alınabilirdi. Önceki sistemin desteklediği arayüzü yeni üçüncü parti bileşene göre güncelleyebilirdik. Bunlar yapılırsa tasarım desenlerinin pek bir anlamı kalmayacaktır. Hedef hep daha az kod, daha çok iş, daha az değişiklik, daha hızlı genişleme.

ADAPTER TASARIM DESENİ



ADAPTER TASARIM DESENİ

```
//Target
interface IIO
{
    void CreateTextFile(string Path);
    void CreateFolder(string Path);
    void Copy(string Source, string Destination);
}

class NoBufferIO : IIO
{
    public void CreateTextFile(string Path)
    {
        Console.WriteLine("{0} CreateTextFile(NetIOClass)", Path);
    }
    public void CreateFolder(string Path)
    {
        Console.WriteLine("{0} CreateFolder(NetIOClass)", Path);
    }
    public void Copy(string Source, string Destination)
    {
        Console.WriteLine("{0}==>{1} Copy(NetIOClass)", Source, Destination);
    }
}
```

ADAPTER TASARIM DESENİ

```
class BufferIO : IIO
{
    public void CreateTextFile(string Path)
    {
        Console.WriteLine("{0} CreateTextFile(BufferIO)",
Path);
    }

    public void CreateFolder(string Path)
    {
        Console.WriteLine("{0} CreateFolder(BufferIO)", Path);
    }

    public void Copy(string Source, string Destination)
    {
        Console.WriteLine("{0}==>{1} Copy(BufferIO)", Source,
Destination);
    }
}
```

ADAPTER TASARIM DESENİ

```
//Adapter
class BatchAdapter : IIO
{
   /*Bu sınıfda Adaptee nesnesinin üzerinden Target de tanımlı işlemleri gerçekleştirip
     Adaptee nesnesini Target yapısına uyarlamış oluyoruz.    */

   private BatchIO Batch;
   public BatchAdapter(BatchIO batch)
   {
      Batch = batch;
   }
   public void CreateTextFile(string Path)
   {
      Batch.DosyaOlustur(Path);
   }
   public void CreateFolder(string Path)
   {
      Batch.KlasorOlustur(Path);
   }
   public void Copy(string Source, string Destination)
   {
      Batch.Kopyala(Source, Destination);
   }
}
```

ADAPTER TASARIM DESENİ

```
//Adaptee
class BatchIO
{
    public void DosyaOlustur(string DosyaYolu)
    {
        Console.WriteLine("{0} BatchIO DosyaOlusturuldu", DosyaYolu);
    }
    public void KlasorOlustur(string DosyaYolu)
    {
        Console.WriteLine("{0} BatchIO KlasorOlusturuldu", DosyaYolu);
    }
    public void Kopyala(string Kaynak, string Hedef)
    {
        Console.WriteLine("{0} ==> {1} BatchIO Kopyalandı.", Kaynak, Hedef);
    }
}
class Program
{
    static void Main(string[] args)
    {
        IIO io = new BatchAdapter(new BatchIO());
        io.CreateFolder(@"c:\a");
        Console.ReadKey();
    }
}
```

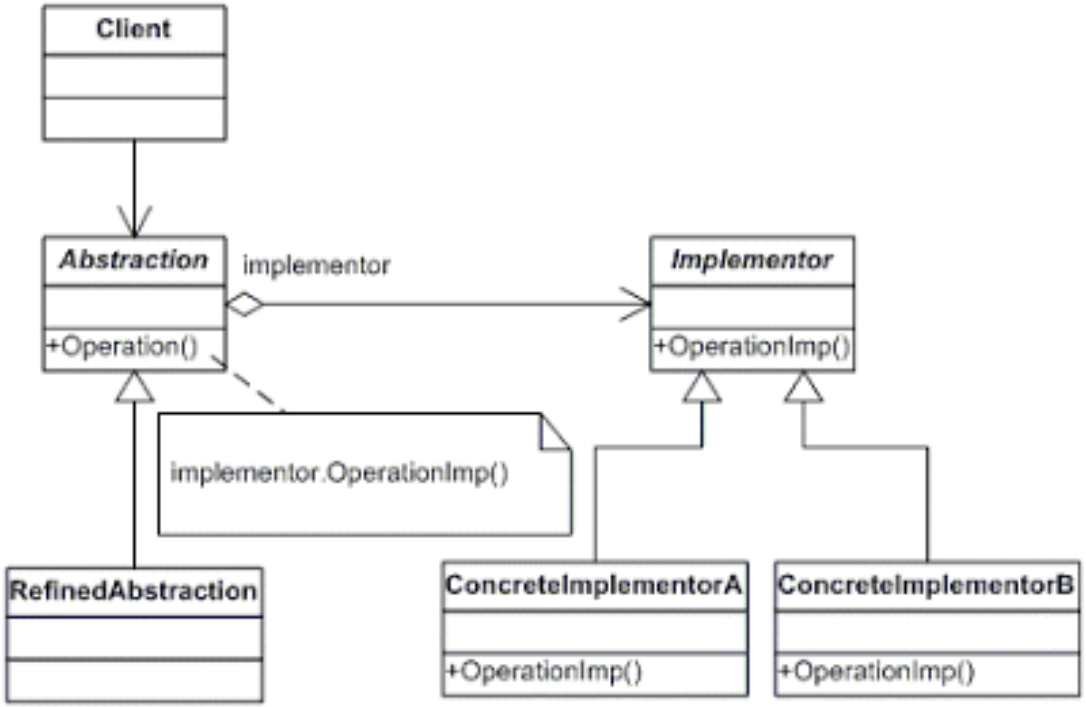
BRIDGE (KÖPRÜ) TASARIM DESENİ

BRIDGE (KÖPRÜ) TASARIM DESENİ

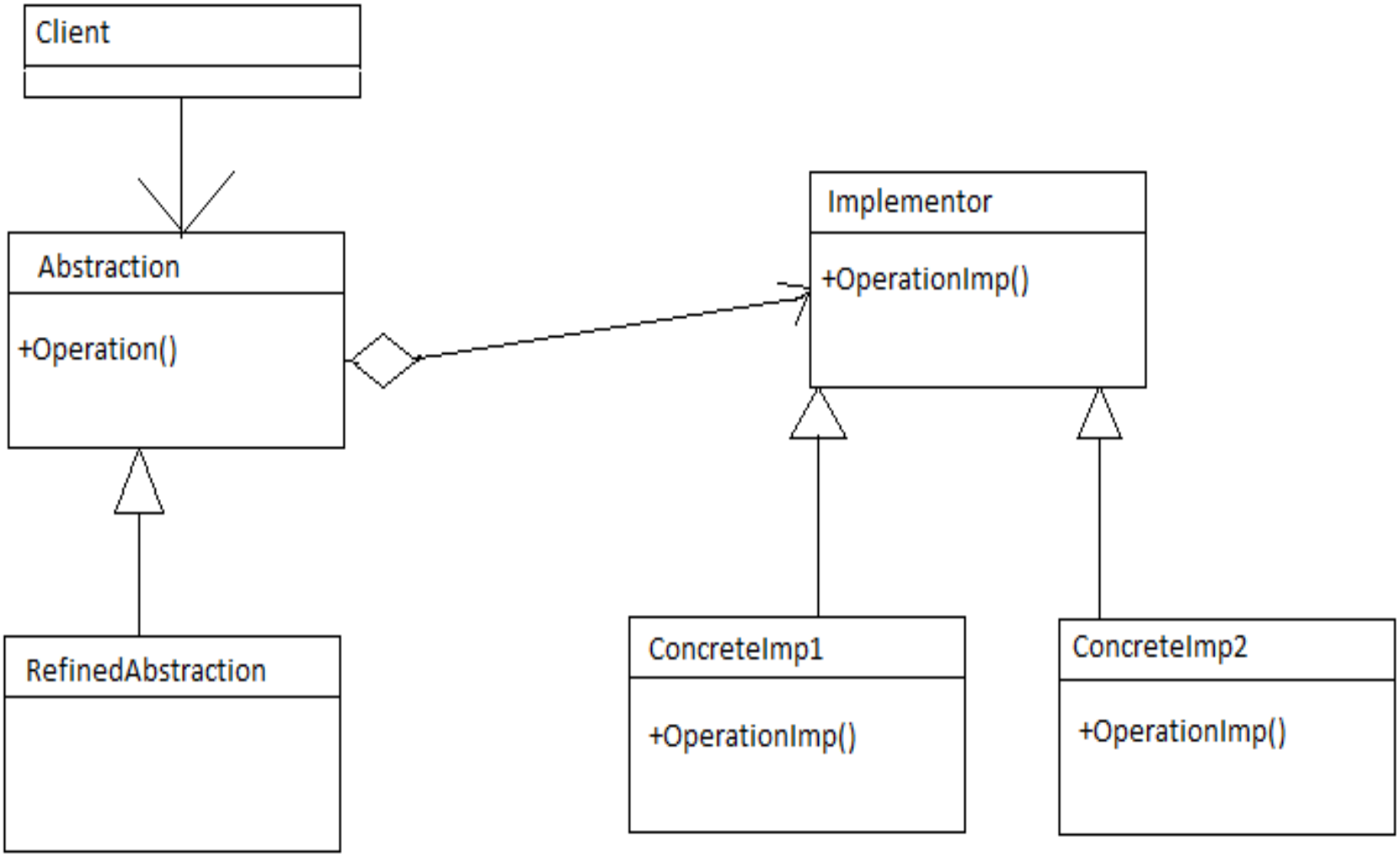
- Köprü tasarım deseni (Bridge Design Pattern), yapısal tasarım desenlerinden biridir.
- Soyutlanılan nesneler ile işi gerçekleyecek somut nesneler arasında köprü kurar.
- Soyut sınıflar ve iş yapacak sınıfları birbirinden ayırdığı için iki sınıf tipinde yapılacak bir değişiklik birbirlerini etkilemez.
- Hangi sınıfın kullanılacağına çalışma zamanında karar verilir.
- Bu mekanizma sayesinde çalışma anında, gerçek işi yapan sınıf değiştirilebilir.

BRIDGE (KÖPRÜ) TASARIM DESENİ

- Soyutlanan nesne ve implementasyonun birbirinden ayrılarak düzenlenmesine olanak sağlar.
- Nesnenin değişimi impelementasyonu, implementasyonun değişimi nesneyi etkilemez. Her ikisi bağımsız olarak geliştirilebilir ve türetilen tipleri üzerinden her türlü nesne-implementasyon kombinasyonu gerçekleşebilir.
- Tek bir implementasyonun farklı nesneler tarafından da kullanılmasına olanak sağlar.

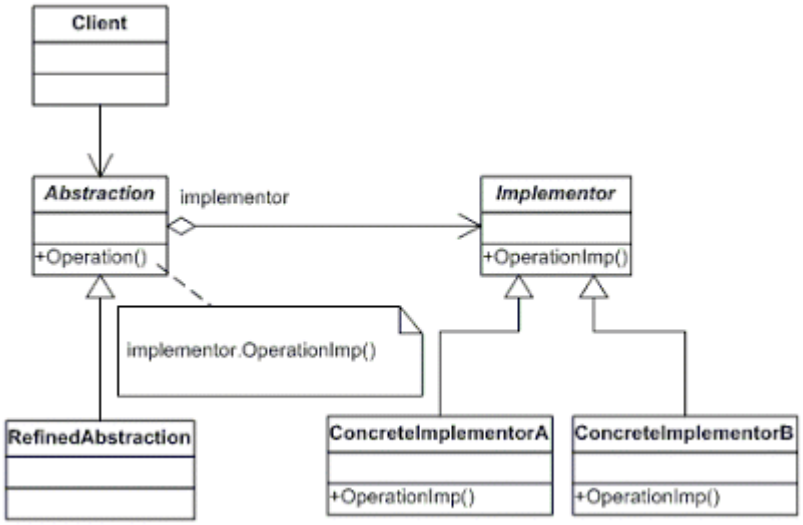


BRIDGE (KÖPRÜ) TASARIM DESENİ



BRIDGE (KÖPRÜ) TASARIM DESENİ

- Köprü tasarım desenini oluşturan elemanlar;
  - Client: İstemci sınıf.
  - Abstraction: Implementor(Uygulayıcı) sınıf arayüzü barındırır. Bu arayüz vasıtası ile Concrete Implementor (somut uygulayıcı) sınıf üzerindeki metotları çağırır.
  - RefinedAbstraction: Abstraction arayüzünü veya soyut sınıfını uygulayan gerçek sınıflardır.
  - ConcreteImplementor: Implementor arayüzünden türeyen, gerçek işi yapan sınıflardır.
- Köprü tasarım deseni bu beş yapıdan oluşmaktadır.
- İstemci taraf, Abstraction ve Implementor arayüzlerinden türemiş sınıflar oluşturarak işlemi gerçekleştirir.



Ne Zaman Kullanılır?

BRIDGE (KÖPRÜ) TASARIM DESENİ

- Köprü tasarım deseninin birden fazla kullanım alanı, yani problem çözümü vardır. İlk problem, çoklu sınıf hiyerarşileridir. Bazen iyi başlayan tasarımlar daha sonra gözden kaçırılan ufak detaylar yüzünden kötü bir hâl alabilir.

- Örneğin, bir raporlama servisi yazıldığı düşünülsün. Veri toplayan bir uygulama olsun ve bu verileri raporlayan bir uygulama yazılmış olsun. Bizim geliştireceğimiz kısım raporlama olacaktır.

