



HALIÇ ÜNİVERSİTESİ  
LİSANSÜSTÜ EĞİTİM ENTİTÜSÜ  
BİLGİSAYAR MÜHENDİSLİĞİ TEZLİ YÜKSEK LİSANS PROGRAMI

ASP.NET CORE PROJELERİNDE TASARIM DESENLERİ KULLANIMI VE ANALİZİ

TUTKU ÇAKIR



# İÇİNDEKİLER

■ ASP.NET Core Projesi	4
■ Model View Controller	6
■ Tasarım Desenlerine Giriş	11
■ Tasarım Deseni Türleri	12
■ Yaratılış Desenler	13
■ Yapısal Desenler	14
■ Davranışsal Desenler	15
■ AbstractFactory	16
■ Builder	22
■ Factory Method	26
■ Prototype	33
■ Singleton	37
■ Adapter	44
■ Bridge	48
■ Composite	51



# İÇİNDEKİLER

▪ Decorator	57
▪ Facade	64
▪ Flyweight	66
▪ Proxy	70
▪ Chain of Responsibility	73
▪ Command	78
▪ Interpreter	84
▪ Iterator	89
▪ Mediator	95
▪ Memento	101
▪ Observer	106
▪ State	111
▪ Strategy	118
▪ Template	123
▪ Visitor	126
▪ Kaynaklar	132

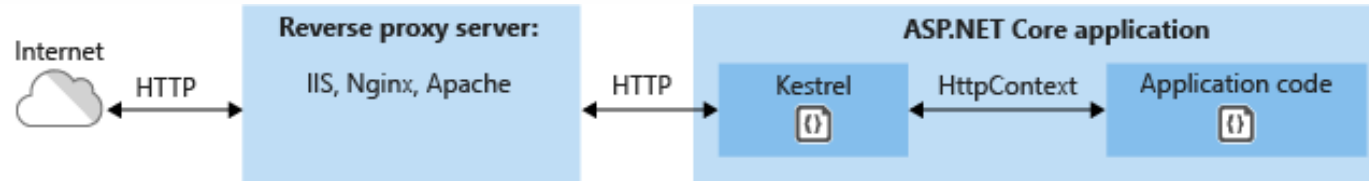
# ASP.NET Core Projesi



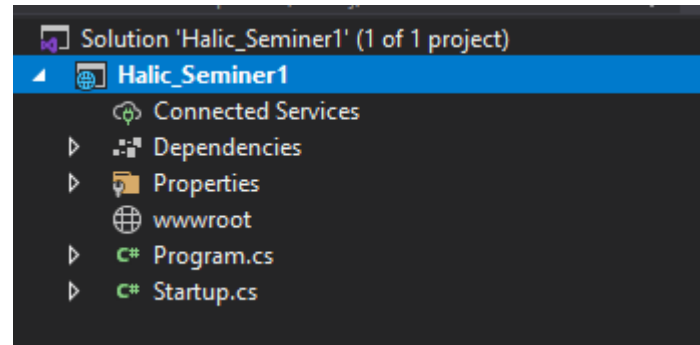
Microsoft tarafından 27 Haziran 2016 günü ASP.NET Core 1.0 olarak ilan edildi. Core sürümleri, ücretsiz ve açık kaynaklıdır. Windows, Linux ve Mac sürümleri üzerinde geliştirilebilir ve çalıştırılabilir web çerçevesidir.

10 Kasım 2020'de son versiyonu olan ASP.NET Core 5.0 yayınlandı. 2021 yılında ise ASP.NET Core 6.0 yayınlanacağı beklenmektedir.

# ASP.NET Core Projesi

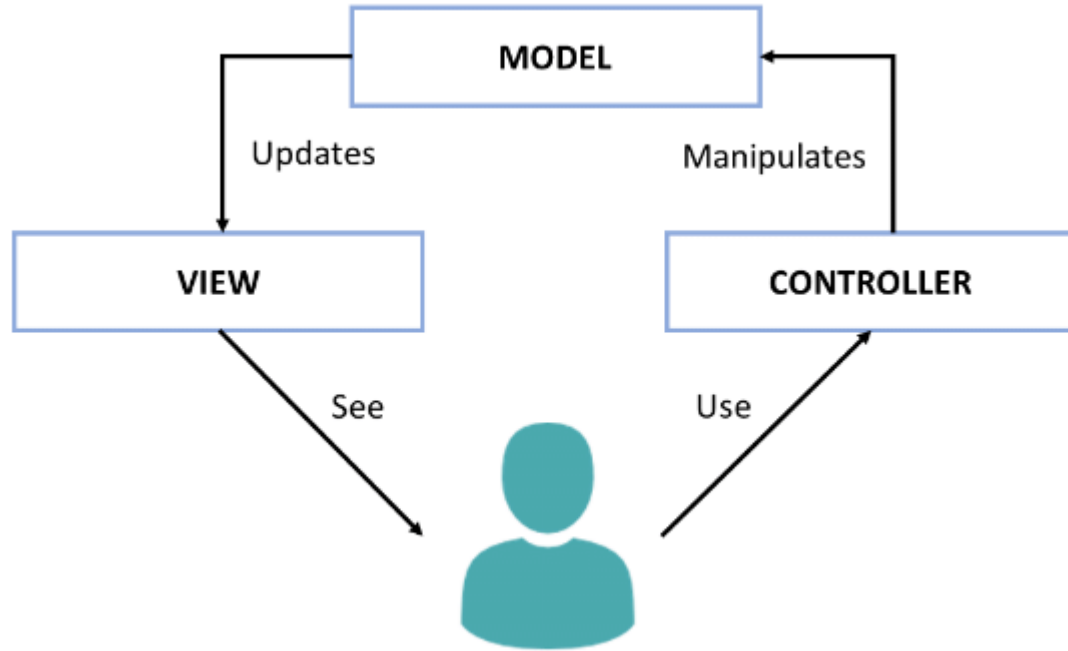


Core projesi C# konsol projesi gibi program.cs ile başlar, bağımlılık ve çeşitli ayarlar startup.cs dosyasında tanımlanır. Proje çalışırken ilk olarak IIS, Apache veya diğer temel web sunucu yapıları ile çalışır ardından Kestrel Server devreye girer. Kestrel server açık kaynaklı ve platform bağımsız, basit bir yapıda olan destekleyici bir web sunucusudur.



Boş bir Core Web Application projesinin Solution Explorer ekran görüntüsü.

# MVC: Model-View-Controller



**Model**, ASP.NET Core projelerinde entitylerin yer aldığı sınıflardır. Bu sınıflar ile veritabanına veya kullanıcı ekranına verilerimizi taşıyoruz.

**Controller**, kullanıcıdan aldığı verileri yorumlar tüm kontrollerden controller sınıfları sorumludur. Kullanıcıdan gelen istek algoritma doğrultusunda yorumlanır ve beklenen sonuç döndürülür. Sonuç herhangi bir mesaj olabileceği gibi model olarak da döndürülebilir.

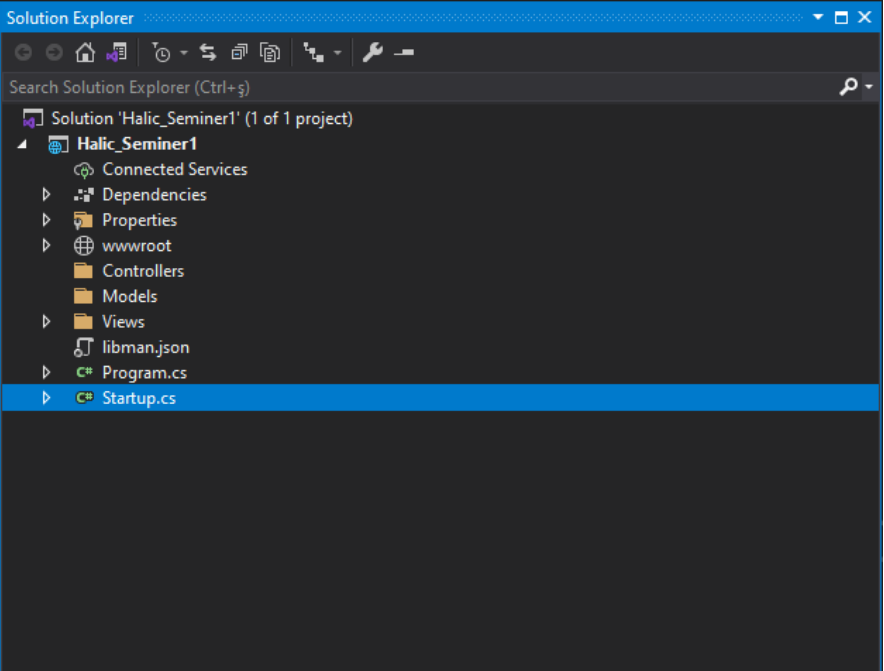
**View** ise kullanıcının karşılaştığı, html kodlarımızın yer aldığı yapıdır. Model ve Controller sınıfları cs uzantılı iken View dosyaları cshtml uzantılıdır.

# MVC: Model-View-Controller

Boş bir ASP.Net Core projesini MVC projesine çevirmek için Models, Views, Controllers klasörleri oluşturulur, ardından Startup.cs dosyasında tanımlamalar yapılır. ConfigureServices metodu içerisinde services.AddMvc(); metodu, ardından Configure metodu içerisinde app.UseMvc(); metodu çağırılarak MVC tanımlaması gerçekleştirilir.

```
1 reference
public class Startup
{
    0 references | 0 exceptions
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
    }

    0 references | 0 exceptions
    public void Configure(IApplicationBuilder app, IHostingEnvironment env)
    {
        if (env.IsDevelopment())...
        app.UseMvcWithDefaultRoute();
        app.UseStaticFiles();
        app.UseStatusCodePages();
    }
}
```



MVC projesine dönüştürülmüş Core projesi ekranı.

# MVC: Model-View-Controller

Startup.cs içerisindeki Configure metodu içerisinde çağrılan `app.UseMvcWithDefaultRoute();` metodu ile web projemiz açılış yaptığında Home/Index adresi ile açılış yapar. Varsayılan olarak MVC url yapısı `Controller=Home/Action=Index/Id?` şeklindedir. Burada Id zorunlu alan değildir. Varsayılan bu değeri değiştirebiliriz.

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    if (env.IsDevelopment()) ...
    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "HalicUni",
            template: "{Controller=HalicStudent}/{Action=Index}/{id?}"
        );
    });
    app.UseStaticFiles();
    app.UseStatusCodePages();
}
```

Core projesinde Route tanımı



# MVC: Model-View-Controller

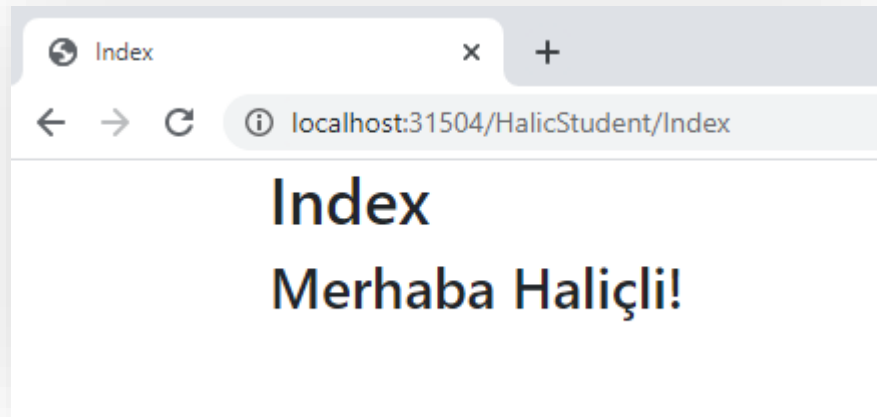
Tanımladığımız rota doğrultusunda projemizi başlatmak için View dosyalarında gerekli ayarları yaptıktan sonra Controller klasörü içerisine HalicStudentController dosyası oluşturuyoruz. Ardından Index metodu için View'da HalicStudent klasörü içerisine Index.cshtml dosyası açıyoruz.

```
0 references | 0 requests
public class HalicStudentController : Controller
{
    0 references | 0 requests | 0 exceptions
    public IActionResult Index()
    {
        return View();
    }
}
```

ASP.Net MVC NetFramework ve MVC Core projelerinde kontrol sınıflarının adı Controller ile bitmek zorundadır. Bu sınıflar Controller sınıflarından türemişlerdir.

Index metodu ise rotada belirtilen action kısmıdır. IActionResult ise Core Projesinde genel bir geri dönüş tipidir.

# MVC: Model-View-Controller



Projeyi çalıştırdığımızda varsayılan olarak karşımıza rotada tanımladığımız sayfa çıkacaktır. View dosyamıza h2 ve h3 etiketleri ile mesajımızı yazdık.

# Tasarım Desenlerine Giriş

Tasarım desenleri yazılım geliştiricilerin projelerinde sık sık karşılaştıkları problemlerin çözümü için oluşturulmuşlardır. Doğru kullanılırlarsa geliştiricilere çok zaman kazandırabilirler çünkü karmaşık problemleri çözmek için basit algoritmalar sunarlar.

Bu kapsamda tasarım desenlerinin kullanımı ile;

- Kodun daha esnek olabilmesini sağlamak,
- Kodun tekrar tekrar kullanılabilmesi,
- Sonradan yapılacak değişikliklere karşı kodun fazla etkilenmemesi hedeflenmektedir.



# Tasarım Desen Türleri

Tasarım desenleri; Yaratılış, Yapısal, Davranışsal olmak üzere 3 kategori altında yer almaktadır.

**Yaratılış Desenleri (Creational Patterns):** new keyword kullanımını yönetmektedir.

**Yapısal Desenler (Structural Patterns):** Sınıflar arası kurulan ilişkiler ile ilgilenir.

**Davranışsal Desenler (Behavioral Patterns):** Algoritmalar ve algoritmaların davranışlarının soyutlanması üzerinedir.

# Yaratılış Desenleri (Creational Patterns)

- **Abstract Factory:** İlişkisel olan birden fazla nesnenin üretimini her ürün ailesi için farklı bir ara yüz tanımlayarak sağlama amacındadır.
- **Builder:** Nesne yapısını temsilinden ayırır.
- **Factory Method:** Çeşitli türetilmiş sınıfların bir örneğini oluşturur.
- **Prototype:** Kopyalanarak veya klonlanarak yeni bir örnek oluşturulur.
- **Singleton:** Yalnızca tek bir örneğin var olabileceği yapıyı oluşturur.

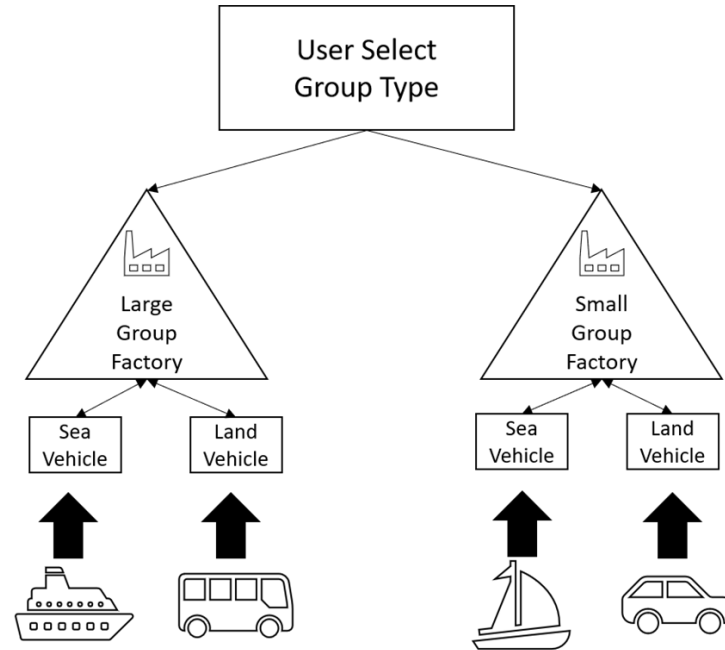


# Yapısal Desenler(Structural Patterns)

- **Adapter:** Farklı sınıfın ara yüzlerinin beraber kullanılmasını sağlar.
- **Bridge:** Bir yapıyı mevcut ara yüzden veya soyut sınıftan ayırır.
- **Composite:** Basit ve bileşik nesnelerden oluşan bir ağaç yapısı oluşturulur.
- **Decorator:** Nesnelere dinamik olarak sorumluluklar ekler.
- **Facade:** Tüm bir alt sistemi temsil eden sade yapı oluşturulur.
- **Flyweight:** Verimlilik için ortak bellek alanları kullanılan desendir.
- **Proxy:** Başka bir nesneyi temsil eden bir nesne söz konusudur.

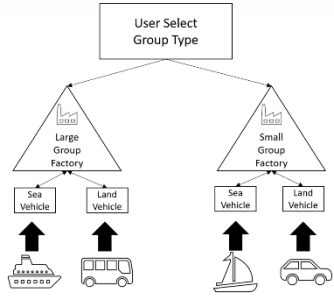
# Davranışsal Desenler(Behavioral Patterns)

- **Chain of Respon..:** Bir işi hangi sınıfın yapacağına karar veren desendir.
- **Command:** Bir komut isteğini bir nesne olarak kapsüller.
- **Interpreter:** Bir programa dil öğelerini dahil etmenin bir yoludur.
- **Iterator:** Bir koleksiyonun öğelerine sırayla erişim sağlar.
- **Mediator:** Sınıflar arası basitleştirilmiş iletişimi tanımlar.
- **Memento:** Bir nesnenin dahili durumunu yakalar ve geri yükler.
- **Observer:** Bir dizi sınıftaki değişikliği bildirmenin bir yoludur.
- **State:** Durumu değiştiğinde bir nesnenin davranışını değiştirmesi söz konusudur.
- **Strategy:** Bir iş için birden çok algoritma kullanımında tercih edilir.
- **Template Method:** Bir algoritmanın tam adımlarını bir alt sınıfı aktarır.
- **Visitor:** Değişiklik olmadan bir sınıfa yeni bir işlem tanımlar.



İlişkisel olan birden fazla nesnenin üretimini tek bir arayüz tarafından değil her ürün ailesi için farklı arayüz tanımlayarak sağlar. Kısacası, soyut fabrika tasarım modeli, müşterinin ihtiyaçlarına en uygun fabrikayı seçtiği bir fabrika fabrikasıdır.





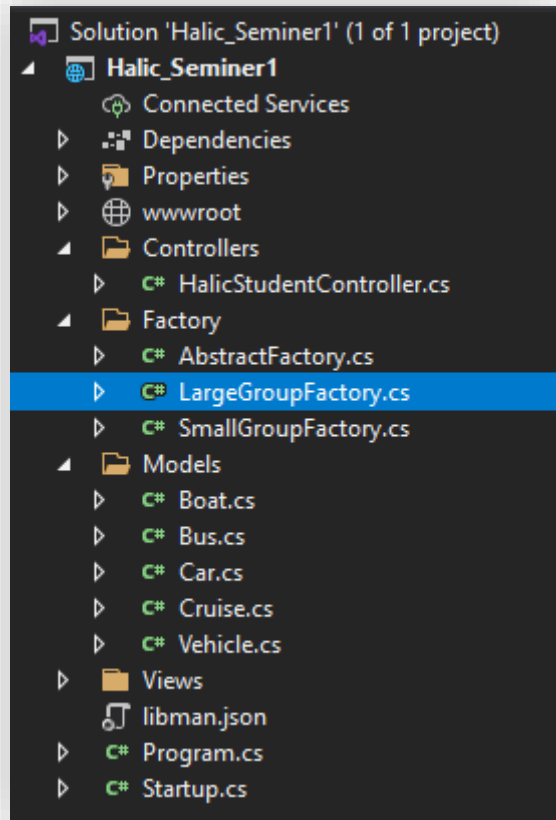
Görselde anlatılmak istendiği üzere Models klasörüne Boat, Bus, Car, Cruise sınıflarını oluşturalım bu sınıflara Vehicle soyut sınıfı ebeveyn olsun.

```
public abstract class Vehicle
{
    internal int capacity;
    2 references | 0 exceptions
    public string GetData() => this.GetType().Name;
    2 references | 0 exceptions
    public int GetCapacity() => capacity;
}
```

Vehicle soyut sınıfı GetData metodu ile bağlı sınıfın adını, GetCapacity metodu ile bağlı sınıfın capacity verisini döndürecektir.

```
2 references
public class Boat:Vehicle
{
    1 reference | 0 exceptions
    public Boat()
    {
        this.capacity = 50;
    }
}
```

Alt sınıflar olan Boat için capacity değeri 50, Bus için 60, Car için 5, Cruise için 250 olarak tanımlanmıştır.



Projenin ana dizini içerisine oluşturulan Factory klasörünün içerisine LargeGroupFactory, SmallGroupFactory ve bu sınıflara ebeveyn olan AbstractFactory soyut sınıfı oluşturulmuştur.

```
abstract class AbstractFactory
{
    3 references | 0 exceptions
    public abstract Vehicle CreateLandVehicle();
    3 references | 0 exceptions
    public abstract Vehicle CreateSeaVehicle();
}
```

AbstractFactory soyut sınıfında Vehicle tipinde CreateLandVehicle(); ve CreateSeaVehicle(); adlarında soyut metod oluşturulmuştur. AbstractFactory soyut sınıfını miras alan alt sınıflar bu metodları ezeceklerdir.

```
class LargeGroupFactory : AbstractFactory
{
    3 references | 0 exceptions
    public override Vehicle CreateLandVehicle() => new Bus();
    3 references | 0 exceptions
    public override Vehicle CreateSeaVehicle() => new Cruise();
}
```

```
class SmallGroupFactory : AbstractFactory
{
    3 references | 0 exceptions
    public override Vehicle CreateLandVehicle() => new Car();
    3 references | 0 exceptions
    public override Vehicle CreateSeaVehicle() => new Boat();
}
```

Taşıtlar boyutlarına göre farklı sınıflara tanımlanmışlardır.

Kodumuzun çalıştığını görmek için Controller ve Action alanlarına eklemeler yapalım.