- Raporlama uygulaması, ortak ürün ailelerinden oluşan veri modelleri için sp (stored procedur)'ler çağırıyor olsun. "Ortak ürün ailesi" cümlesinden dolayı tasarım abstract factory (Soyut Fabrika) üzerine kurulabilir. İlk bakışta doğru bir tasarım gibi duruyor. Fakat sistemi sorgulamaya devam edip birkaç soru sorulması gerekiyor. Kaç farklı ürün ailesi var? Ürün aileleri arasında hiyerarşi kaç kademeli?

BRIDGE (KÖPRÜ) TASARIM DESENİ

- Bu soruları sorma nedeni sistem kodlanmaya başlandığında, devasa alt alta kalıtım ile geçmiş sınıf modelleri ile karşılaşmamaktır.

- Böyle bir sonuç karşısında karışık hiyerarşiler ile çalışmak zorunda kalınabilir. Bunun sonucunda sistemi kontrol etmek zorlaşacaktır. Bazı alt sınıfların üzerinde olmaması gereken özellikler taşıdığı görülür.

- Hiyerarşinin büyük olacağı cevabı alındıktan sonra tasarım köprü tasarım desenine çevrilir. Çünkü köprü desen soyut sınıflar ile işi yapan sınıfları birbirinden ayıracağı için devasa factory ve ürün ailesi sınıfları oluşturmaya gerek kalmayacaktır.

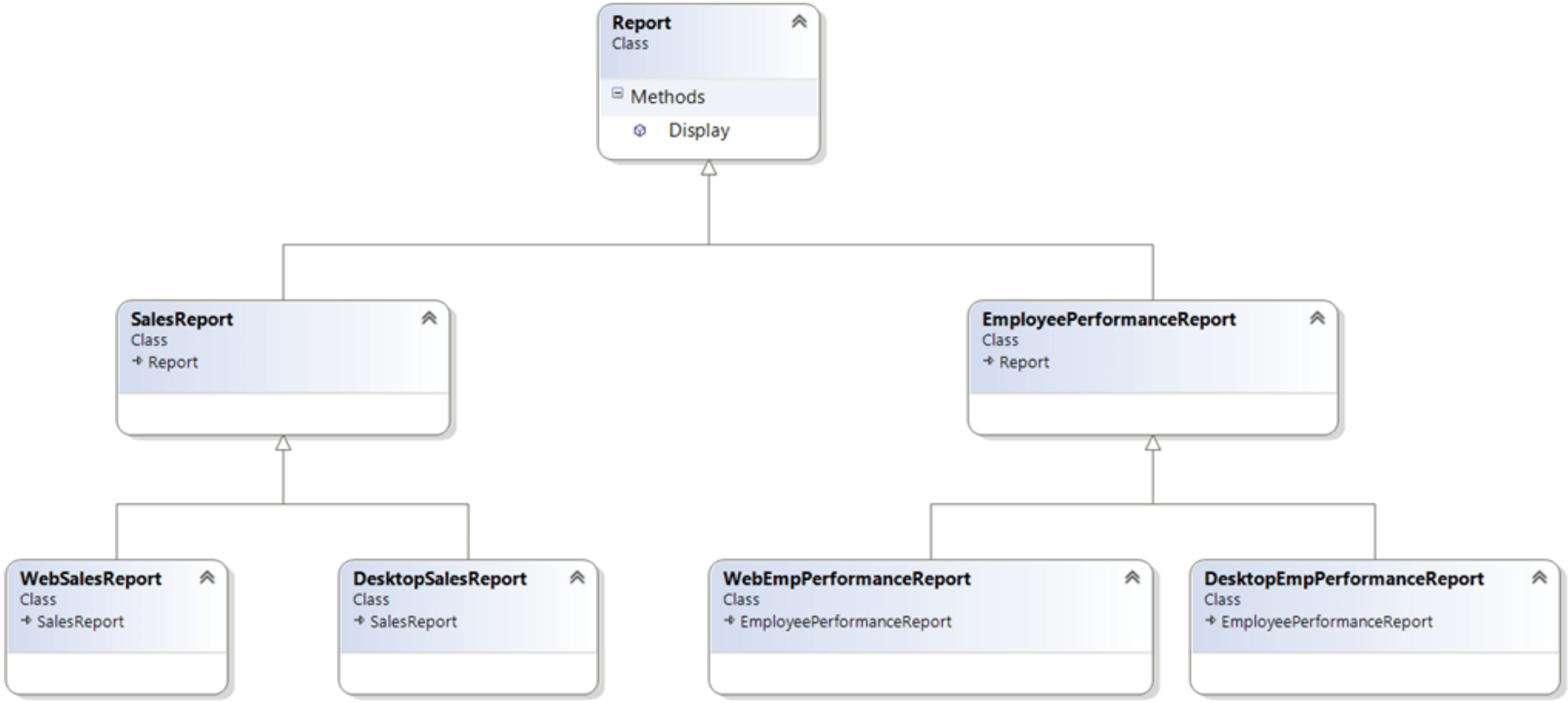
BRIDGE (KÖPRÜ) TASARIM DESENİ

- **Hiyerarşi dikey yönde değil yatay yönde büyüyecektir**.
- Bu yöntem ile büyük boyuttaki veri modellerinden kurtulmuş olunur.
- Köprü tasarım deseni için, soyutlanan nesneler ve işi yapacak gerçek nesneler arasında köprü kurar denmişti.
- Bu bilgiyi biraz irdeleyelim. Elde bir grup soyut sınıf ve işi yapacak sınıf var. Bu sınıflar birbirini tanımıyor ve köprü desen ile çalışma anında birleştiriliyor ve çalışıyorlar. Nasıl çalışacaklarına çalışma anında karar veriliyorsa, verilen kararı değiştirerek soyut sınıflar başka bir iş yapıcı sınıfa da yönlendirilebilir.
- Bu bilgi doğrultusunda, eğer sistem çalışma anında çalışacağı öğeleri değiştirecekse köprü deseni kullanılabilir.

BRIDGE (KÖPRÜ) TASARIM DESENİ

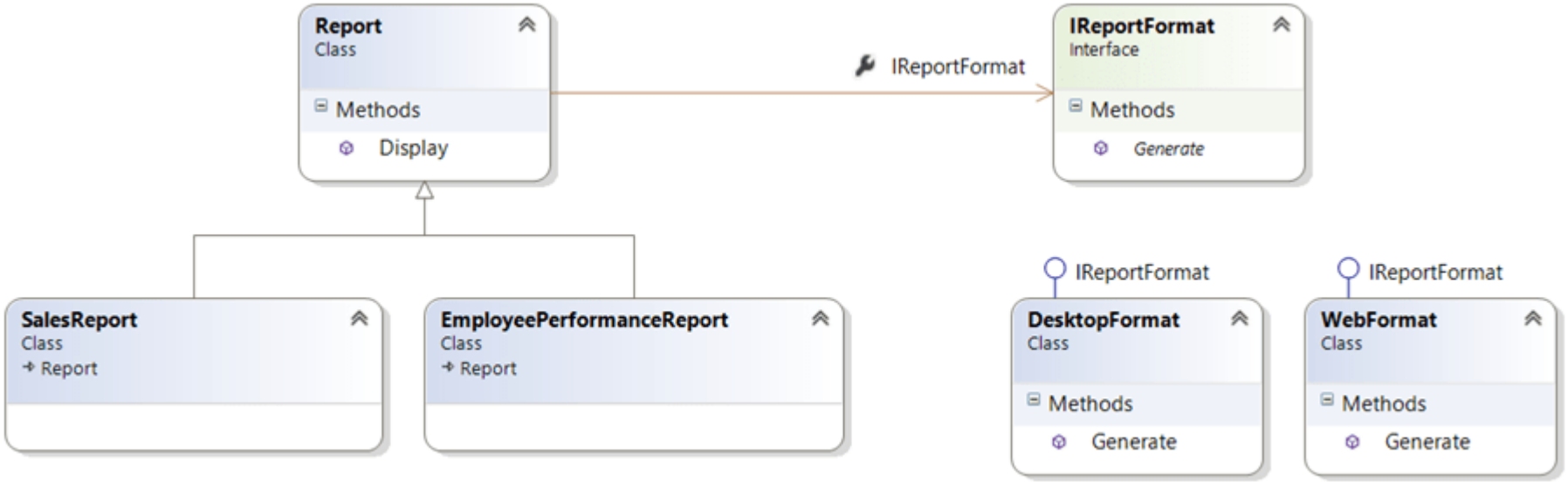
- Uygulamamız belirli bir şablonda rapor hazırlayan (satış ve çalışan performans raporları gibi) ve bu raporları da farklı formatlarda (Web Formatı ya da masaüstü formatı gibi) sisteme kaydeden bir işleve sahip olsun. Böyle bir durumda sınıf diyagramı nasıl tasarlanır?

BRIDGE (KÖPRÜ) TASARIM DESENİ



BRIDGE (KÖPRÜ) TASARIM DESENİ

- İki rapor türü ve iki rapor formatı için uygun bir model.
- Bridge Design Pattern diyor ki; soyutlaşmış (abstract) bir yapıyı, implementasyondan ayır. Böylece bağımsız olarak geliştirilebilir iki yapı elde edilir. **Rapor Formatı, bu modelde soyutlaşmış bir yapıdır.** Yani, satış ya da çalışan performans raporunu oluştururken, hangi formatta (Masaüstü veya web) kaydetmesi gerektiğini söylemek yeterli olacak. Bu durumu ayarlamak için **IReportFormat** isminde bir interface oluşturulur ve ilgili formatlar bu interface'den implemente edilir.



BRIDGE (KÖPRÜ) TASARIM DESENİ

```
public interface IReportFormat
{
    3 references
    void Generate();
}

public class WebFormat : IReportFormat
{
    3 references
    public void Generate()
    {
        Console.WriteLine("Rapor Web Formatında oluşturuldu");
    }
}

public class DesktopFormat : IReportFormat
{
    3 references
    public void Generate()
    {
        Console.WriteLine("Rapor Desktop Formatında oluşturuldu");
    }
}
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

```
public class Report
{
    2 references
    public IReportFormat ReportFormat { get; private set; }

    2 references
    public Report(IReportFormat reportFormat)
    {
        ReportFormat = reportFormat;
    }

    6 references
    public virtual void Display()
    {
        ReportFormat.Generate();
    }
}
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

```
public class SalesReport : Report
{
    1 reference
    public SalesReport(IReportFormat format)
        : base(format)
    {

    }
    6 references
    public override void Display()
    {
        Console.WriteLine("--- Satış Raporu ---");
        base.Display();
    }
}

public class EmployeePerformanceReport : Report
{
    1 reference
    public EmployeePerformanceReport(IReportFormat format)
        : base(format)
    {

    }
    6 references
    public override void Display()
    {
        Console.WriteLine("--- Çalışan Performans Raporu ---");
        base.Display();
    }
}
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

Report sınıfına ve ondan türeyen sınıflara dikkat edildiğinde varsayılan Constructor üyeleri, **IReportFormat** interface'ini parametre olarak alıyorlar ve bunu **ReportFormat** özelliğine set ediyorlar.

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Report report = new EmployeePerformanceReport(new DesktopFormat());
        report.Display();

        Console.WriteLine();

        Report report2 = new SalesReport(new WebFormat());
        report2.Display();

        Console.ReadLine();

    }
}
```