```
AbstractFactory factory = null;  
0 references | 0 requests | 0 exceptions  
public IActionResult Servis(int id)  
{  
    if (id > 15)  
        factory = new LargeGroupFactory();  
    else  
        factory = new SmallGroupFactory();  
  
    var landVehicle = factory.CreateLandVehicle();  
    var seaVehicle = factory.CreateSeaVehicle();  
  
    ViewData["landVehicle"] = $"Land vehicle {landVehicle.GetData()} with capacity of: {landVehicle.GetCapacity()}";  
    ViewData["seaVehicle"] = $"Sea vehicle {seaVehicle.GetData()} with capacity of: {seaVehicle.GetCapacity()}";  
  
    return View();  
}
```

```
Servis.cshtml  x
1  @{
2      ViewData["Title"] = "Servis";
3  }
4  <h2>Servis</h2>
5  <h3>@ViewData["landVehicle"]</h3>
6  <h3>@ViewData["seaVehicle"]</h3>
```

Servis x +

localhost:31504/HalicStudent/Servis/14

## Servis

Land vehicle Car with capacity of: 5

Sea vehicle Boat with capacity of: 50

Servis x +

localhost:31504/HalicStudent/Servis/16

## Servis

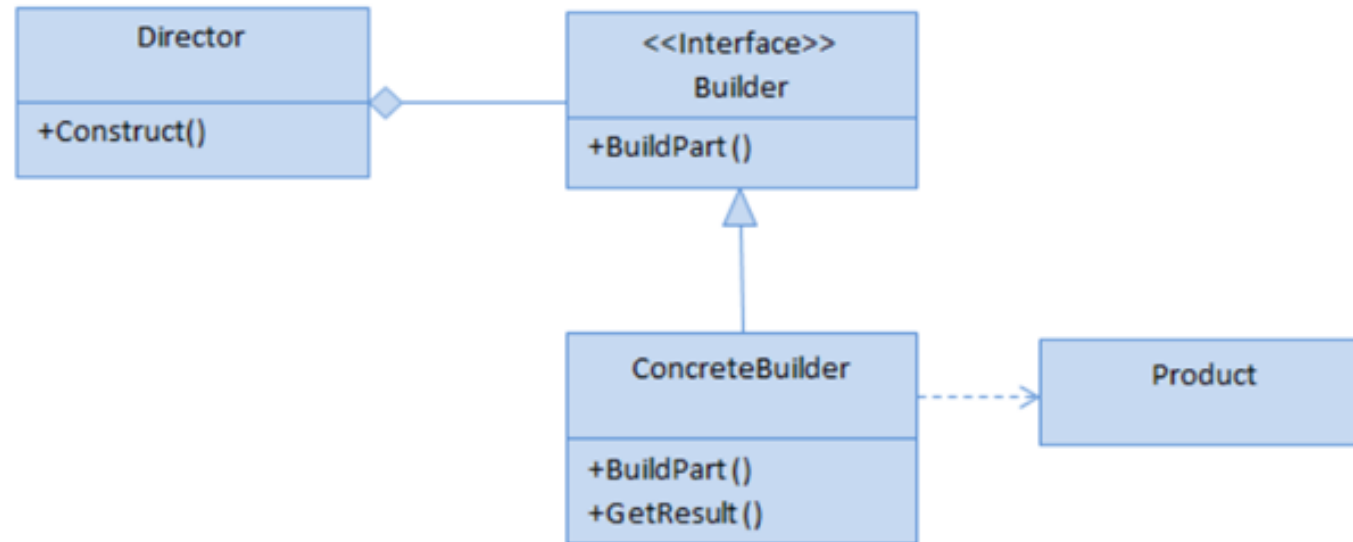
Land vehicle Bus with capacity of: 60

Sea vehicle Cruise with capacity of: 250

ViewData'yı View dosyamıza ekleyelim. Projemizi <http://localhost:PortNo/HalicStudent/Servis/14> ve ardından <http://localhost:PortNo/HalicStudent/Servis/16> adresleri ile çalıştıralım.

Özetle: Bu çalışmada 2 deniz 2 kara olmak üzere 4 taşıt kapasitesine göre ekrana yazdırıldı. URL'den gelen değer 15'in üzerinde ise yüksek kapasiteli taşıtlar, altında ise düşük kapasiteli taşıtlar ekrana yazdırıldı.

Projemiz inşa süresindeyken oluşturacağımız bazı nesnelerin üretimleri oldukça maliyetli olabilir, zamanla bu nesnelerin yapısı değişebilir. Builder tasarım deseni ile bu nesneler genişletilebilir bir hale getirilmekte ve kod karmaşıklığı minimize edilmektedir.



Builder Pattern

Model klasörümüzü ilk olarak Toy ürünümüzü ardından IToyBuilder arayüzünü, arayüzden miras alan ToyABuilder ve ToyBBuilder sınıflarını ve ToyCreator sınıfını oluşturuyoruz.

```
public class Toy
{
    public string Model { get; set; }
    public string Head { get; set; }
    public string Limbs { get; set; }
    public string Body { get; set; }
    public string Legs { get; set; }
}
```

```
public interface IToyBuilder
{
    void SetModel();
    void SetHead();
    void SetLimbs();
    void SetBody();
    void SetLegs();
    Toy GetToy();
}
```

```
public class ToyABuilder : IToyBuilder
{
    Toy toy = new Toy();
    public Toy GetToy()
    {
        return toy;
    }

    public void SetBody()
    {
        toy.Body = "Plastic";
    }

    public void SetHead()
    {
        toy.Head = "1";
    }

    public void SetLegs()
    {
        toy.Legs = "2";
    }

    public void SetLimbs()
    {
        toy.Limbs = "4";
    }

    public void SetModel()
    {
        toy.Model = "TOY A";
    }
}
```

```
public class ToyCreator
{
    private IToyBuilder _toyBuilder;

    public ToyCreator(IToyBuilder toyBuilder)
    {
        _toyBuilder = toyBuilder;
    }

    public void CreateToy()
    {
        _toyBuilder.SetModel();
        _toyBuilder.SetHead();
        _toyBuilder.SetLimbs();
        _toyBuilder.SetBody();
        _toyBuilder.SetLegs();
    }

    public Toy GetToy() =>
        _toyBuilder.GetToy();
}
```

ToyABuilder ve ToyBBuilder sınıfları IToyBuilder arayüzünden miras aldığı için arayüzde belirtilen metotları ezmektedir.

```
public IActionResult Index()
{
    var toyACreator = new ToyCreator(new ToyABuilder());
    toyACreator.CreateToy();

    var toyBCreator = new ToyCreator(new ToyBBuilder());
    toyBCreator.CreateToy();

    List<Toy> model = new List<Toy>();
    model.Add(toyACreator.GetToy());
    model.Add(toyBCreator.GetToy());

    return View(model);
}
```

Models klasöründe tanımladığımız ToyCreator sınıfının başlatıcı metodu içerisine ToyABuilder ve ToyBBuilder sınıflarını gönderiyoruz. Toy oluşumunu sağladıktan sonra elde edilen değerleri View'da basabilmek için List koleksiyonuna gönderiyoruz.

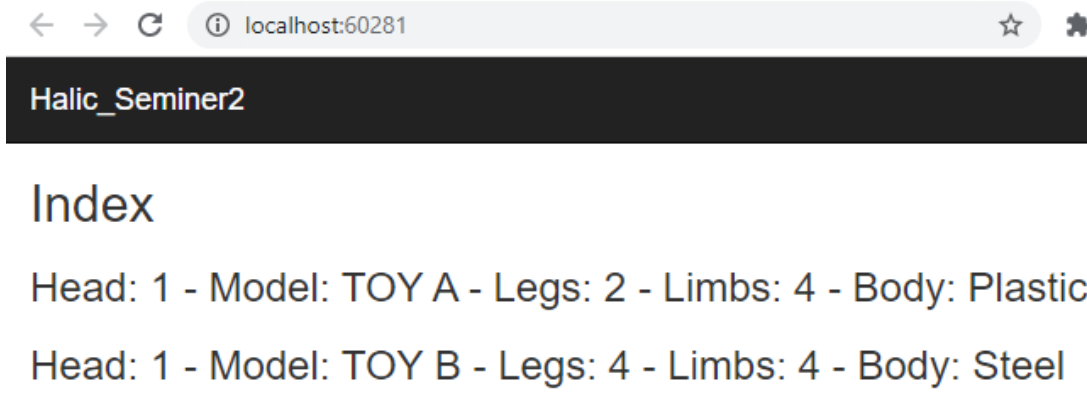
Action adını sağ tıklayıp "Add View" ile view ekliyoruz.



```
@model List<Toy>
@{
    ViewData["Title"] = "Index";
}

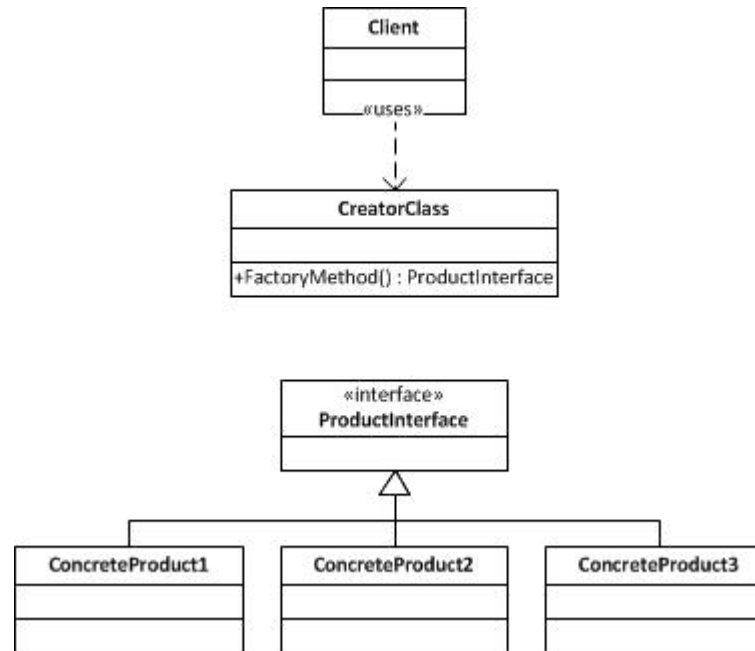
<h2>Index</h2>

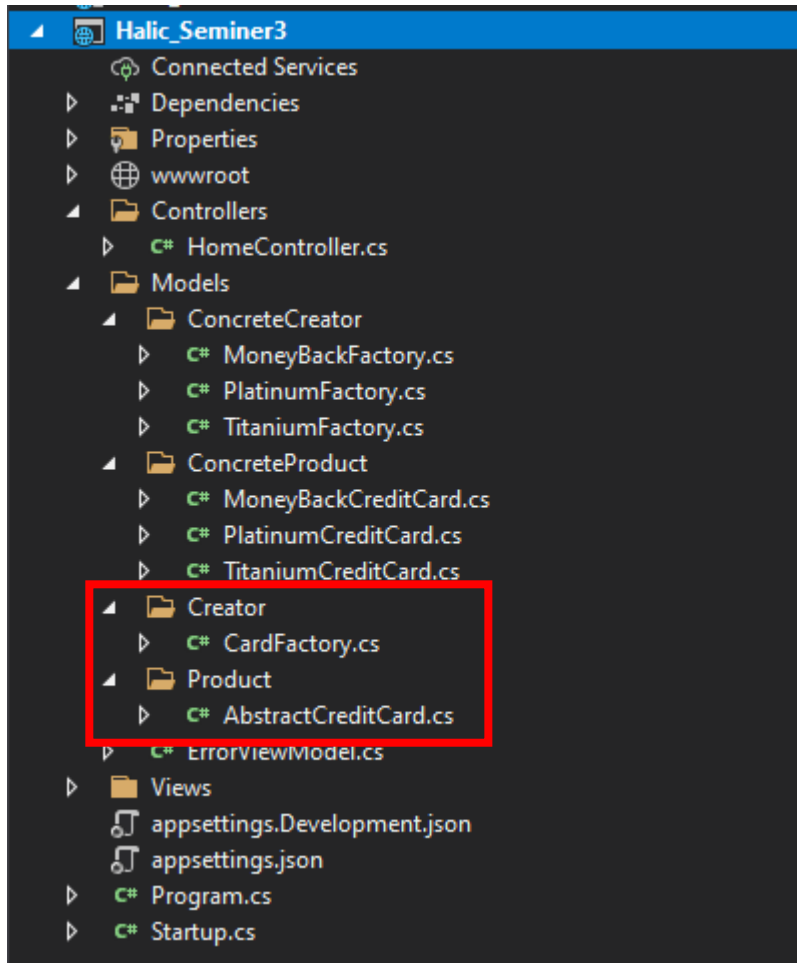
@foreach (var item in Model)
{
    <h3>Head: @item.Head - Model: @item.Model - Legs: @item.Legs - Limbs: @item.Limbs - Body: @item.Body</h3>
}
```



Koleksiyon içerisindeki tüm Toy nesnelerini ekrana foreach döngüsü ile yazdırıyoruz.

Oluşturmak istediğimiz sınıfın kendisinden bir örnek istemek yerine Factory Metod deseni sayesinde tek bir örnek (instance) üzerinden gerekli nesnenin üretilmesini sağlar.





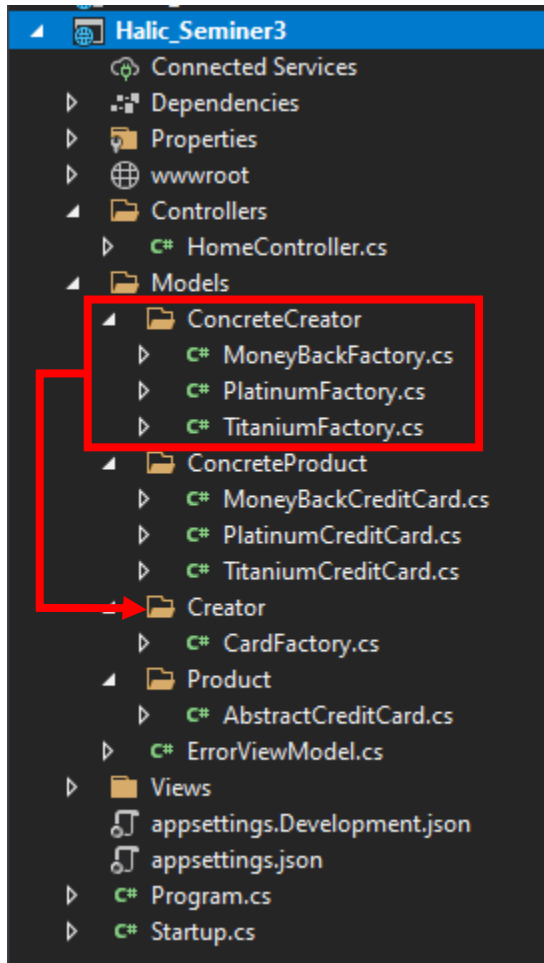
Bu mimaride Abstract, Concrete, Creator sınıfları için ayrıca klasör oluşturulabilirdi; ancak projemizde bu klasörleri daha da ayırarak models klasörü içerisinde oluşturduk.

Kartımızın özelliklerinin yer aldığı sınıf AbstractCreditCard sınıfımızdır. Kart türleri bu sınıftan miras edinerek, ilgili metodları ve propertyleri ezeceklerdir.

```
abstract class AbstractCreditCard
{
    public abstract string CardType { get; }
    public abstract int CreditLimit { get; set; }
    public abstract int AnnualCharge { get; set; }
}
```

Uygulamamızı çalıştırdığımızda ilk tetiklenecek sınıf ise CardFactory sınıfımızdır. Bu sınıf üzerinden AbstractCreditCard sınıfımız çalışacaktır.

```
abstract class CardFactory
{
    public abstract AbstractCreditCard GetCreditCard();
}
```

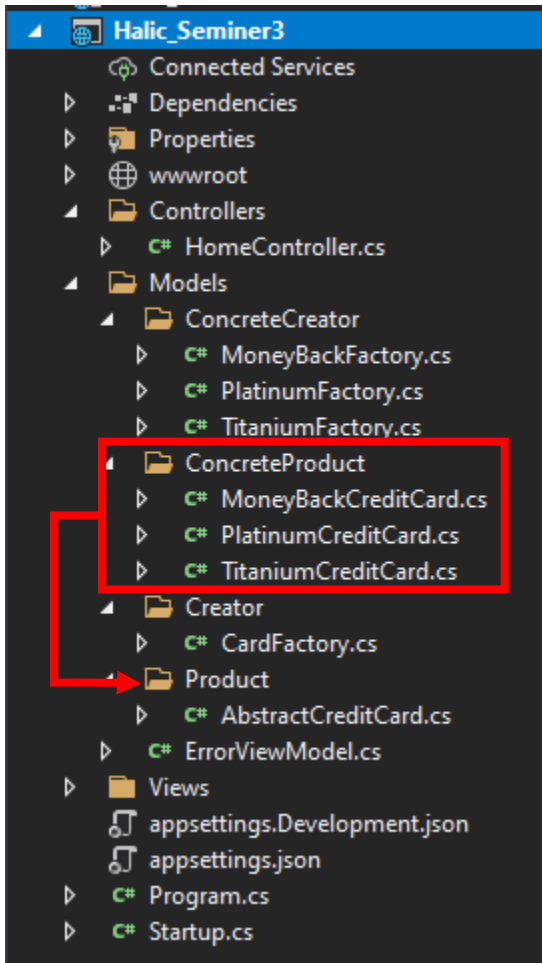


CardFactory Creator sınıfından miras alan kart çeşitlerini oluşturalım. Turuncu renkle belirtilen sınıf adları her sınıf için farklı şekilde oluşturulacaktır. Bu örnekte MoneyBack kullanıldı.

```
class MoneyBackFactory : CardFactory
{
    private int _creditLimit;
    private int _annualCharge;

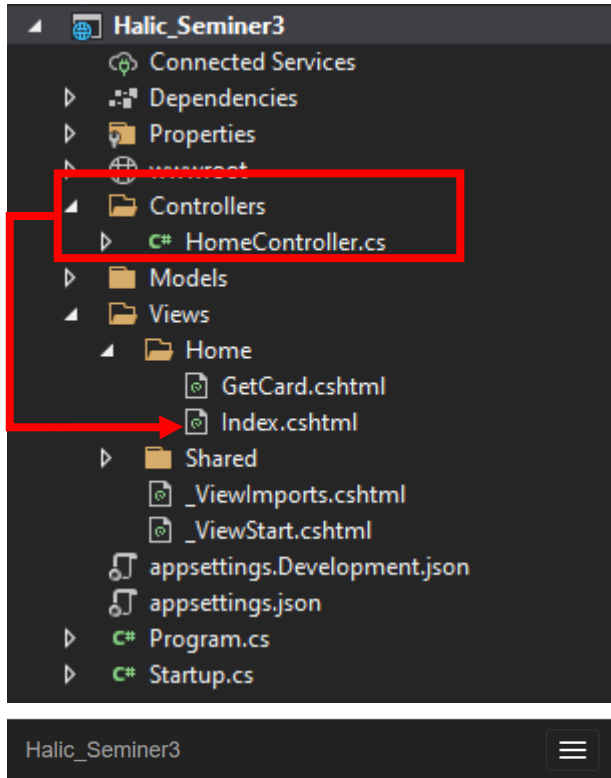
    public MoneyBackFactory(int creditLimit, int annualCharge)
    {
        _creditLimit = creditLimit;
        _annualCharge = annualCharge;
    }

    public override AbstractCreditCard GetCreditCard() =>
        new MoneyBackCreditCard(_creditLimit, _annualCharge);
}
```



AbstractCreditCard Product sınıfından miras alan kart çeşitlerini oluşturalım. Turuncu renkle belirtilen sınıf adları her sınıf için farklı şekilde oluşturulacaktır. Bu örnekte MoneyBack kullanıldı. Bu sınıfta ürün için isim verildi. (CardType)

```
class MoneyBackCreditCard : AbstractCreditCard
{
    private readonly string _cardType;
    private int _creditLimit;
    private int _annualCharge;
    public MoneyBackCreditCard(int creditLimit, int annualCharge)
    {
        _cardType = "MoneyBack";
        _creditLimit = creditLimit;
        _annualCharge = annualCharge;
    }
    public override string CardType => _cardType;
    public override int CreditLimit { get => _creditLimit;
        set
        {
            _creditLimit = value;
        }
    }
    public override int AnnualCharge { get => _annualCharge;
        set
        {
            _annualCharge = value;
        }
    }
}
```



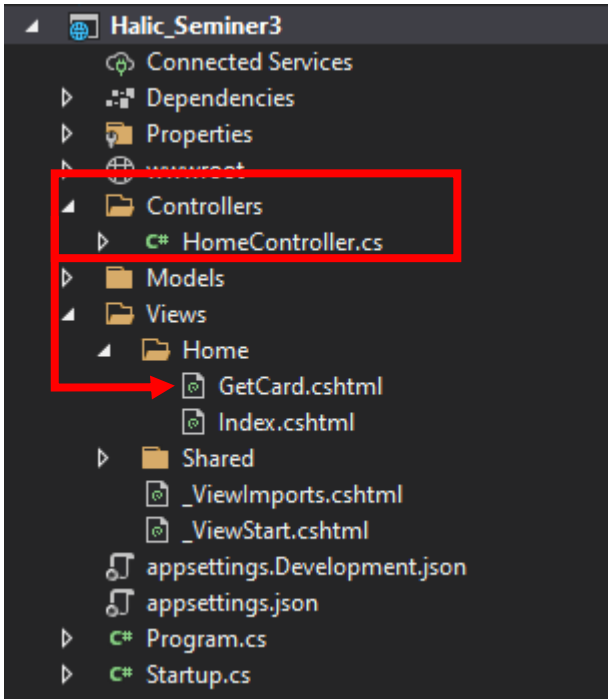
Bu projede web sayfamız Home/Index adresi ile açılacaktır. Bu yüzden HomeController dosyasını oluşturduktan sonra Index action metodunda oynama yapmadan view oluşturduk.

```
public IActionResult Index() => View();
```

Index.cshtml View dosyamızın kodları:

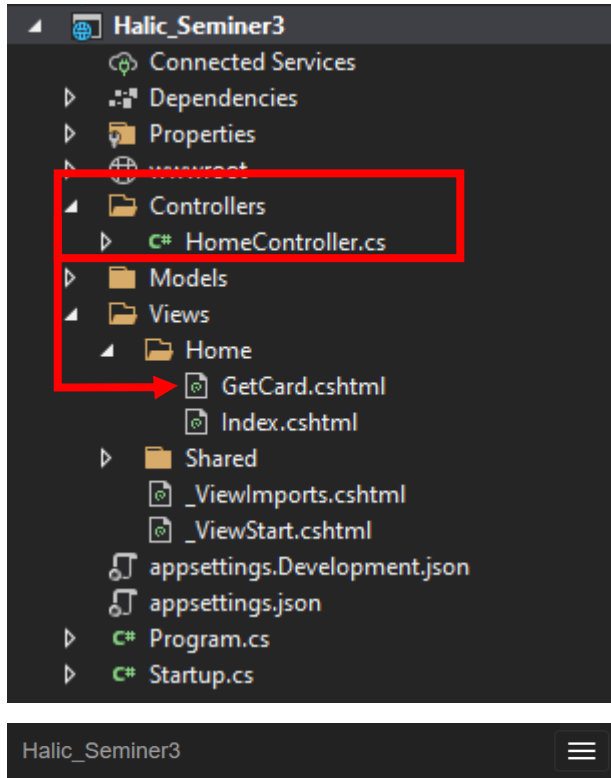
```
@{
    ViewData["Title"] = "Home Page";
}
<div class="row" style="margin-top:40px;">
    <div class="col-lg-4 col-lg-offset-4">
        <form asp-action="GetCard" method="post">
            <div class="form-group">
                <select class="form-control" name="cardName">
                    <option value="moneyback">Moneyback</option>
                    <option value="titanium">Titanium</option>
                    <option value="platinum">Platinum</option>
                </select>
            </div>
            <button class="btn btn-primary btn-block">Get Credit Card</button>
        </form>
    </div>
</div>
```

Seçilen kart adı ile bulunan sonuçlar Home/GetCard adresi üzerinden görüntülenecektir.



Index action'dan gönderilen (post) formdan okunan kart adına göre switch karar yapıyla kart ürünümüzü oluşturduk ve View'da görüntülemek için ViewBag'lere yükledik.