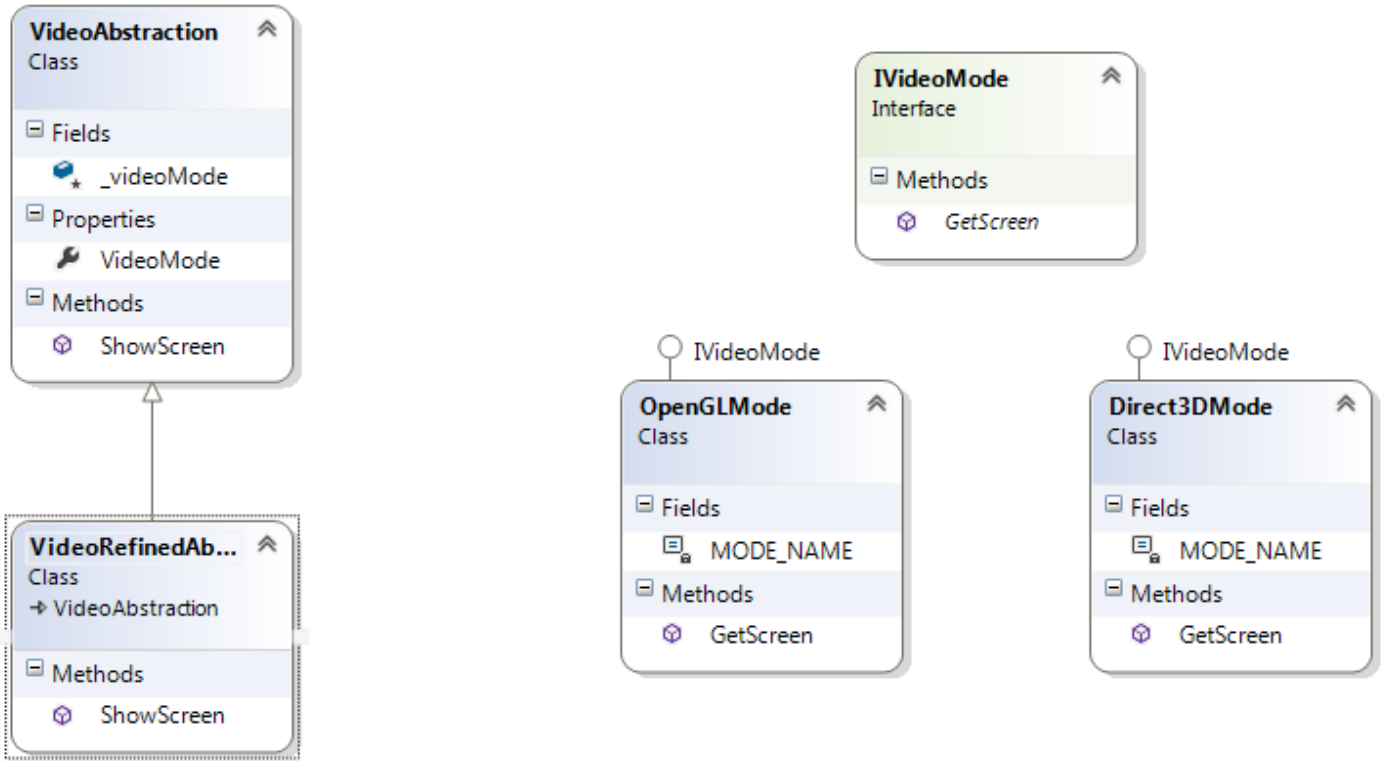
BRIDGE (KÖPRÜ) TASARIM DESENİ

- Başka bir örnek üzerinden konuya devam edersek. Örneğin bir oyun aldınız ve oyunu açarken size video modlarını soruyor.
- OpenGL, Direct3D bu seçeneklerden birini seçtikten sonra oyun artık hep bu seçimle bağlantılı olarak çalışır.
- Bunu factory class ile de yapabilirdik fakat bridge ile yapmamız bize runtime'da değiştirebilmemizi sağlayacaktır. Video modunu oyunda istediğiniz anda değiştirebilirsiniz öyle düşünün.

Kısacası en önemli kullanım alanları **eğer bir nesneyi run-time'da oluşturacaksak** ve **bu nesne diğer farklı nesneler tarafından kullanılacaksa** bu pattern'i kullanmamız doğru olacaktır.

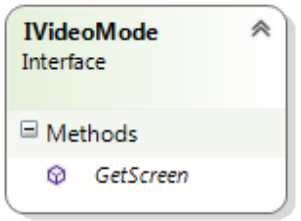
BRIDGE (KÖPRÜ) TASARIM DESENİ

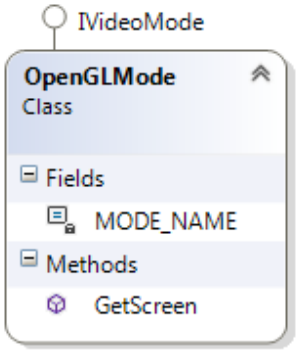
- RefinedAbstraction nesnemizi bu seçimle implemente ettikten sonra farklı nesneler tarafından da kullanılmasına olanak sağlar. Bu programı en basit anlamda yazdığımızı düşünelim class diagramı şu şekilde olacaktır:

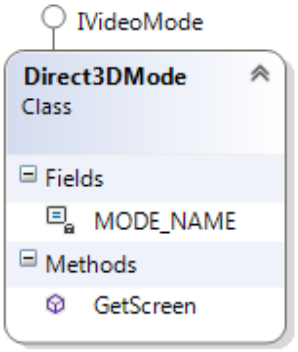


BRIDGE (KÖPRÜ) TASARIM DESENİ

```
/// <summary>
/// Implementor Class
/// </summary>
public interface IVideoMode
{
    string GetScreen();
}
/// <summary>
/// ConcreteImplementor for OpenGL
/// </summary>
public class OpenGLMode:IVideoMode
{
    const string MODE_NAME = "OpenGL Mode";
    public string GetScreen()
    {
        return string.Format("Video started with {0}", MODE_NAME);
    }
}
```

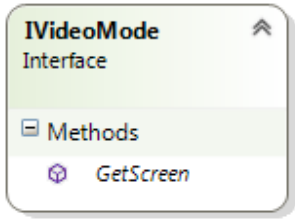


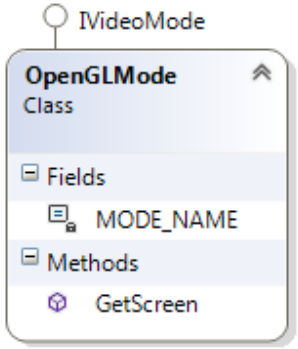


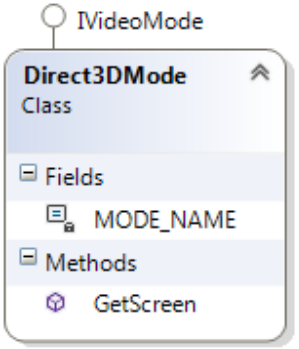


BRIDGE (KÖPRÜ) TASARIM DESENİ

```
/// <summary>
/// ConcreteImplementor for Direct3D
/// </summary>
public class Direct3DMode:IVideoMode
{
    const string MODE_NAME = "Direct3D Mode";
    public string GetScreen()
    {
        return string.Format("Video started with {0}", MODE_NAME);
    }
}
```

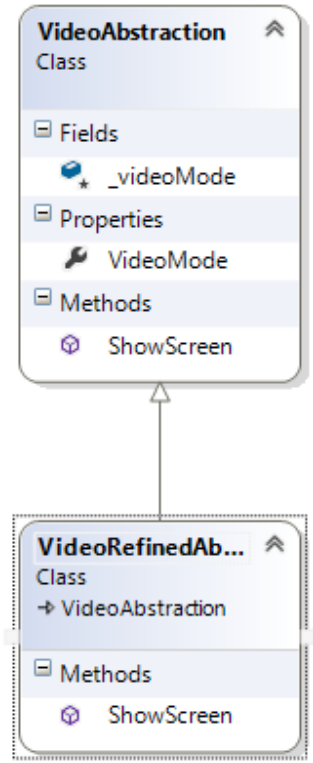






BRIDGE (KÖPRÜ) TASARIM DESENİ

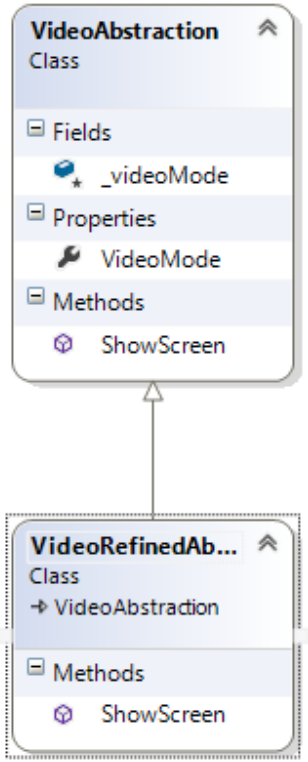
```
/// <summary>
/// The 'Abstraction' class
/// </summary>
public class VideoAbstraction
{
    protected IVideoMode _videoMode;
    // Property
    public IVideoMode VideoMode
    {
        set { _videoMode = value; }
    }

    public virtual void ShowScreen()
    {
        Console.WriteLine(_videoMode.GetScreen());
    }

}
```



Görüldüğü gibi Abstraction classına eklediğimiz VideoMode bir köprü görevi gördü.
Bu VideoMode'un tek bir implementasyonu ile farklı nesneler tarafından da kullanılmasına olanak sağladık.
Birden fazla RefinedAbstract kullanılabilir programda.

BRIDGE (KÖPRÜ) TASARIM DESENİ

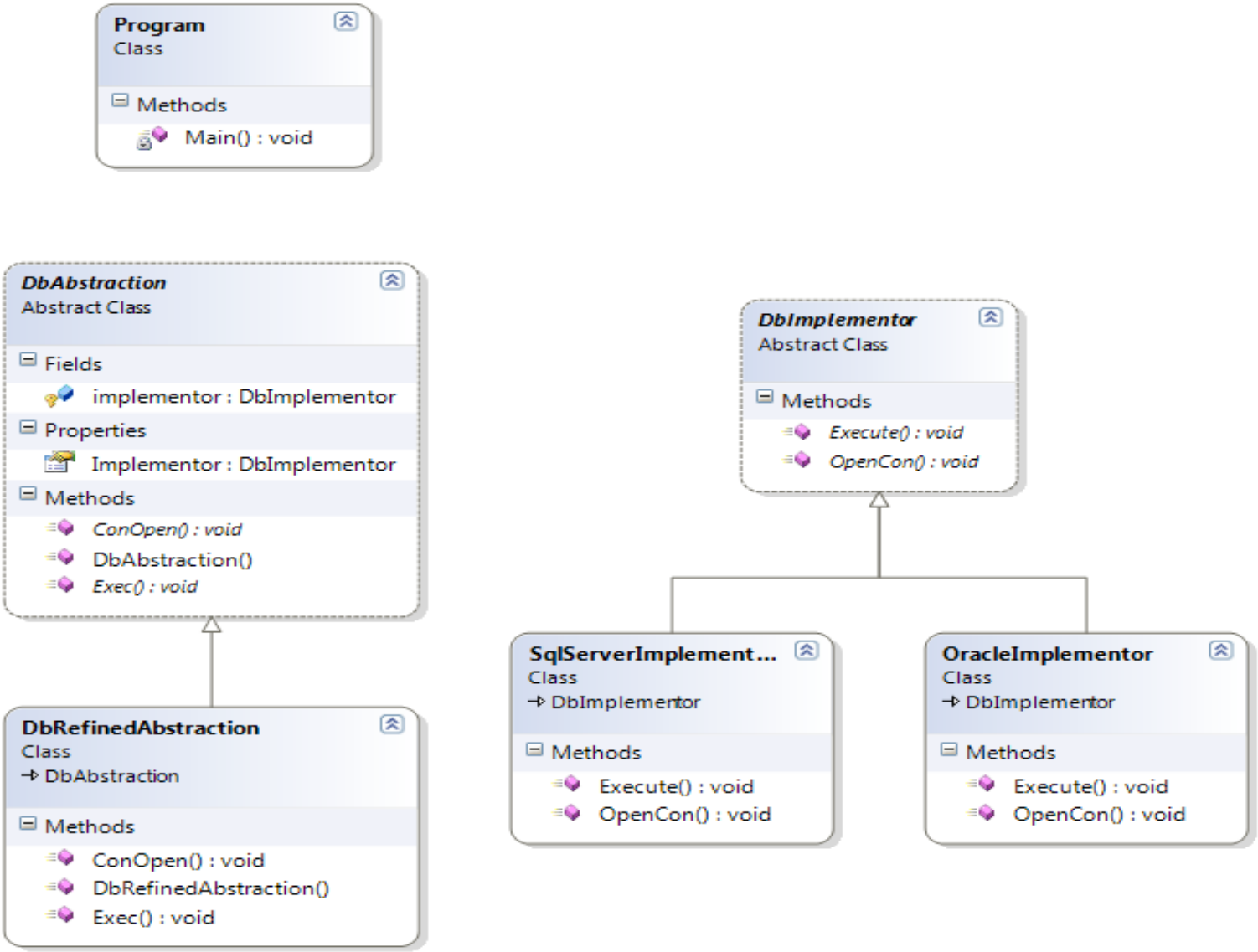
```
public class VideoRefinedAbstraction: VideoAbstraction
{
    public override void ShowScreen()
    {
        Console.WriteLine(_videoMode.GetScreen());
    }

}

static void Main(string[] args)
{
        VideoAbstraction video = new VideoRefinedAbstraction();
        video.VideoMode = new OpenGLMode();
        video.ShowScreen();

        Console.ReadLine();
}
```



BRIDGE (KÖPRÜ) TASARIM DESENİ

- **ÖRNEK:**
  - Bridge tasarım deseni ile ilgili basit bir örnek uygulama: Uygulamada veri tabanı yapısı ele alınsın.
  - Uygulama birden fazla veri tabanı desteği veriyor olsun ve execute, connection açma gibi operasyonlar her veri tabanı için farklı olsun. Uygulamanın class diyagramı aşağıdadır.