```
CardFactory factory = null;
[HttpPost]
public IActionResult GetCard(string cardName)
{
    switch (cardName)
    {
        case "moneyback":
            factory = new MoneyBackFactory(50000, 0);
            break;
        case "titanium":
            factory = new TitaniumFactory(100000, 500);
            break;
        case "platinum":
            factory = new PlatinumFactory(500000, 1000);
            break;
        default:
            break;
    }
    AbstractCreditCard creditCard = factory.GetCreditCard();
    ViewBag.CardType = creditCard.CardType;
    ViewBag.CreditLimit = creditCard.CreditLimit.ToString();
    ViewBag.AnnualCharge = creditCard.AnnualCharge.ToString();
    return View();
}
```



## GetCard.cshtml

```
@{
    ViewData["Title"] = "GetCard";
}
<h2>Card Details</h2>
<h3>Card Type: @ViewBag.CardType</h3>
<h3>Card Credi Limit: @ViewBag.CreditLimit</h3>
<h3>Card Annual Charge: @ViewBag.AnnualCharge</h3>
```

Bu örnekte soyut sınıfı kullanan bir creator sınıfından kart sınıfları ürettik, kart sınıfları soyut sınıftaki metodları ezerek taşıdıkları özelliklere veri atanmasını sağladılar. Kart türleri index sayfamıza list olarak gönderildi. Seçilen kart switch case ile new edilerek, soyut sınıfımızın örneğine atandı. O örnek üzerinden de veriler GetCard sayfasına gönderildi. Burada veri ilgili sayfaya soyut sınıf üzerinden gönderilmiştir.

### Card Details

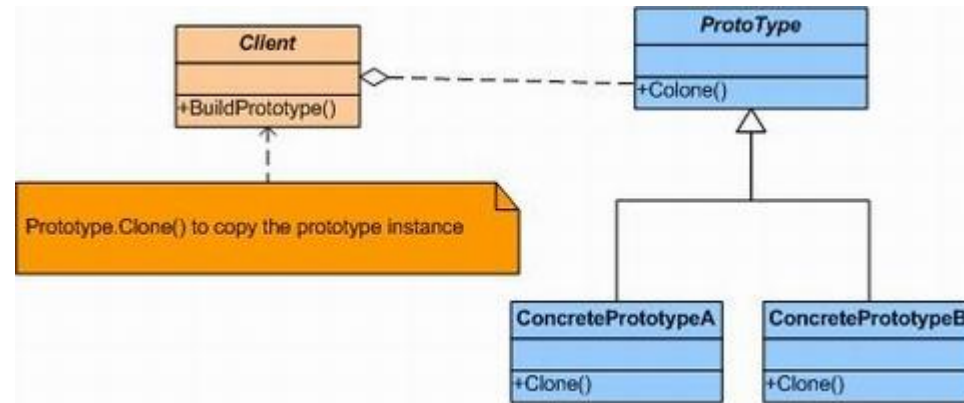
Card Type: Titanium

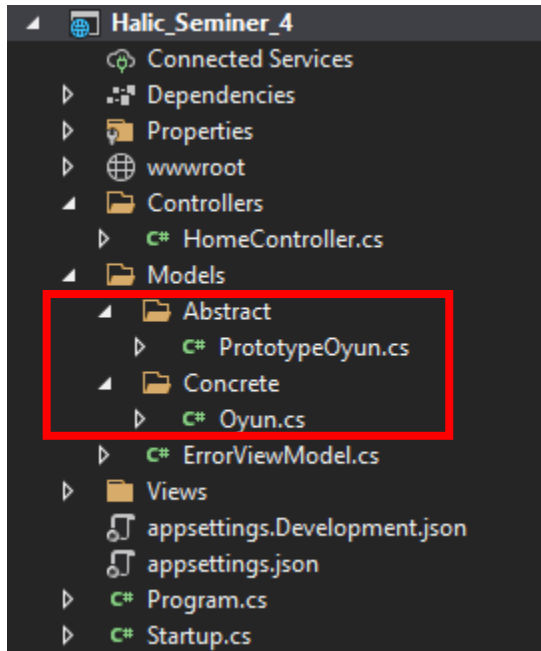
Card Credi Limit: 100000

Card Annual Charge: 500



Bu modelde amaç, bir nesne klonu oluşturmaktır; başka bir deyişle mevcut örnekleri kopyalarak yeni bir örnek oluşturmamıza izin verir. Bu da ilgili nesne üzerinden aynı tipte başka bir nesnenin hızlı üretilmesini sağlar. Üretilmesi maliyetli nesneler, new anahtar sözcüğü kullanılmadan oluşturulmuş olur. Derin kopyalama ile birebir kopyalama yapılır.





Models klasörü içerisinde Abstract ve Concrete klasörlerini açtıktan sonra Abstract klasörü içerisine PrototypeOyun ve Concrete klasörü içerisine Oyun sınıflarını oluşturduk.

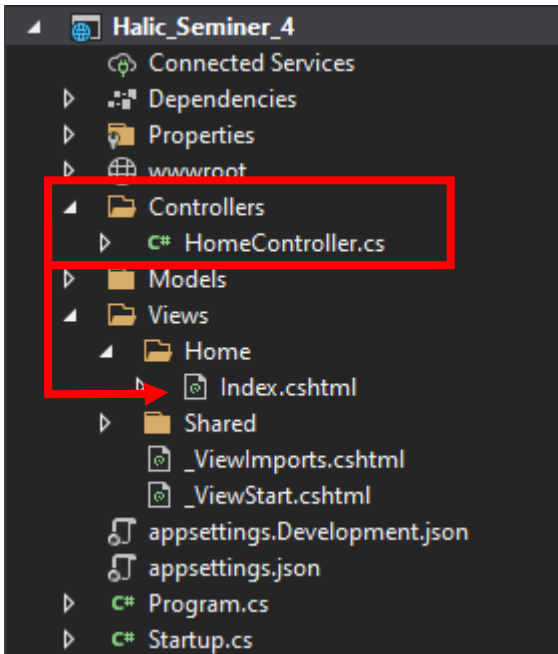
PrototypeOyun.cs Kodu:

```
abstract class PrototypeOyun
{
    public abstract PrototypeOyun Clone();
}
```

Oyun.cs Kodu:

```
class Oyun : PrototypeOyun
{
    public int OyunID { get; set; }
    public string OyunAdi { get; set; }
    public string OyunTuru { get; set; }
    public bool Durum { get; set; }
    public Oyun(int OyunID, string OyunAdi, string OyunTuru, bool Durum)
    {
        this.OyunID = OyunID;
        this.OyunAdi = OyunAdi;
        this.OyunTuru = OyunTuru;
        this.Durum = Durum;
    }

    public override PrototypeOyun Clone() => this.MemberwiseClone() as PrototypeOyun;
}
```



Yapımızı HomeController'ın Index action'ında test ediyoruz ve sonucu View'a gönderiyoruz.

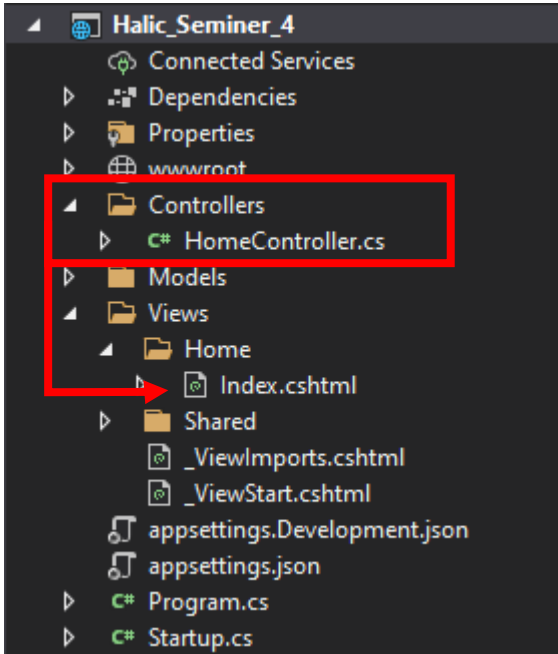
HomeController/Index:

```
public IActionResult Index()
{
    Oyun o1 = new Oyun(3, "Oyun Adı", "Oyun Türü", true);
    Oyun o2 = (Oyun)o1.Clone();

    if (o1.Equals(o2))
        ViewBag.durum = "Eşit";
    else
        ViewBag.durum = "Eşit Değil";

    if (o1.OyunAdi==o2.OyunAdi)
        ViewBag.durum2 = "Eşit";
    else
        ViewBag.durum2 = "Eşit Değil";

    return View();
}
```



Yapımızı HomeController'ın Index action'ında test ediyoruz ve sonucu View'a gönderiyoruz.

Index.cshtml

```
@{
    ViewData["Title"] = "Home Page";
}
```

<h3>Farklı kopya nesneler olduğundan eşit değil döner: @ViewBag.durum </h3>

<h3>Nesne elemanlarını karşılaştırdığımız için eşit çıkar: @ViewBag.durum2 </h3>

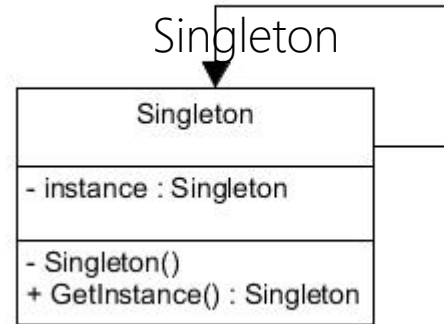
Halic\_Seminer\_4 Home

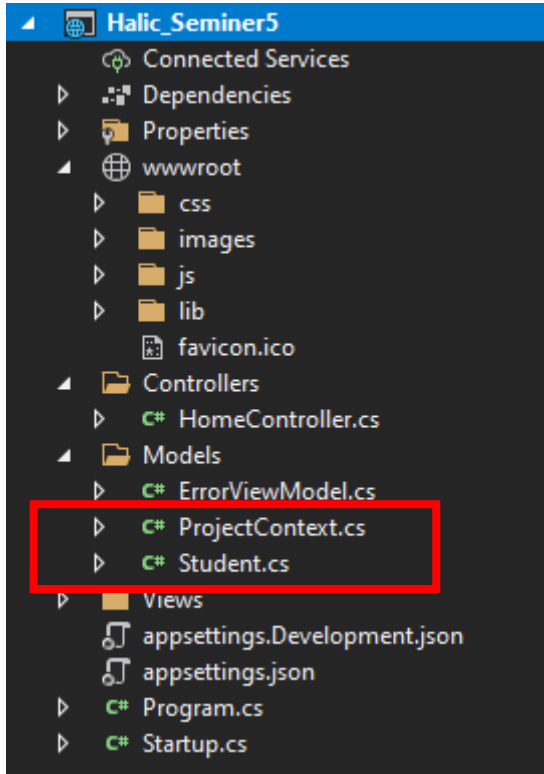
Farklı kopya nesneler olduğundan eşit değil döner: Eşit Değil

Nesne elemanların değerlerini karşılaştırdığımız için eşit çıkar: Eşit

© 2021 - Halic\_Seminer\_4

Singleton desende bir sınıfın örneği yalnızca bir kez oluşturulabilir. Her çağrıldığında o oluşturulmuş olan örnek döndürülür. Creational Patternler arasında en sık kullanılan tasarım desendir.



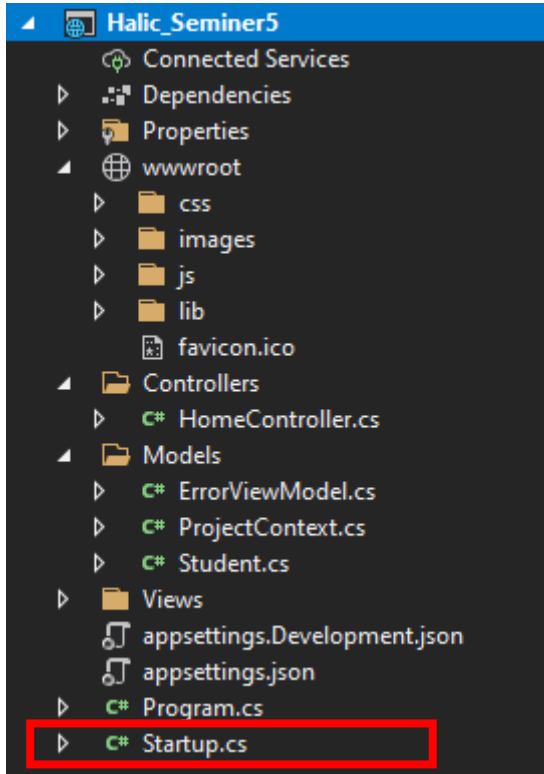


Singleton deseni ile oluşturacağımız örnekte gerçek hayatta kullanılan en basit projelerden birini gerçekleştirelim. İlk olarak Models klasörü içerisine Student sınıfı oluşturalım.

```
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Surname { get; set; }
    public string Phone { get; set; }
    public string Address { get; set; }
    public string District { get; set; }
    public string City { get; set; }
}
```

Entity Framework kullanacağımızdan veritabanı bağlantısı ve ilgili sınıfı çağırma işlemlerini gerçekleştireceğimiz DbContext'ten miras alan ProjectContext sınıfımızı oluşturalım.

```
public class ProjectContext:DbContext
{
    public ProjectContext(DbContextOptions<ProjectContext> options) : base(options)
    { }
    public DbSet<Student> Students { get; set; }
}
```



Startup.cs dosyamızda ProjectContext ile ilgili ayarları gerçekleştirelim. Bu ayarlara servis yaşam ömrü olarak Singleton'ı atayalım. Asp.Net Core projelerinde bağımlılıklar startup.cs üzerinde gerçekleştirilir.

```
services.AddDbContext<ProjectContext>(options =>  
options.UseSqlServer("server=.;database=HalicProje1;uid=sa;pwd=VeriTabaniParolasi;"),  
ServiceLifetime.Singleton, ServiceLifetime.Singleton);
```

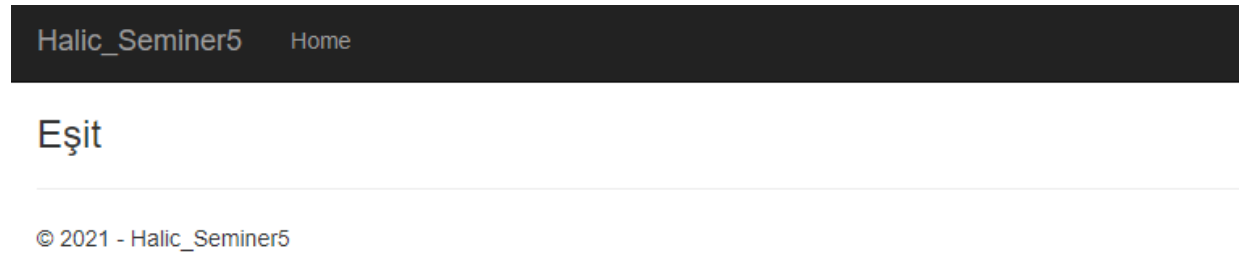
Aynı nesnenin çağrıldığını görebilmek için HomeController dosyasında 2 kez context oluşturalım ve Index metodunda contextleri karşılaştırarak elde edilen bilgiyi View'a gönderelim.

```
ProjectContext context;  
ProjectContext context2;  
public HomeController(ProjectContext _context, ProjectContext _context2)  
{  
    context = _context;  
    context2 = _context2;  
}  
  
public IActionResult Index()  
{  
    if (context == context2  
        ViewBag.durum = "Eşit";  
    else  
        ViewBag.durum = "Eşit değil";  
    return View();  
}
```



# Singleton

View/Home/Index.cshtml içerisine `<h3>@ViewBag.durum</h3>` kodunu ekleyerek projeyi başlatalım.





Asp.Net Core'da singleton uygulanacak DbContext için işlem bu kadar basitti. Şimdi farklı bir örnek olan Repository sınıfı için Singleton deseni oluşturalım.

```
public class Tools
{
    private static DbContext _dbInstance;

    public static DbContext DbInstance
    {
        get
        {
            if (_dbInstance == null)
                _dbInstance = new DbContext();
            return _dbInstance;
        }
    }
}
```

Bu kod ile DbContext nesnesi için örnek alınmak istendiğinde, mevcut bir örnek var ise o örneği geri döndürecek; daha önce örnek alınmadıysa yeni örnek oluşturulacak. Böylece DbContext için bir nesne oluşturulmuş ve her çağrıldığında o nesne döndürülmüş olacaktır.



```
public interface IRepository<T> where T : class
{
    List<T> SelectAll();
    List<T> SearchList(Expression<Func<T, bool>> predicate);
    T SelectById(int Id);
    T SearchEntity(Expression<Func<T, bool>> predicate);
    ResultModel<T> Insert(T item);
    ResultModel<T> Update(T item);
    ResultModel<bool> Delete(int Id);
}
```

IRepository arayüzü en çok tercih edilen desenlerden biridir. Genel olarak Singleton'dan context alınır, IRepository'yi implement eden Repository sınıfıyla tüm veri tabanı işlemleri tek bir nesne üzerinden gerçekleştirilir. Repository ile UnitOfWork deseni de kullanılır bu desenler birbirlerini tamamlar. Bu örneğimizin girişini Singleton ile gerçekleştirdik, devamında sadece Repository örneği kodlanmıştır.

```
public class BaseRepository<T> : IRepository<T> where T : class
{
    internal readonly DataContext _db;
    public BaseRepository()
    {
        _db = Tools.DbInstance;
    }
    public List<T> SelectAll()
    {
        return _db.Set<T>().ToList();
    }

    public List<T> SearchList(Expression<Func<T, bool>> predicate)
    {
        return _db.Set<T>().Where(predicate).ToList();
    }

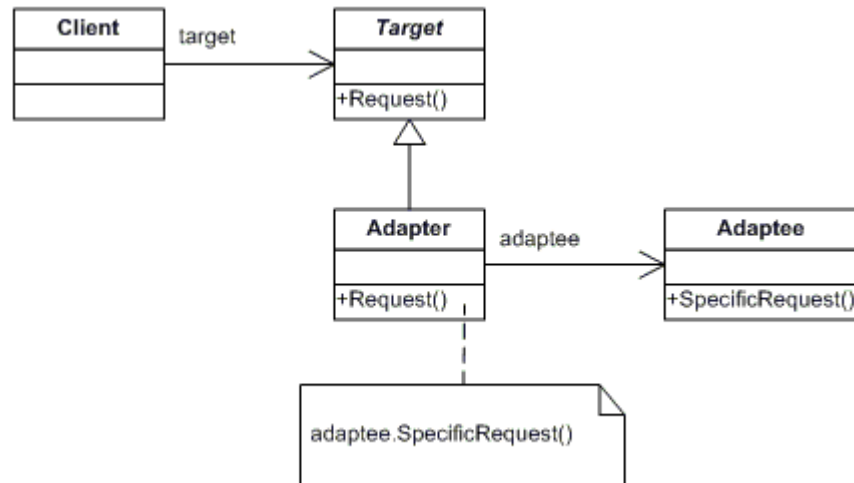
    // Diğer ezilen kodlar sunum için silinmiştir.
}
```

BaseRepository sınıfı, IRepository arayüzünden implement edilmiştir. IRepository'de belirtilen komutları ezmek zorundadır. Projede gerçekleştirilen işlemlerin döndüğü iskelet sınıftır.

Singleton desenini uyarladığımız Tools sınıfının içerisinde oluşturulmuş olan DbInstance metodu bu sınıfın başlatıcı metodunda çağırılmıştır. Çağrılan context üzerinden veritabanında işlemler gerçekleştirilmesi için metodlar oluşturulmuştur. (IRepository'den gelen ezilen metotlar.)

Adapter Design Pattern;

- İki uyumsuz interface'i beraber kullanmamızı sağlar,
- Yazılmış olan kodların değiştirilmesi gerekmez.



Sağ taraftaki görsel görüldüğü üzere farklı coğrafyaların kullandığı fişler için oluşturulmuş olan her coğrafyanın yapısına uyan bir fiş tasarımı var buna da adaptör adı verilmiş.



Senaryoda kriptolama yapan 2 sınıf mevcut arayüz yapısına uyarken, 3. sınıf uymamaktadır. Ek bir adaptör sınıfı ile 3.sınıfın da arayüzümüze uygun hareket etmesini sağlıyoruz.

```
public interface ICrypt
{
    string encrypt(String text);
    string decrypt(String text);
}
```

ICrypt arayüzümüze encrypt ve decrypt metodları tanımlanmıştır.

```
public class CryptA : ICrypt
{
    public string encrypt(string text) => text+ " CryptA_EnCoded";
    public string decrypt(string text) => text.Substring(0, text.Length - 15);
}
```

CryptA ve CryptB sınıfları ICrypt arayüzünü implement ederek metodları ezmişlerdir.

```
public class CryptB : ICrypt
{
    public string encrypt(string text) => text + " CryptB_EnCoded";
    public string decrypt(string text) => text.Substring(0, text.Length - 15);
}
```

```
public class CryptX
{
    public string textToCode(String text) => text + " CryptX_EnCoded";
    public string codeToText(String text) => text.Substring(0, text.Length - 15);
}
```

Farklı zamanda kodlanmış olan CryptX sınıfı ICrypt arayüzünün yapısına uymamaktadır. SOLID prensiplerine göre çalışan sınıf değiştirilmemelidir. Bu yüzden adapter sınıfı oluşturarak CryptX'in metodlarını arayüzümüze uygun hale getireceğiz.

```
public class CryptXAdapter : ICrypt
{
    private CryptX _cryptx;
    public CryptXAdapter(CryptX cryptx)
    {
        _cryptx = cryptx;
    }
    public string encrypt(string text) => _cryptx.textToCode(text);
    public string decrypt(string text) => _cryptx.codeToText(text);
}
```

Arayüzümüze uygun Adapter sınıfı oluşturmak için gerekli implemantasyonu sağladıktan sonra ilgili sınıfı, başlatıcı metod içerisinde örnekledik. Böylece CryptX sınıfının metodları üzerinden encode ve decode işlemleri gerçekleştirdik.

```
public ActionResult Index()
{
    ICrypt crypt = new CryptA();
    ViewBag.kriptoA= crypt.encrypt("Haliç Üniversitesi");

    crypt = new CryptB();
    ViewBag.kriptoB = crypt.encrypt("Bilgisayar Mühendisliği");

    CryptX kriptoX = new CryptX();
    crypt = new CryptXAdapter(kriptoX);
    ViewBag.kriptoX = crypt.encrypt("Tutku ÇAKIR");
    return View();
}
```

HomeController/Index metodunda ICrypt arayüzünü kullanarak onu implement eden sınıfları örnekledik ve encode metodlarını kullanarak verdiğimiz metinlerin şifrlenmesini sağladık. Dönen sonucu da View üzerinden ekrana yazdırmak için ViewBag'ler üzerine atama yaptık.