BRIDGE (KÖPRÜ) TASARIM DESENİ



BRIDGE (KÖPRÜ) TASARIM DESENİ

Uygulama kodları aşağıdadır.

```
//Implementor
abstract class DbImplementor
{
    public abstract void Execute(string Sql);
    public abstract void OpenCon(string SqlCon);
}
//ConcreteImplementor
class SqlServerImplementor : DbImplementor
{
    public override void Execute(string Sql)
    {
        Console.WriteLine("\"{0}\" - SqlServer işletildi.", Sql);
    }
    public override void OpenCon(string SqlCon)
    {
        Console.WriteLine("\"{0}\" - Sql Server Con. Açıldı.", SqlCon);
    }
}
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

```
//ConcreteImplementor
class OracleImplementor : DbImplementor
{
    public override void Execute(string Sql)
    {
        Console.WriteLine("\"{0}\" - oracle işletildi.", Sql);
    }
    public override void OpenCon(string SqlCon)
    {
        Console.WriteLine("\"{0}\" - oracle Con. Açıldı.", SqlCon);
    }
}
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

```
//Abstraction

abstract class DbAbstraction
{
    protected DbImplementor implementor;

    public DbAbstraction(DbImplementor imp)
    {
        Implementor = imp;
    }

    // Property
    public DbImplementor Implementor
    {
        set { implementor = value; }
    }
    public abstract void Exec(string Sql);
    public abstract void ConOpen(string ConStr);
}
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

```
//RefinedAbstraction

class DbRefinedAbstraction : DbAbstraction
{
    public DbRefinedAbstraction(DbImplementor imp) :
base(imp)
    {
    }
    public override void Exec(string Sql)
    {
        implementor.Execute(Sql);
    }
    public override void ConOpen(string ConStr)
    {
        implementor.OpenCon(ConStr);
    }
}
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

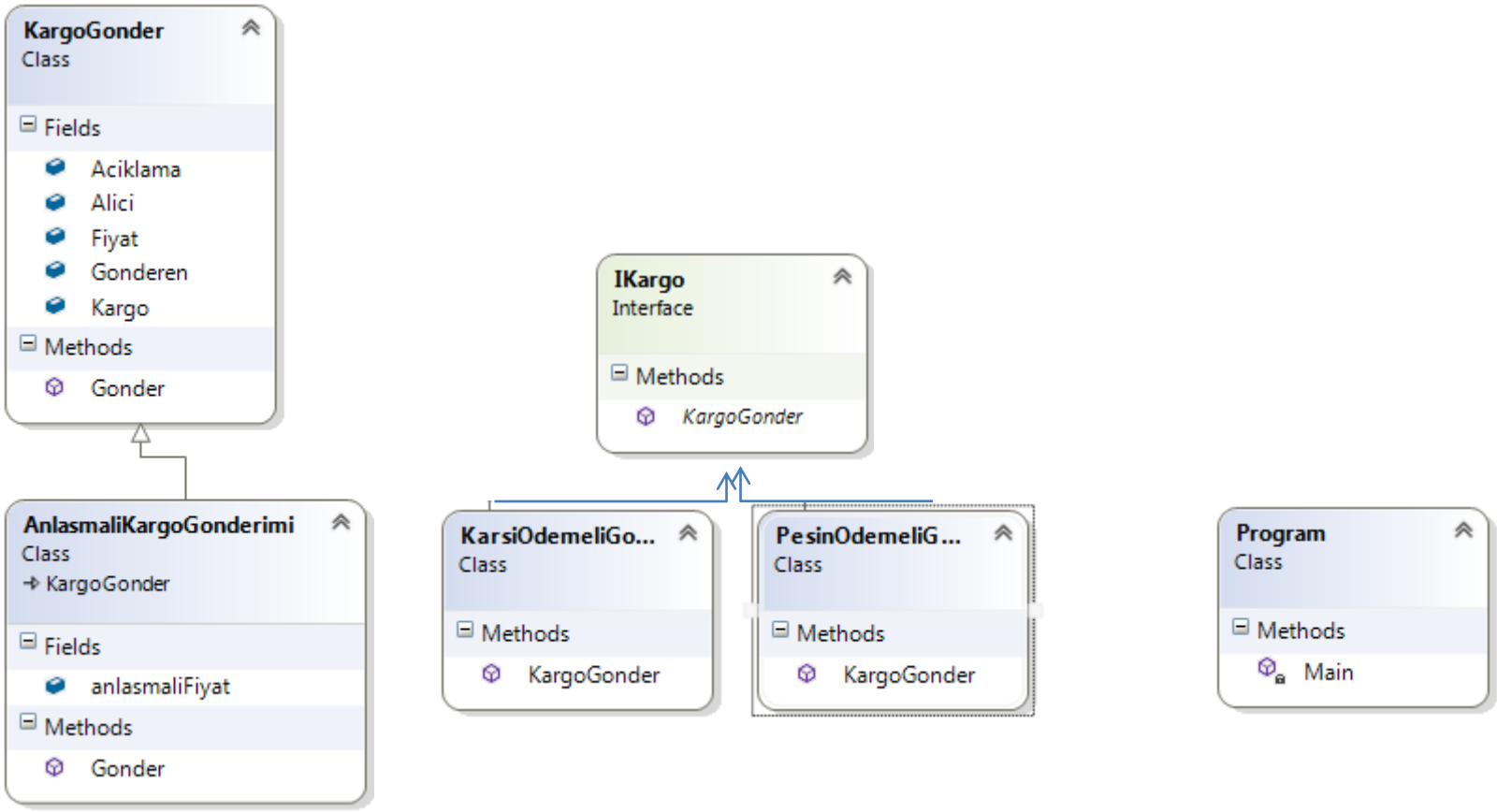
```
//client

class Program
{
    static void Main(string[] args)
    {
        DbAbstraction absDb = new
DbRefinedAbstraction(new SqlServerImplementor());
        absDb.ConOpen("e-ticaret db");
        absDb.Exec("select * from Urun");
        absDb = new DbRefinedAbstraction(new
OracleImplementor());
        absDb.ConOpen("e-ticaret db");
        absDb.Exec("select * from Urun");

        Console.ReadKey();
    }
}
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

- **ÖRNEK:**

Kargo gönderimi yapan bir yapı ele alınsın. Bu yapı peşin ödemeli ve karşı ödemeli olmak üzere iki çeşit kargo gönderimi yapıyor. Daha sonra bu yapıya anlaşmalı olunan firmaların kargo gönderimini yapacak bir özellik daha eklensin.

Sınıf Diyagramı



BRIDGE (KÖPRÜ) TASARIM DESENİ

- **IKargo:** Sınıfların implemente edileceği arayüz.
- **KarsiOdemeliKargoGonderimi:** Arayüzü implemente eden sınıf.
- **PesinOdemeliKargoGonderimi:** Arayüzü implemente eden sınıf.
- **AnlasmaliKargoGonderimi:** Abstraction sınıfını implemente eden sınıf. Projeye yeni işlev kazandırdığımız sınıf.
- **KargoGonder:** Köprü görevi gören sınıf.(Abstraction)

BRIDGE (KÖPRÜ) TASARIM DESENİ

```
public interface IKargo
    {
        void KargoGonder (string Gonderen, string Alici);
    }
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

//Concrete1 Arayüzü implemente eden sınıf.

```
public class KarsiOdemeliGonderim : IKargo
   {
      public void KargoGonder (string Gonderen, string Alici)
            {
                  // Kodlar
            }
   }
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

//Concrete2 Arayüzü implemente eden sınıf.

```
public class PesinOdemeliGonderim :IKargo
    {
        public void KargoGonder(string Gonderen, string Alici)
        {
            // Kodlar buraya
        }
    }
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

```
// Abstraction yani Köprü görevini üstlenen nesnemiz

public  class KargoGonder
   {
      public IKargo Kargo;

      public string Gonderen;
      public string Alici;

      public virtual void Gonder()
      {
         Kargo.KargoGonder(Gonderen, Alici);
      }
   }
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

```
public class AnlasmaliKargoGonderimi : KargoGonder

    {
            public override void Gonder()
            {
                    Kargo.KargoGonder(Gonderen, Alici);
            }
    }
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

```
//Client
class Program
   {
       static void Main(string[] args)
      {
                        KargoGonder kargo = new KargoGonder();
                        kargo.Kargo = new PesinOdemeliGonderim();
                        kargo.Alici = "XX";
                        kargo.Gonderen = "YY";
                        kargo.Gonder();
              }
    }
```

BRIDGE (KÖPRÜ) TASARIM DESENİ

```
KargoGonder kargo1 = new KargoGonder();
            kargo1.Kargo = new KarsiOdemeliGonderim();
            kargo1.Alici = "XX";
            kargo1.Gonderen = "YY";
                        kargo1.Gonder();

//Refined
          AnlasmaliKargoGonderimi kargo2 = new AnlasmaliKargoGonderimi();
            kargo2.Alici = "XX";
            kargo2.Gonderen "YY";
            kargo2.Kargo = new PesinOdemeliGonderim();
            kargo2.Gonder();
            Console.ReadLine();
        }
    }
}
```

COMPOZİTE

Tasarım Deseni

COMPOZITE TASARIM DESENİ

- Bu tasarım deseninin amacı, nesneleri ağaç yapısına göre düzenleyerek, ağaç yapısındaki alt üst ilişkisini kurmaktır.
- Bu tasarım desenine göre, ağaç yapısındaki üst ve alt nesneler aynı arayüz sınıfından türeyerek, birbirlerine benzerler.
- Yani istemci, yaprak nesneye de üst nesneye de aynı şekilde davranır.
- Böylece hiyerarşiyi ifade etmek ve hiyerarşi üzerinde işlem yapmak kolaylaşır.
- Hiyerarşiye yeni nesneler eklemek de kolay hale gelir.

COMPOZITE TASARIM DESENİ

- Composite tasarım desenini hiyerarşik nesne yapıları kurmak istediğimizde ve sistemimizde parçalardan bütün oluşturmak istediğimizde kullanırız.

- Sistemin bütünü ve parçaları birbirinden soyutlandığı için operasyonel olarak birbirinden farklı işler yaptırılabiliriz.

- Sistemin bütünü, parçalardan oluşur.

- Geliştirdiğimiz uygulamalardaki menüleri ve menü üzerindeki menü item'ları düşünebiliriz. Menü, menü item'lardan oluşur.

COMPOZITE TASARIM DESENİ

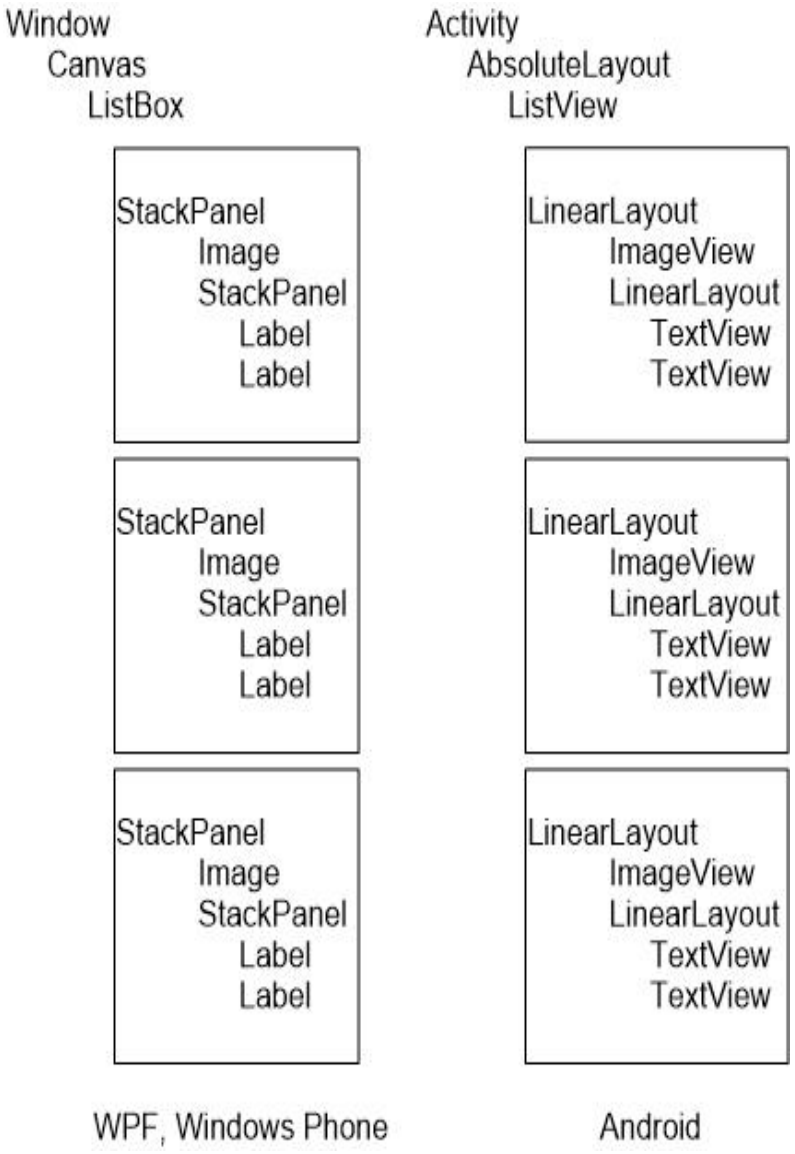
- Gündelik hayatta parçalarının başka alt parçalardan oluştuğu pek çok kompozit varlıkla karşılaşmaktayız. Örneğin bir dijital entegre devre n tane transistörden oluşmakta, transistörlerin de kendi içlerinde -kavramsal olarak- diyotlardan oluştuğu bilinmektedir.

- Nesne yönelimli sistemlerin pek çoğunda da (bilhassa GUI frameworklerinde) kompozisyon olgusuyla sıkça karşılaşılmaktadır. Bunun anlamı; nesnelerin belirli bir hiyerarşiye uygun şekilde -genellikle ağaç hiyerarşisi biçiminde- iç içe geçmesi demektir.

- Bu duruma örnek vermek gerekirse WPF/Windows Phone ile Android platformlarından söz edilebilir zira her ikisinde de ekran tasarımları tümüyle kompozisyon olgusuna dayanmaktadır.

- Söz gelimi aşağıdaki şekilde bu platformlarda benzer işlevler gösteren nesneler kullanılarak birer kompozisyon oluşturulmaktadır.

COMPOZITE TASARIM DESENİ



- **Window/Activity** içerisinde bir **Canvas/AbsoluteLayout** kullanılmış ve içerisine bir **ListBox/ListView** nesnesi konumlandırılmıştır.
- **ListBox/ListView** nesnesinin listelediği elemanların her birisi ise (bir dikdörtgen içerisinde gösterilmiştir) yatay yönetimli bir **StackPanel/LinearLayout** içerisindeki **Image/ImageView** ile başka bir düşey yönelimli **StackPanel/LinearLayout** barındırmakta onun içerisinde ise aynı seviyede iki adet **Label/TextView** bulunmaktadır.
- Ve bu benzer yapılar adeta birden çok alt parçanın birleşimiyle oluşan bütünlere örnektir.

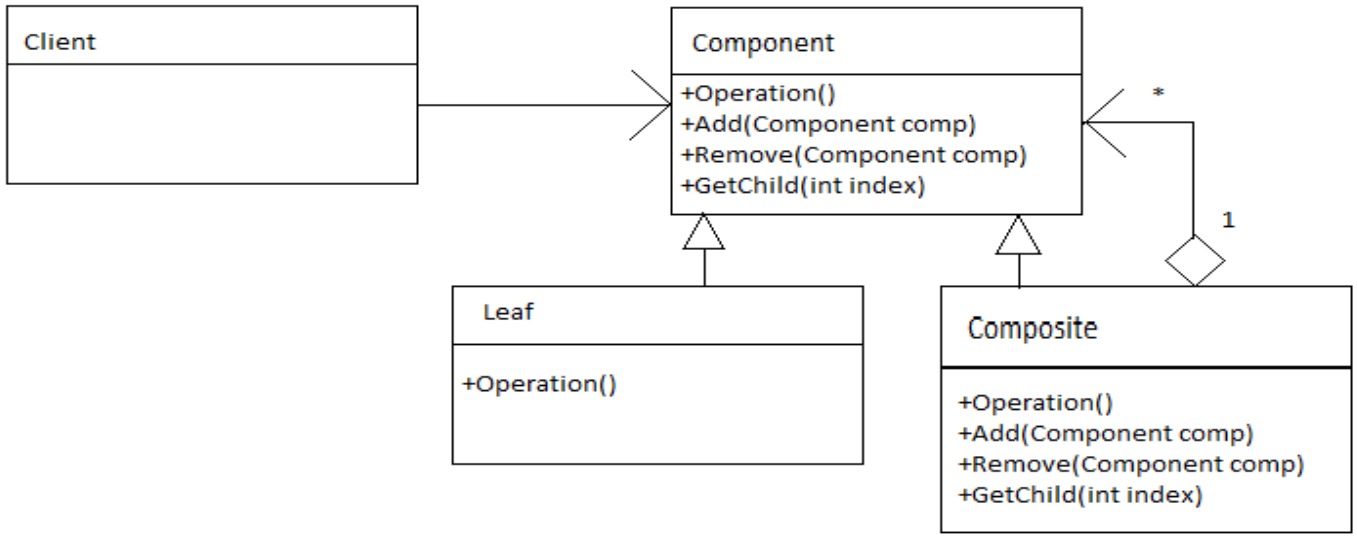
Şekil.2 : Farklı GUI Framework'lerinde kompozisyon örnekleri

COMPOZITE TASARIM DESENİ

Nasıl Çalışır?

- Composite tasarım deseni(Composite Design Pattern) yapısal tasarım desenlerindendir.
- Hiyerarşik olarak ilişkisel nesneleri düzenlemeyi hedefler. Hiyerarşik ilişkilerden kasıt alt-üst ilişkileridir.
- Bir nesneye yaptırılmak istenen iş, o nesnenin altındaki tüm ilişkili nesneler için geçerli olur. Bu durum çoğu zaman askeri düzene benzetilir. Komutan emir verdiğinde alt-komutanlardan başlayarak er askerlere kadar emir geçerli olur.

COMPOZITE TASARIM DESENİ



- Composite tasarım desenini oluşturan elemanları anlatacak olursak;

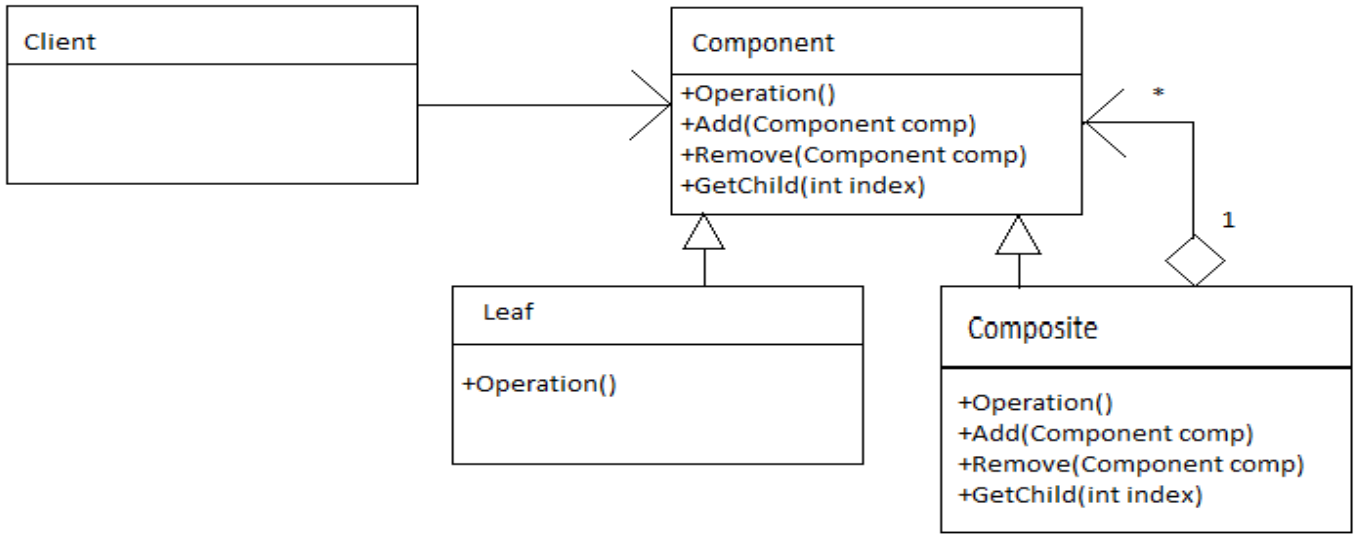
i. **Client:** İstemci sınıf.

ii. **Component:** Soyut sınıftır. Özellikler ve operasyon tanımlarını barındırır.

iii. **Composite:** Component nesnesinin somut hâlidir. İçerisinde Component listesi tutar. Bu liste sayesinde hiyerarşik yapıyı kurar. Component listesi için gerekli olan ekle, sil, component değerine ulaşma fonksiyonlarını barındırır.

iv. **Leaf:** Temel operasyon sınıfı.

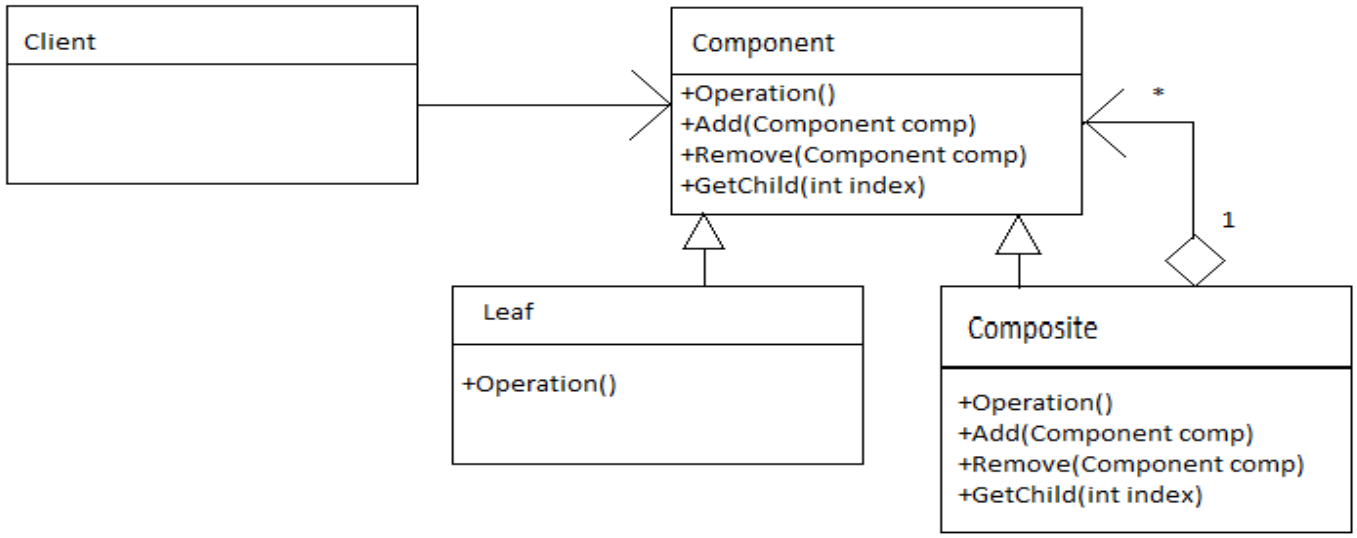
COMPOZITE TASARIM DESENİ



- Composite tasarım deseninde kullanacağımız component nesnesi ve bunu implement edecek olan **composite** ve **leaf** nesneleri bulunmaktadır.

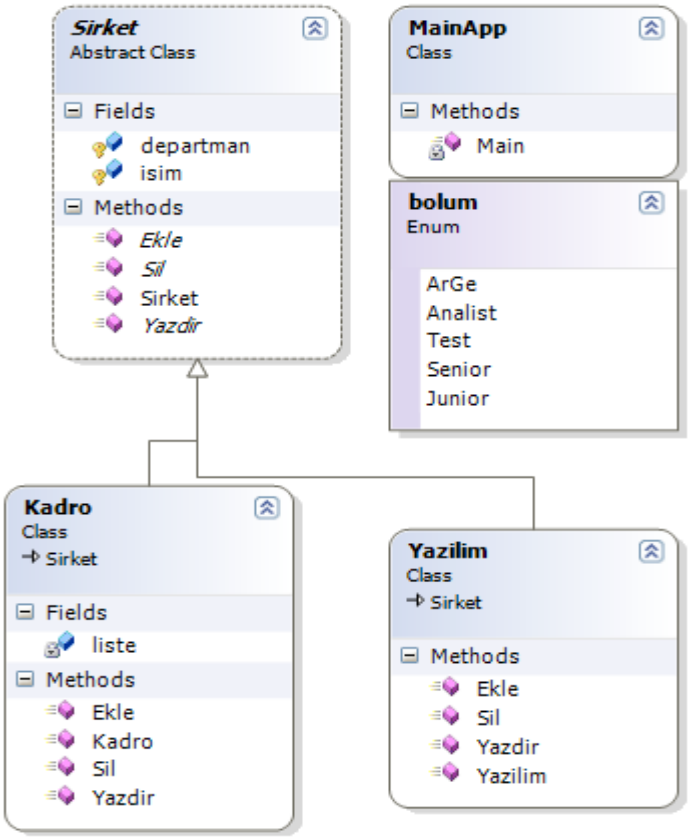
- Component tipindeki nesnemizin metotları abstract yada virtual olarak tanımlanmalıdır.

COMPOZITE TASARIM DESENİ



- **Component ;** Nesneler için bir arayuz olarak kullanılır. Kullanılacak tüm sınıflar için ortak bir arayuz olması ile birlikte alt bileşenlere erişirken nesneleri ve metotları da kendisinde barındırabilir, yani abstract yada interface olarak kullanılmaktadır. Bir nevi tanımlayıcı rolünü üstlenmektedir.
- **Composite ;** Ağaç yapısındayken kendisinden türeyen nesnelerin bulunduğu bir sınıftır. Hiyerarşi içerisinde alt nesnelere sahiptir. Bileşik yapı rolünü üstlenmektedir.
- **Leaf ;** Ağaç yapısındayken kendisinden herhangi bir nesne türemeyecek şekilde kullanılmaktadır. Hiyerarşi içerisindeyken alt nesnelere sahip olamamaktadır.
- **Client ;** Oluşturduğumuz nesneleri kullanmak için talepte bulanacak olan istemci nesnemizdir.

Bir şirkete ait bölümler ve bu bölümlerde çeşitli kadrolarda çalışanlar mevcuttur.