```
@{
    ViewBag.Title = "Index";
}
<h2>Index</h2>
<h5>@ViewBag.kriptoA</h5>
<h5>@ViewBag.kriptoB</h5>
<h5>@ViewBag.kriptoX</h5>
```

HomeController'ın Index aksiyonundan gelen ViewBag verileri localhost:portno/Home/Index adresinde aşağıdaki gibi görüntülenmektedir. Bu projede arayüze uymayan sınıf arayüz kontrolü altına alınmıştır.

## Index

Haliç Üniversitesi CryptA\_EnCoded

Bilgisayar Mühendisliği CryptB\_EnCoded

Tutku ÇAKIR CryptX\_EnCoded

© 2021 - My ASP.NET Application

Bu tasarım deseni Adapter Design Pattern'in aksine bir sınıfı veya yapıyı mevcut ara yüzden veya soyut sınıftan ayırma amacındadır.

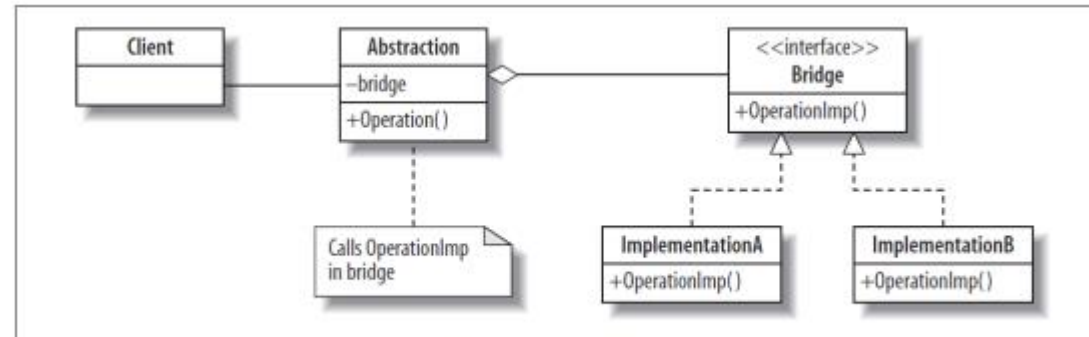
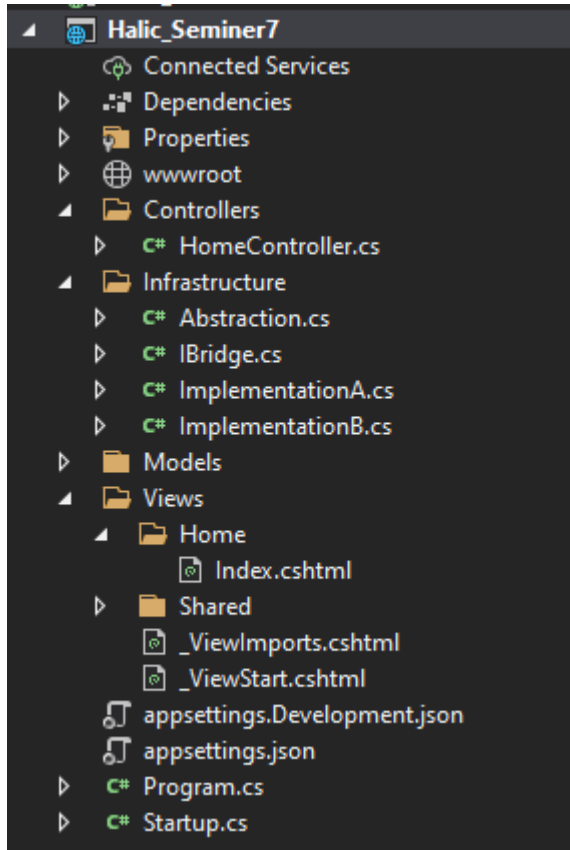


Figure Bridge pattern UML diagram



Infrastructure klasörü oluşturup içerisine IBridge arayüzünü, Abstraction, ImplementationA, ImplementationB sınıflarını oluşturuyoruz.



```
public interface IBridge
{
    string OperationImp();
}
```

```
public class ImplementationA : IBridge
{
    public string OperationImp() => "Implementation A";
}
```

```
public class ImplementationB : IBridge
{
    public string OperationImp() => "Implementation B";
}
```

ImplementationA ve ImplementationB sınıfları IBridge arayüzünü implement ettiğinden string OperationImp() metodunu oluşturmak zorundadır.

```
public class Abstraction
{
    private IBridge bridge;
    public Abstraction(IBridge _bridge)
    {
        bridge = _bridge;
    }

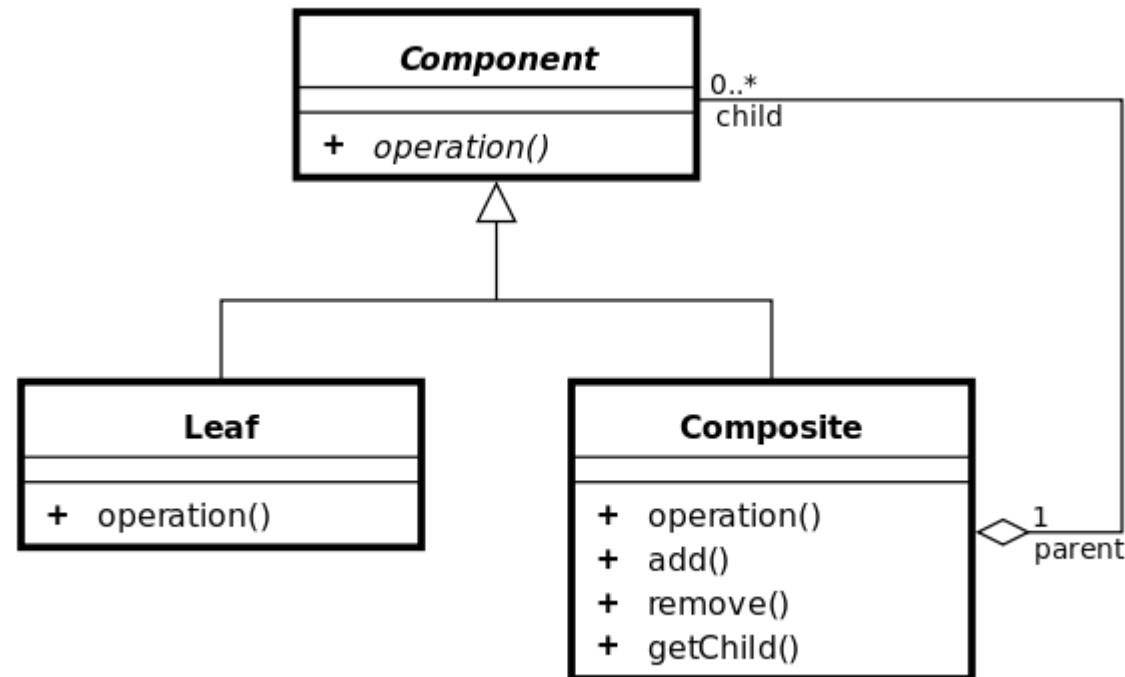
    public string Operation()
    {
        return "Abstraction <> " + bridge.OperationImp();
    }
}
```

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        ViewBag.ImpA = new Abstraction(new ImplementationA()).Operation();
        ViewBag.ImpB = new Abstraction(new ImplementationB()).Operation();
        return View();
    }
}
```

```
@{
    ViewData["Title"] = "Home Page";
}
<h3>Bridge Pattern</h3>
<h5>@ViewBag.ImpA</h5>
<h5>@ViewBag.ImpB</h5>
```

Abstraction sınıfı IBridge arayüzünü örneklemiş, Operation metodu ile arayüzün OperationImp metoduna erişmiştir. Gerçekleşen erişimle HomeController sınıfının Index aksiyonunda örneklenen ImplementationA ve ImplementationB sınıflarının döndürdüğü mesajlara erişim sağlanmıştır.

Basit ve bileşik nesnelerden oluşan bir ağaç yapısıdır. Tekil component ve birbirinden farklı componentler grubunun hiyerarşik bir yapıda benzer şekilde hareket etmesini yani kendi içlerinde birbirlerinden farklı olan bir grup nesnenin sanki tek bir bütün nesneymiş gibi kullanılmasını bileşik kalıp sağlar.



```
public enum Rank
{
    General,
    Colonel,
    LieutenantColonel,
    Major,
    Captain,
    Lieutenant
}
```

```
public abstract class AbstractSoldier
{
    protected string _name;
    protected Rank _rank;
    public AbstractSoldier(string name, Rank rank)
    {
        _name = name;
        _rank = rank;
    }
    public abstract void AddSoldier(AbstractSoldier soldier);
    public abstract void RemoveSoldier(AbstractSoldier soldier);
    public abstract string ExecuteOrder();
}
```

Tanımda hiyerarşiden bahsedilmişti. Askeri rütbeler Rank enumu içerisine tanımlandı.

Ardından AbstractSoldier adında soyut bir asker sınıfı oluşturuldu. Askerin adı ve rütbesi başlatıcı metod tarafından isteniyor. Aynı zamanda AddSoldier, RemoveSoldier ve ExecuteOrder metodlarının ezilmesi isteniyor.

```
public class CompositeSoldier : AbstractSoldier
{
    private List<AbstractSoldier> _soldiers = new List<AbstractSoldier>();
    public CompositeSoldier(string name, Rank rank) : base(name, rank)
    {
    }

    public override void AddSoldier(AbstractSoldier soldier)
    {
        _soldiers.Add(soldier);
    }

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

    string soldierList="";
    public override string ExecuteOrder()
    {
        soldierList += String.Format("{0} {1} ", _rank, _name);
        foreach (AbstractSoldier soldier in _soldiers)
        {
            soldierList += "<br />" + soldier.ExecuteOrder();
        }
        return soldierList;
    }
}
```

AbstractSoldier soyut sınıfından türeyen CompositeSoldier sınıfında ise asker listesi, askerin adı ve rütbesi (temel sınıf baz alınan), ezilmek zorunda kalınan AddSoldier, RemoveSoldier, ExecuteOrder metodları bulunmaktadır.

```
public class PrimitiveSoldier : AbstractSoldier
{
    public PrimitiveSoldier(string name, Rank rank) : base(name, rank)
    {
    }

    public override void AddSoldier(AbstractSoldier soldier)
    {
        throw new NotImplementedException();
    }

    public override void RemoveSoldier(AbstractSoldier soldier)
    {
        throw new NotImplementedException();
    }

    public override string ExecuteOrder()
    {
        return String.Format("{0} {1}", _rank, _name);
    }
}
```

Hiyerarşinin sonunda yer alan PrimitiveSoldier sınıfı ise AbstractSoldier sınıfından türemiş olsa da asker listesi tutma kabiliyetine sahip değildir. Bu yüzden PrimitiveSoldier sınıfına bağlı askerler bulunmamaktadır. Bu sebeple de AddSoldier, RemoveSoldier metotları ezilse de bir anlamı olmayacaktır, hata fırlatılacaktır. Ancak başlatıcı metodda asker adı ve rütbesi alındığından ExecuteOrder metodu ile bu bilgi çağrıldığı yere gönderilecektir.