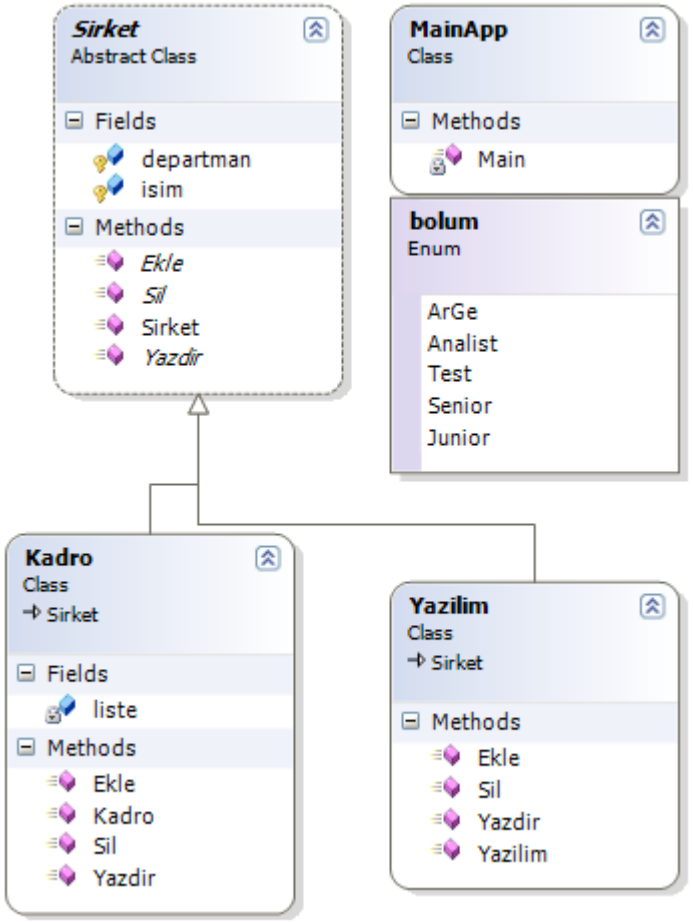



```
enum bolum
{
    ArGe,
    Analist,
    Test,
    Senior,
    Junior
}
```

Uygulama icerisinde kullanılan Bolum enum sabitinin kod yapısı ;

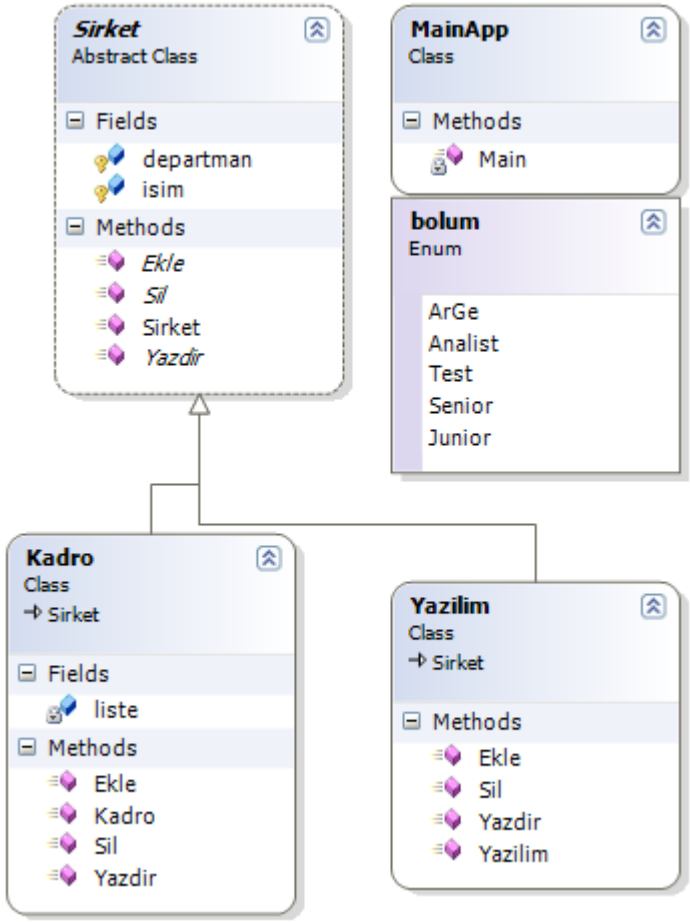
Component yapısı için oluşturulan Sirket abstract sınıfına ait kod;

```
abstract class Sirket
{
    protected string isim;
    protected bolum departman;

    public Sirket(string _isim, bolum _departman)
    {
        this.isim = _isim;
        this.departman = _departman;
    }

    public abstract void Ekle(Sirket ys);
    public abstract void Sil(Sirket ys);
    public abstract void Yazdir(int satirbasi);
}
```



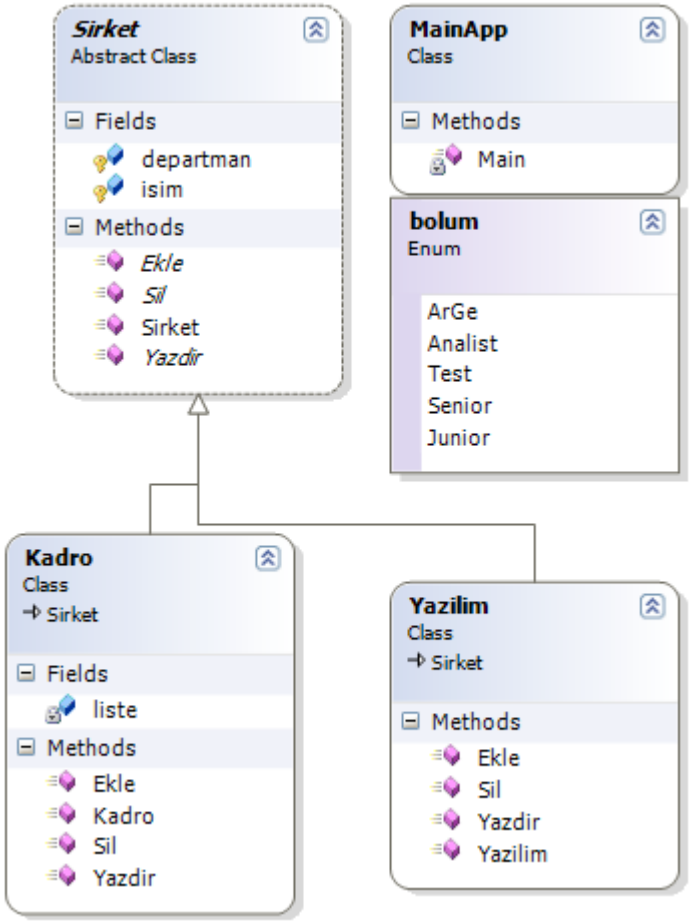
Leaf yapısına uygun olacak şekilde oluşturulacak olan yazılım sınıfının kod yapısı;

```
class Yazilim : Sirket
{
    public Yazilim(string isim, bolum departman)
        : base(isim, departman)
    {
    }

    public override void Ekle(Sirket t)  {  }
    public override void Sil(Sirket t)   {  }
    public override void Yazdir(int satirbasi)
    {
        Console.WriteLine(new String('-', satirbasi) +
            "> " + isim + " - " + departman);
    }
}
```



Composite yapısı için oluşturulan Kadro sınıfının kod yapısı ;

```
class Kadro : Sirket
{
    private List<Sirket> liste = new List<Sirket>();

    public Kadro(string isim, bolum departman)
        : base(isim, departman)
    {        }

    public override void Ekle(Sirket u)
    {
        liste.Add(u);
    }

    public override void Sil(Sirket u)
    {
        liste.Remove(u);
    }

    public override void Yazdir(int satirbasi)
    {
        Console.WriteLine(new String('-', satirbasi) +
            "+ " + isim + " - " + departman );

        foreach (Sirket u in liste)
        {
            u.Yazdir(satirbasi + 2);
        }
    }
}
```



COMPOZITE TASARIM DESENİ

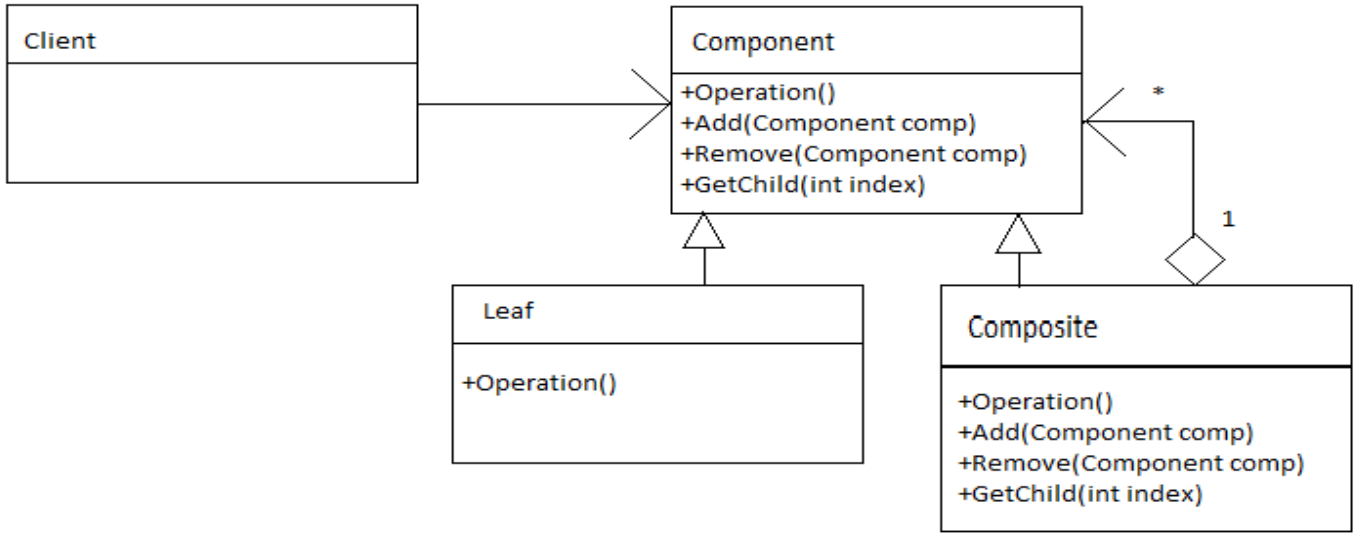
i. UML diyagramından anlaşılacağı üzere Composite tasarım deseninde 4 temel yapı vardır.

ii. Component yapısı abstract sınıf olarak tasarlanır, içinde property ve operasyonların tanımları yapılır.

iii. Composite ve life nesneleri ise Component nesnesini uygulayan gerçek nesneleri temsil eder.

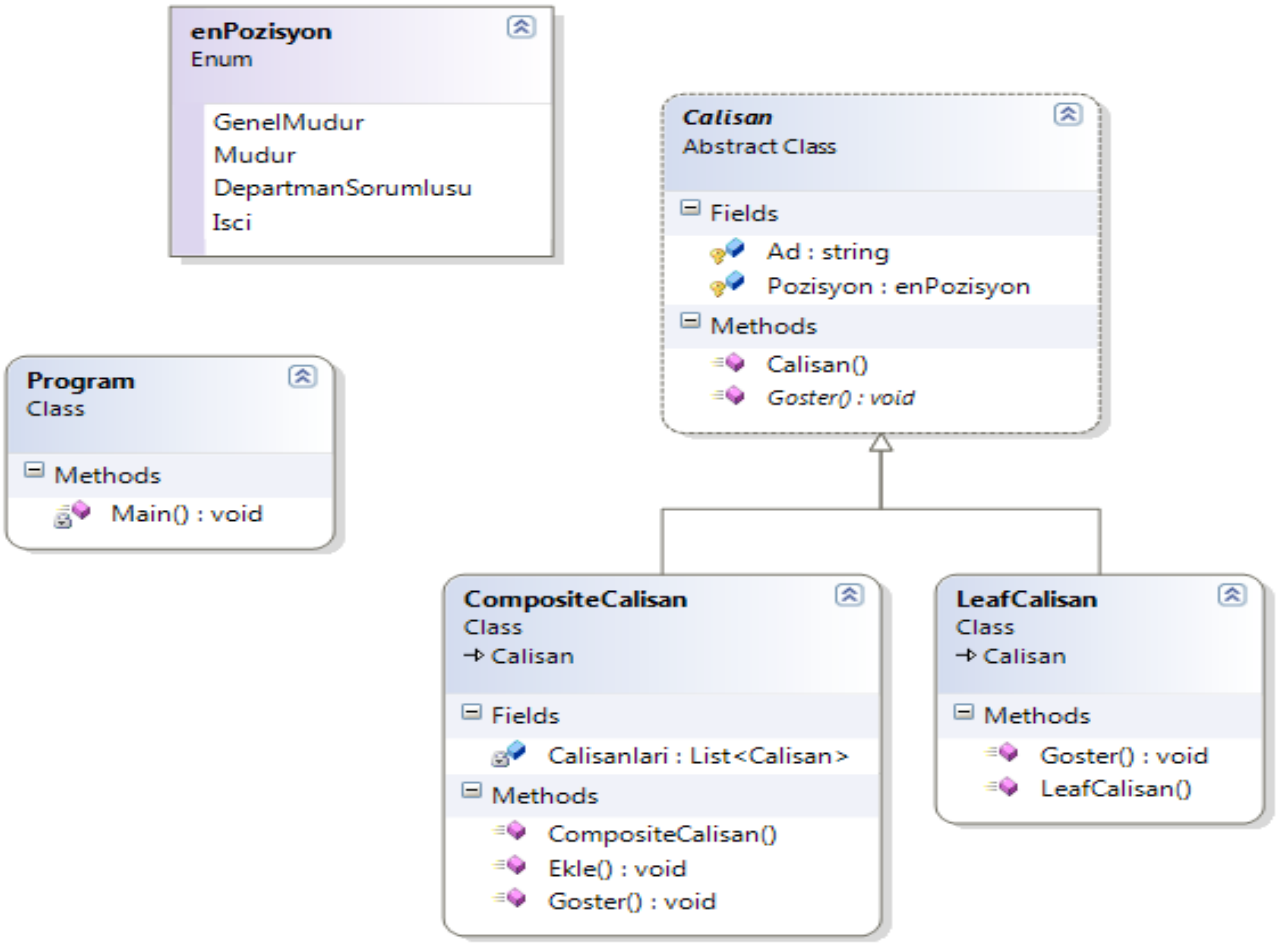
iv. Composite nesnesi içinde Component listesi barındırır ve hiyerarşik olarak altında olan nesneler bu listede tutulurlar.

v. Leaf nesnesinde ise altında nesneler olmayan en alt katmanı temsil eder.



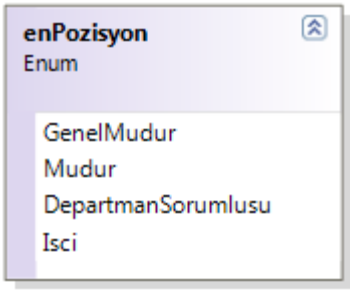
COMPOZITE TASARIM DESENİ

Örnek uygulamada şirket yapısı ele alınsın. Çalışanlar hiyerarşik olarak alınsın ve göstermelik olarak ismi konsola yazdırılsın. Uygulamanın class diyagramı aşağıdadır.



Örnek uygulanın kodları açıklamaları ile beraber aşağıdadır.

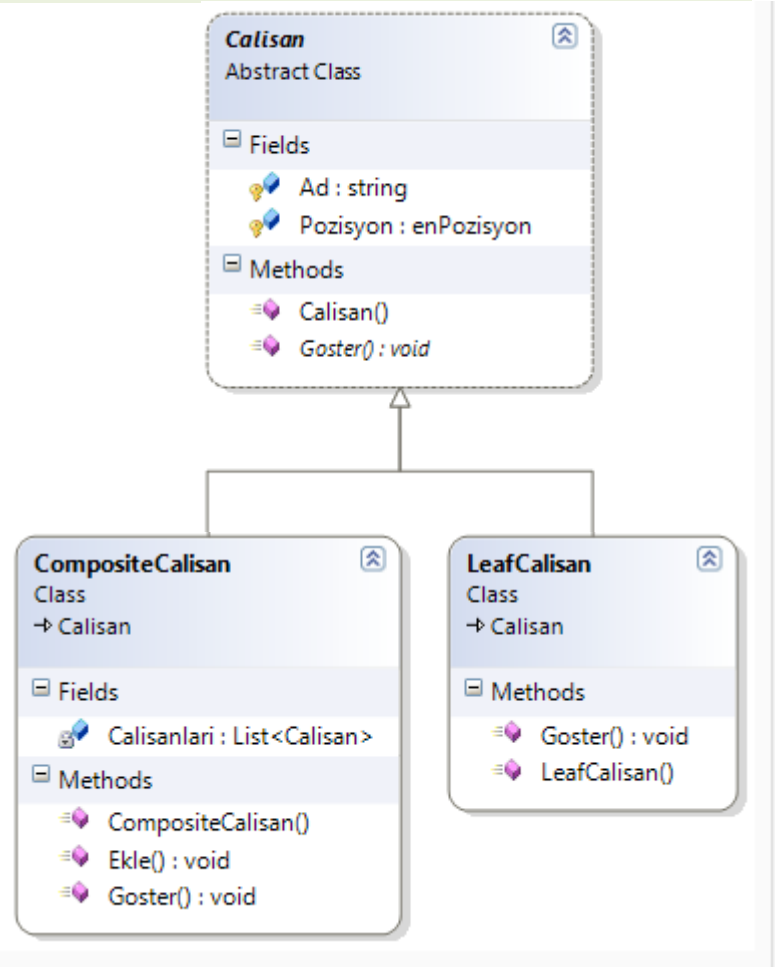
```
//yardımcı enum

enum enPozisyon
{
    GenelMudur = 1,
    Mudur = 2,
    DepartmanSorumlusu = 3,
    Isci = 4
}
```



COMPOZITE TASARIM DESENİ

Component yapısı abstract sınıf olarak tasarlanır, içinde property ve operasyonların tanımları yapılır.

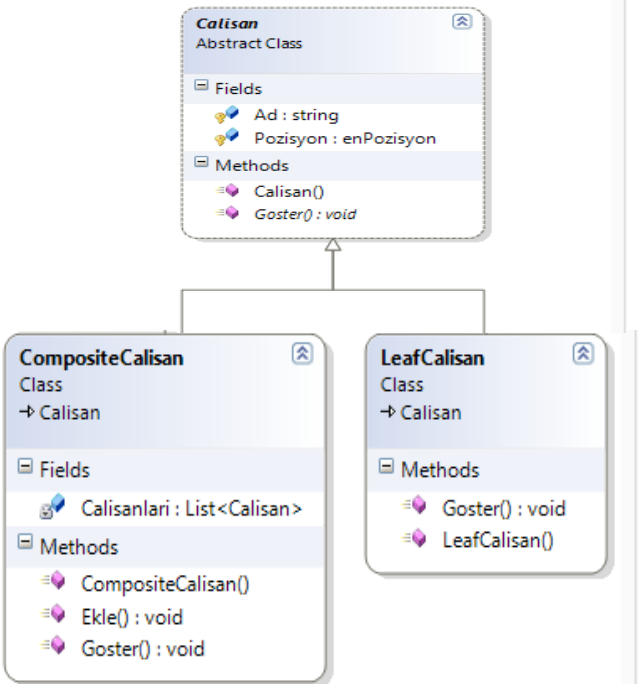
```
//component yapısı

abstract class Calisan
{
    protected string Ad;
    protected enPozisyon Pozisyon;
    public Calisan(string Ad, enPozisyon Pozisyon)
    {
        this.Ad = Ad;
        this.Pozisyon = Pozisyon;
    }
    public abstract void Goster();
 //Leaf ve Composite de uygulanacak metot
}
```



COMPOZITE TASARIM DESENİ

Composite ve life nesneleri ise Component nesnesini uygulayan gerçek nesneleri temsil eder.

```
//Leaf yapısı
class LeafCalisan : Calisan
{
    public LeafCalisan(string Ad, enPozisyon Pozisyon): base(Ad, Pozisyon)
    {
    }
    public override void Goster()
    {
        Console.WriteLine("{0} {1}", base.Pozisyon.ToString(), base.Ad);
    }
}
```

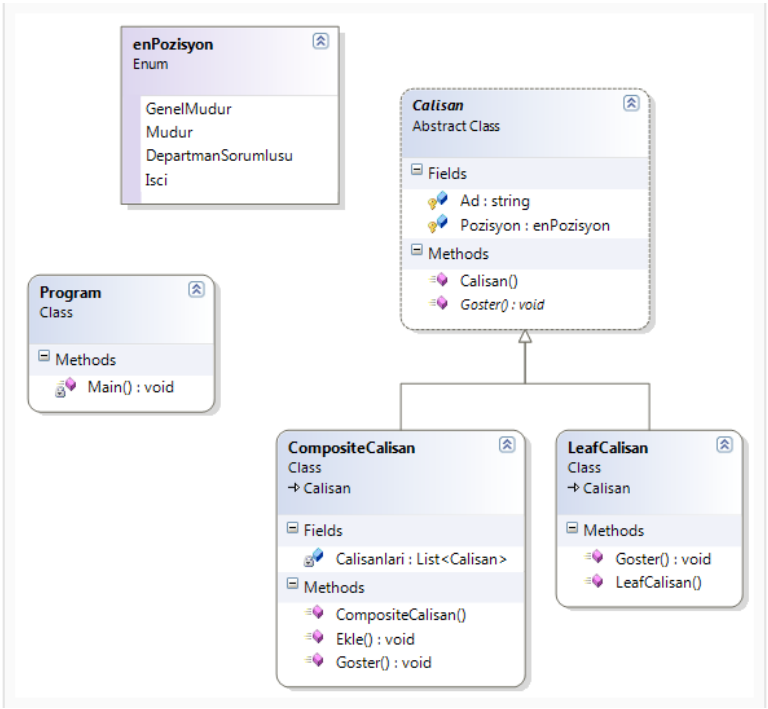


COMPOZITE TASARIM DESENİ

Composite ve leaf nesneleri ise Component nesnesini uygulayan gerçek nesneleri temsil eder.

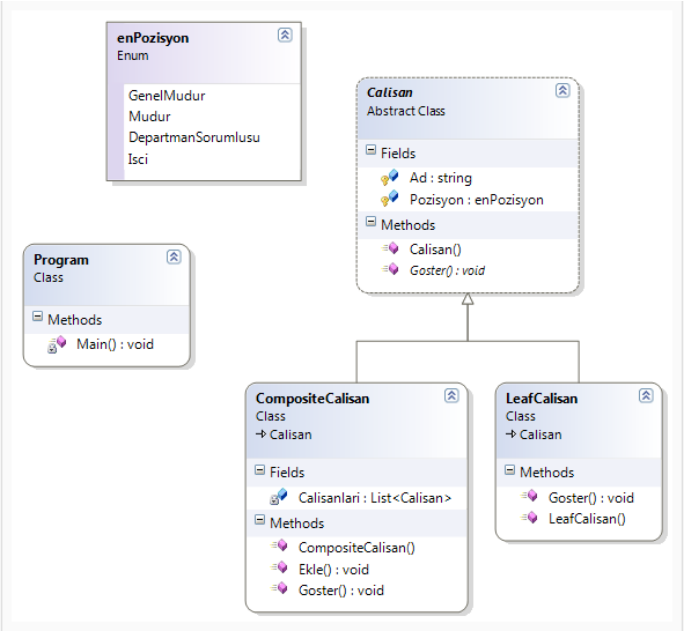
Composite nesnesi içinde Component listesi barındırır ve hiyerarşik olarak altında olan nesneler bu listede tutulurlar.

```
//Composite  yapısı

class CompositeCalisan : Calisan
{
    List<Calisan> Calisanlari;
    public CompositeCalisan(string Ad, enPozisyon Pozisyon) : base(Ad,
Pozisyon)
    {
        Calisanlari = new List<Calisan>();
    }
    public void Ekle(Calisan c)
    {
        Calisanlari.Add(c);
    }
    public override void Goster()
    {
        Console.WriteLine("{0} {1}", base.Pozisyon.ToString(), base.Ad);
        foreach (Calisan item in Calisanlari)
        {
            item.Goster();
        }
    }
}
```



COMPOZITE TASARIM DESENİ

```
class Program
{
    static void Main(string[] args)
    {
        //İlk olarak Root Composite yi oluşturuyoruz
        CompositeCalisan GenelMudur = new CompositeCalisan("ali", enPozisyon.GenelMudur);
        //Genel müdürün altında çalışan diğer çalışanları hiyerarşik olarak ekliyoruz
        //Altında eleman olmayan çalışanlar LeafCalisan sınıfı ile ifade edilir.

        CompositeCalisan Mudur = new CompositeCalisan("Ahmet ", enPozisyon.Mudur);
        Mudur.Ekle(new LeafCalisan("mehmet ", enPozisyon.Isci));
        Mudur.Ekle(new LeafCalisan("ayşe ", enPozisyon.Isci));
        //Root komposite altındaki Composite yi ekliyoruz.
        GenelMudur.Ekle(Mudur);
        //Composite için döngü ile altındaki Calisan sınıfları, Leaf için sadece kendisi
        GenelMudur.Goster();
        Console.ReadKey();
    }
}
```



Yazılım Tasarımı ve Mimarisi

Örnek:

- Yazılım firmasına yeni bir istek geldiği varsayılsın. Firma, bir CRM uygulaması ile entegre çalışacak, aylık masraflarını raporlayabileceği bir uygulama istiyor. Firma kendi hiyerarşisinde genel müdürlüğüne bağlı müdürlükler, bu müdürlüklere bağlı bölgeler, bölgelere bağlı bayiler ve son olarak bu bayilerde çalışan insanlar olduğunu söylüyor. Maliyet rakamını hem tüm şirketi için genel olarak hem de istediği müdürlük, bölge, bayi gibi farklı hiyerarşi seviyelerinde görmek istediğini söylüyor.

- Müşteriden alınan bu bilgilere göre şirketin bir hiyerarşik yapısının olacağı ve her seviyedeki hiyerarşik yapının maliyetini hesaplamak gerektiği bilgisine ulaşılır. Bu bilgi biraz irdelenirse en alttan yukarıya doğru çıkıldığında hiyerarşi; parçalardan bir araya gelerek hiyerarşinin tamamını yani sistemin bütününü oluşturuyor. Bu bilgileri de göz önüne alarak tasarım Composit tasarım deseni ile gerçeklenebilir.

Yazılım Tasarımı ve Mimarisi

```
//Crm Soyut Çalışan
public abstract class Worker
{
    public abstract int GetCost();
}
```

- Worker sınıfı composit desendeki component sınıfına karşılık gelir. Sıra geldi somut sınıflarını yazmaya. İlk olarak hiyerarşinin temel yapısı oluşturulsun.

```
//Çalışan insan
public class Working Person : Worker
{
    //Çalışan ücreti
    public override int GetCost()
    {
        return 25;
    }
}
```

- Varsayılan değer olarak, maaş 25 lira olarak ayarlandı. Şimdi hiyerarşiyi oluşturacak compozite sınıfını inceleyelim.

Yazılım Tasarımı ve Mimarisi

```
//Bayii, Bölge, Müdürlük, Genel Müdürlük yerine geçecek
//Composite Sınıf
public class CompositeDealers : Worker
{
    List<Worker> workerList;
    public CompositeDealers()
    {
        //Çalışan listesi
        workerList = new List<Worker>();
    }
    //Yeni çalışan ekle
    public void Add(Worker person)
    {
        workerList.Add(person);
    }
    //Hiyerarşik olarak kendi seviyesinin altındaki tüm
    //çalışanların maaşını hesapla
    public override int GetCost()
    {
        int result = 0;
        foreach (Worker worker int workerList)
        {
            result += worker.GetCost();
        }
        return result;
    }
}
```

- Yorum satırları ile açıklanan sınıf, hiyerarşik yapının kurulmasını sağlar. Yazılan örnek metod ile daha iyi anlaşılan yapı, bir firmanın tüm genel müdürlük, bölge müdürlüğü, bayi, çalışan ilişkilerini karşılayacaktır.

Yazılım Tasarımı ve Mimarisi

```
//Genel Müdürlüğümüz
CompositeDealers genelMudurluk = new CompositeDealers();
//Genel müdürlüğe bağlı müdürlüklerimiz
CompositeDealers İcAnadoluMudurlugu = new CompositeDealers();
//Müdürlüğe bağlı bölgelerimiz
CompositeDealers ankaraBolgesi = new CompositeDealers();
CompositeDealers sivasBolgesi = new CompositeDealers();
//Bölgelere bağlı bayiilerimiz
CompositeDealers ankara1 = new CompositeDealers();
CompositeDealers ankara2 = new CompositeDealers();
CompositeDealers sivas1 = new CompositeDealers();

//Bayii Çalışanlarımız
ankara1.Add(new WorkingPerson());
ankara1.Add(new WorkingPerson());
ankara1.Add(new WorkingPerson());
ankara1.Add(new WorkingPerson());
ankara1.Add(new WorkingPerson());

ankara2.Add(new WorkingPerson());

ankaraBolgesi.Add(ankara1);
ankaraBolgesi.Add(ankara2);

sivas1.Add(new WorkingPerson());

sivasBolgesi.Add(sivas1);


İcAnadoluMudurlugu.Add(ankaraBolgesi);
İcAnadoluMudurlugu.Add(sivasBolgesi);

genelMudurluk.Add(İcAnadoluMudurlugu);
```

Örnek test kodu şu şekilde olacaktır.

Yazılım Tasarımı ve Mimarisi

```
Console.WriteLine("-----------------------------------------------------");
Console.WriteLine("Ankara 1 Bayi Maliyet : " + ankara1.GetCost().ToString());
Console.WriteLine("Ankara 2 Bayi Maliyet : " + ankara2.GetCost().ToString());
Console.WriteLine("Ankara Bölge Maliyet : " + ankaraBolgesi.GetCost().ToString());
Console.WriteLine("-----------------------------------------------------");
Console.WriteLine("Sivas 1 Bayi Maliyet : " + sivas1.GetCost().ToString());
Console.WriteLine("Sivas Bölge Maliyet : " + sivasBolgesi.GetCost().ToString());
Console.WriteLine("-----------------------------------------------------");
Console.WriteLine("İç Anadolu Müdürlüğü Maliyet : " + İcAnadoluMudurlugu.GetCost().ToString());
Console.WriteLine("-----------------------------------------------------");
Console.WriteLine("Toplam Maliyet : " + genelMudurluk.GetCost().ToString());
Console.WriteLine("-----------------------------------------------------");
```

Ekran Çıktısı:

```
-----------------------------------------------------
Ankara 1 Bayi Maliyet : 125
Ankara 2 Bayi Maliyet : 25
Ankara Bölge Maliyet : 150
-----------------------------------------------------
Sivas 1 Bayi Maliyet : 25
Sivas Bölge Maliyet : 25
-----------------------------------------------------
İç Anadolu Müdürlüğü Maliyet : 175
-----------------------------------------------------
Toplam Maliyet : 175
-----------------------------------------------------
```

Yazılım Tasarımı ve Mimarisi

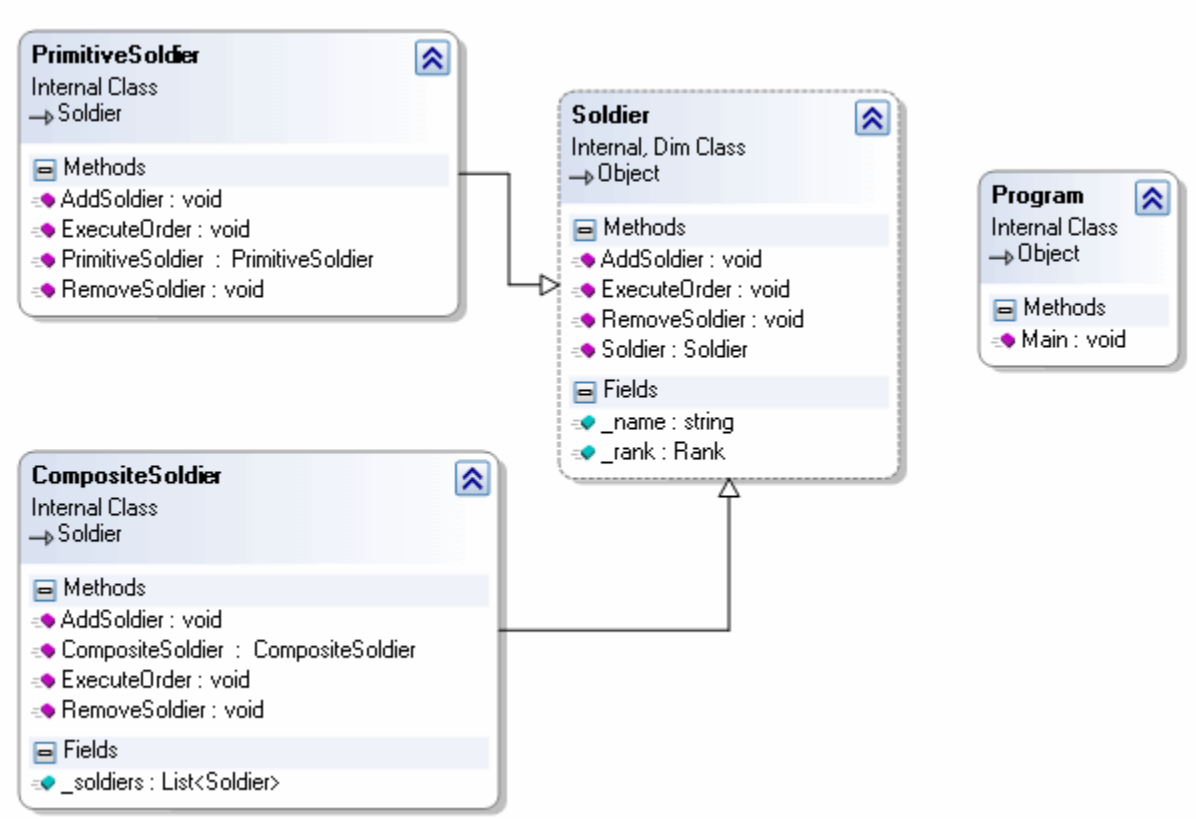
- Örnekde, kurduğumuz ordunun içerisindeki organizasyonel ağacı tasarlamaya çalışılacak. Buna göre;

General
Colonel
LieutenantColonel
Major
Captain
Lieutenant

şeklinde bir organizasyon olduğu göz önüne alınabilir.

- Organizasyondaki herkes bir **askerdir(Soldier)** ki bu da, **Component** tipi ile ifade edilmektedir. **Composite** tipi (**CompositeSoldier**) isterse kendi içerisinde birden fazla başka **Component**(**PrimitiveSoldier** veya **CompositeSoldier** olabilir) tiplerini içerebilmelidir. Tüm askerlerin, ister **Leaf** ister **Composite** olsun uygulayacağı birde ortak operasyonu vardır: (**ExcuteOrder**). Sınıf diagramı ve uygulama kodları:

Yazılım Tasarımı ve Mimarisi



Yazılım Tasarımı ve Mimarisi

```
using System;
using
System.Collections.Generic;
namespace CompositePattern
{
          /// <summary>
         /// Askerlerin
rütbeleri
         /// </summary>
         enum Rank
          {
           General,
           Colonel,
           LieutenantColonel,
           Major,
           Captain,
           Lieutenant
          }

          /// <summary>
         /// Component sınıfı
          /// </summary>
          abstract class Soldier
          {
           protected string
_name;
           protected Rank _rank;
```

```
           public Soldier(string
name,Rank rank)
           {
            _name=name;
            _rank=rank;
           }

  public abstract void
AddSoldier(Soldier soldier);
  public abstract void
RemoveSoldier(Soldier soldier);
  public abstract void
ExecuteOrder(); // Hem Leaf hemde
Composite tipi için uygulanacak
olan fonksiyon

 }
```

Yazılım Tasarımı ve Mimarisi

```
/// <summary>
 /// Leaf class
 /// </summary>

 class PrimitiveSoldier  :Soldier{

       public PrimitiveSoldier(string name,Rank rank):base(name,rank)
       {

       }

       // Bu fonksiyonun Leaf için anlamı yoktur.
      public override void AddSoldier(Soldier soldier)
      {
            throw new NotImplementedException();
      }

      // Bu fonksiyonun Leaf için anlamı yoktur.
      public override void RemoveSoldier(Soldier soldier)
      {
           throw new NotImplementedException();
      }

      public override void ExecuteOrder()
     {
          Console.WriteLine(String.Format("{0} {1}",_rank,_name));
     }
 }
```

Yazılım Tasarımı ve Mimarisi

```
/// <summary>
        /// Composite Class
        /// </summary>

        class CompositeSoldier : Soldier{

        // Composite tip kendi içerisinde birden fazla Component tipi
        //içerebilir. Bu  tipleri bir koleksiyon içerisinde tutabilir.

         private List<Soldier> _soldiers=new List<Soldier>();

         public CompositeSoldier(string name,Rank rank) :base(name,rank)
         {

        }

        // Composite tipin altına bir Component eklemek için kullanılır
        public override void AddSoldier(Soldier soldier)
          {
                _soldiers.Add(soldier);
         }
```

Yazılım Tasarımı ve Mimarisi

```
// Composite tipin altındaki koleksiyon içerisinden bir Component
tipinin çıkartmak      için kullanılır

    public override void RemoveSoldier(Soldier soldier)
    {
        _soldiers.Remove(soldier);
    }

  // Önemli nokta. Composite tip içerisindeki bu operasyon, Composite
tipe bağlı tüm   Component'ler için gerçekleştirilir.

    public override void ExecuteOrder()
    {
        Console.WriteLine(String.Format("{0} {1}",_rank,_name));
        foreach(Soldier soldier in _soldiers)
        {
                soldier.ExecuteOrder();
        }
    }

 }
```

Yazılım Tasarımı ve Mimarisi

```
class Program
 {
        public static void Main(string[] args)
        {
            // Root oluşturulur.

            CompositeSoldier generalBurak=new CompositeSoldier("Burak",Rank.General);


            // root altına Leaf tipten nesne örnekleri eklenir.
            generalBurak.AddSoldier(new PrimitiveSoldier("Mayk",Rank.Colonel));
            generalBurak.AddSoldier(new PrimitiveSoldier("Tobiassen",Rank.Colonel));


            // Composite tipler oluşturulur.
            CompositeSoldier colonelNevi=new CompositeSoldier("Nevi", Rank.Colonel);
            CompositeSoldier lieutenantColonelZing=new CompositeSoldier("Zing",
            Rank.LieutenantColonel);


            // Composite tipe bağlı primitive tipler oluşturulur.
            lieutenantColonelZing.AddSoldier(new PrimitiveSoldier("Tomasson", Rank.Captain));
            colonelNevi.AddSoldier(lieutenantColonelZing);
            colonelNevi.AddSoldier(new PrimitiveSoldier("Mayro", Rank.LieutenantColonel));


            // Root' un altına Composite nesne örneği eklenir.
            generalBurak.AddSoldier(colonelNevi);
```

Yazılım Tasarımı ve Mimarisi

```
generalBurak.AddSoldier(new
PrimitiveSoldier("Zulu",Rank.Colonel));

    // root için ExecuteOrder operasyonu
uygulanır. Buna göre root altındaki tüm
nesneler için bu operasyon uygulanır
    generalBurak.ExecuteOrder();

    Console.ReadLine();
  }
 }
}
```

- Uygulamayı çalıştırıldığında yandaki sonuc alınır.

- Görüldüğü gibi emir modeli general üzerinden uygulandığı için organizasyonda generale bağlı olan herkese iletilebilmektedir.

KAYNAKÇA

1. C# ile Tasarım Desenleri ve Mimarileri - Ali KAYA & Engin BULUT
2. C++, Java ve C# ile UML ve Dizayn Paternleri - Aykut TAŞDELEN
3. http://www.dofactory.com/net/composite-design-pattern
4. http://www.bayramucuncu.com/adapter-tasarim-deseni-design-pattern/
5. http://www.rehabayar.net/?p=150
6. http://harunozer.com/makale/adapter_tasarim_deseni__adapter_design_pattern.htm
7. http://harunozer.com/makale/bridge_tasarim_deseni__bridge_design_pattern.htm
8. http://www.buraksenyurt.com/post/Tasarc4bım-Desenleri-Composite.aspx
9. http://ayhanugur.com/kopru-tasarim-deseni-bridge-design-pattern/
10. http://harunozer.com/makale/composite_tasarim_deseni__composite_design_pattern.htm