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        //Root Oluşturuldu
        CompositeSoldier generalAhmet = new CompositeSoldier("Ahmet", Rank.General);

        //root altına leaf tipler oluşturuldu
        generalAhmet.AddSoldier(new PrimitiveSoldier("Serkan", Rank.Colonel));
        generalAhmet.AddSoldier(new PrimitiveSoldier("Mahmut", Rank.Colonel));

        //Composite tipler oluşturuldu
        CompositeSoldier colonelAziz = new CompositeSoldier("Aziz", Rank.Colonel);
        CompositeSoldier lieutenantColonelZing = new CompositeSoldier("Zing",
            Rank.LieutenantColonel);

        //Composite tiplerine bağlı primitiveler
        lieutenantColonelZing.AddSoldier(new PrimitiveSoldier("Tomasson", Rank.Captain));
        colonelAziz.AddSoldier(lieutenantColonelZing);
        colonelAziz.AddSoldier(new PrimitiveSoldier("Mayro", Rank.LieutenantColonel));

        //Root'un altına Composite nesne örneği eklendi
        generalAhmet.AddSoldier(colonelAziz);
        generalAhmet.AddSoldier(new PrimitiveSoldier("Zulu", Rank.Colonel));

        ViewBag.sonuc = generalAhmet.ExecuteOrder();

        return View();
    }
}
```

Index action içine ilk olarak hiyerarşinin en başı, root oluşturuldu. generalAhmet adındaki nesneye Serkan ve Mahmut adlarında PrimitiveSoldier eklendi. Ardından colonelAziz ve lieutenantColonelZing adlarında CompositeSoldier nesnesi oluşturuldu ve Zing'e bağlı Tomasson, adında PrimitiveSoldier oluşturularak colonelAziz'e Zing bağlandı. colonelAziz için Mayro adında PrimitiveSoldier eklendi, colonelAziz root olan generalAhmet'e bağlandı ve aynı zamanda Zulu PrimitiveSoldier eklendi. ExecuteOrder metoduyla root ViewBag.sonuc'a taşındı.

```
@{  
    ViewData["Title"] = "Home Page";  
}  
<h3>@Html.Raw(ViewBag.sonuc)</h3>
```

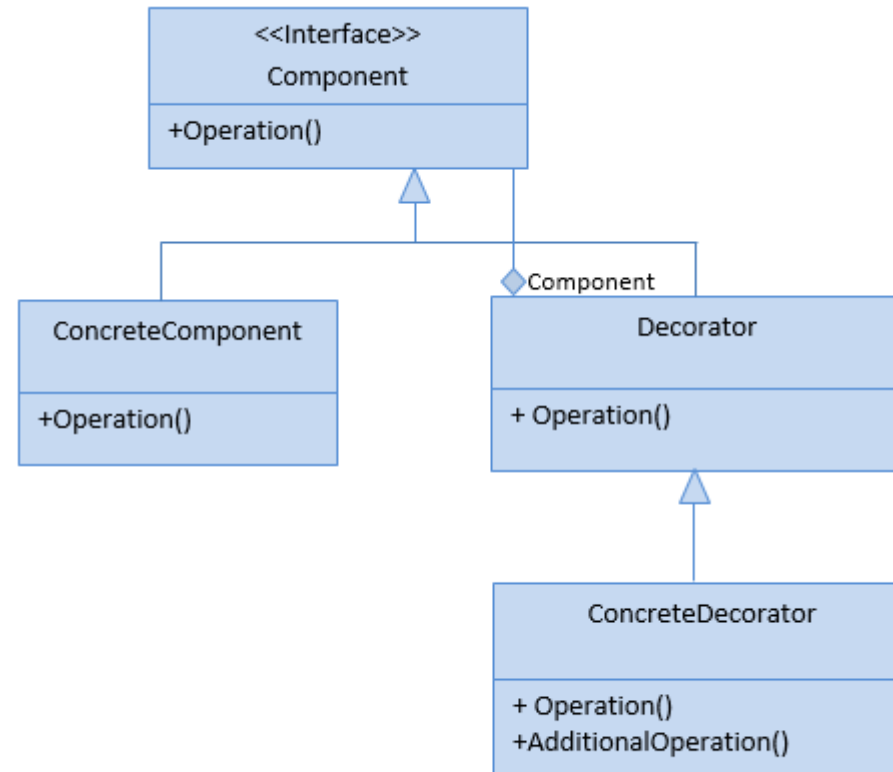
Halic\_Seminer8   Home

General Ahmet  
Colonel Serkan  
Colonel Mahmut  
Colonel Aziz  
LieutenantColonel Zing  
Captain Tomasson  
LieutenantColonel Mayro  
Colonel Zulu

Index action 'da ViewBag.sonuc'a taşınan veri Index.cshtml view dosyasında html kodlarının çalışması için @Html.Raw() metodu içerisine alındı.



Nesnelere dinamik olarak sorumluluklar ekler. Bu desenin amacı mevcut bir sınıfı sarmak, ona başka işlemler eklemek ve ardından aynı ara yüzü dış dünyaya sunmaktır. Bu yüzden dekoratör, onu kullanan insanlara tam olarak orijinal sınıf gibi görünür.



Decorator Pattern

```
public interface ICar
{
    string GetDescription();
    double GetCost();
}
```

```
public class EconomyCar : ICar
{
    public double GetCost() => 450000.0;
    public string GetDescription() => "Economy Car";
}
```

```
public class DeluxCar : ICar
{
    public double GetCost() => 750000.0;
    public string GetDescription() => "Delux Car";
}
```

```
public class LuxuryCar : ICar
{
    public double GetCost() => 1000000.0;
    public string GetDescription() => "Luxury Car";
}
```

Bu örnekte, ICar arayüzü oluşturularak, GetDescription ve GetCost metodlarını tanımlayarak oluşturulan EconomyCar, DeluxCar, LuxuryCar sınıfları tarafından kullanılması zorunlu kılındı.

```
public abstract class CarAccessoriesDecorator : ICar
{
    private ICar _car;
    public CarAccessoriesDecorator(ICar car)
    {
        _car = car;
    }
    public virtual double GetCost() => _car.GetCost();
    public virtual string GetDescription() => _car.GetDescription();
}
```

```
public class BasicAccessories:CarAccessoriesDecorator
{
    public BasicAccessories(ICar car):base(car) { }
    public override string GetDescription() => base.GetDescription() + ",Basic
Accessories Package";
    public override double GetCost() => base.GetCost() + 2000.0;
}
```

```
public class ExtremeSportAccessories : CarAccessoriesDecorator
{
    public ExtremeSportAccessories(ICar car) : base(car){ }
    public override double GetCost() => base.GetCost() + 25000;
    public override string GetDescription()=> base.GetDescription() + ",Extreme Sport
Accessories";
}
```

CarAccessoriesDecorator isimli soyut sınıf oluşturularak bu sınıf da ICar arayüzünden gelen metodları virtual etiketiyle kullandı. Böylece bu soyut sınıftan türeyen BasicAccessories, ExtremeSportAccessories, AdvancedAccessories, SportsAccessories ilgili metodları override ederek sahip oldukları ek aksesuarlar için fiyat ve aksesuar bilgisi döndürme yeteneğine haiz oldular.



# Decorator

```
public class AdvancedAccessories : CarAccessoriesDecorator
{
    public AdvancedAccessories(ICar car) : base(car) { }
    public override double GetCost() => base.GetCost() + 10000.0;
    public override string GetDescription()=>base.GetDescription() + ",Advanced
Accessories Package";
}
```

```
public class SportsAccessories : CarAccessoriesDecorator
{
    public SportsAccessories(ICar car) : base(car) { }
    public override double GetCost()=> base.GetCost() + 15000.0;
    public override string GetDescription()=>base.GetDescription() + ",Sports
Accessories Package";
}
```

```
public IActionResult Index()
{
    //EconomyCar örneği oluşturun.
    ICar objCar = new EconomyCar();

    //BasicAccessories içeren EconomyCar örneği.
    CarAccessoriesDecorator objAccessoriesDecorator = new BasicAccessories(objCar);

    //EconomyCar örneğini AdvancedAccessories örneğiyle sarın.
    objAccessoriesDecorator = new AdvancedAccessories(objAccessoriesDecorator);

    ViewBag.carDetails = objAccessoriesDecorator.GetDescription();
    ViewBag.carCost = objAccessoriesDecorator.GetCost().ToString();
    return View();
}
```

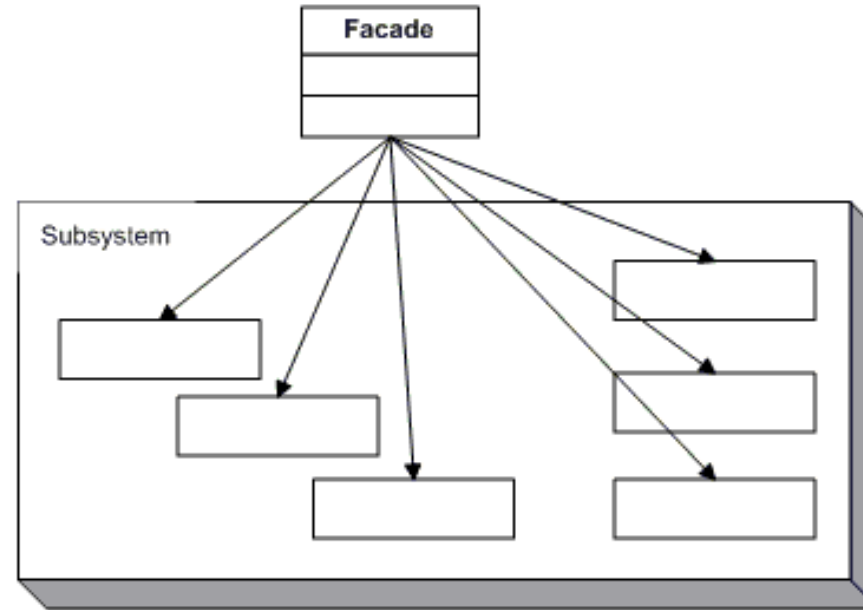
```
@{
    ViewData["Title"] = "Home Page";
}

<h3>Car Details: @ViewBag.carDetails</h3>
<h3> Total Price: @ViewBag.carCost</h3>
```

Index action'ında Economy sınıfı araç örneği oluşturularak bu araç için basic aksesuar ve advanced aksesuar paketleri tercih edildi ve toplam maliyet döndürüldü.

Örnekte; EconomyCar için fiyat 450.000, BasicAccessories için fiyat 2.000, AdvancedAccessories için fiyat 10.000 gelen fiyatlar toplandığında sonuç ekranda görüntülenen 462.000 olacaktır.

Karmaşık alt sistemleri olan bir yapıyı, sade bir ara yüz sağlayan Facade sınıfı ile basitleştirebiliriz. Bir alt sistemin parçalarını oluşturan sınıfları, istemciden soyutlayarak kullanımı daha da kolaylaştırmaktadır. Bu design pattern oldukça sık kullanılmaktadır.



```
public interface IPizza
{
    string GetVegPizza();
    string GetNonVegPizza();
}
```

```
public class PizzaProvider : IPizza
{
    public string GetNonVegPizza() => "Getting Non Veg Pizza" + GetNonVegToppings();
    public string GetVegPizza() => "Getting Veg Pizza";
    public string GetNonVegToppings() => "Getting Non Veg Pizza Toppings.";
}
```

```
public interface IBread
{
    string GetGarlicBread();
    string GetCheesyGarlicBread();
}
```

```
public class BreadProvider : IBread
{
    public string GetCheesyGarlicBread() => "Getting Cheesy Garlic Bread." +
    GetCheese();
    public string GetGarlicBread() => "Getting Garlic Bread.";
    public string GetCheese() => "Getting Cheese.";
}
```

Vegan pizza ile vegan olmayan pizza metod belirtecinin yer aldığı IPizza arayüzü bu arayüzü kullanan PizzaProvider isimli sınıf şart koşulan metodlara ek olarak vegan olmayan soslar metodu da oluşturulmuştur.

Sarımsaklı ekmek ve Pernirli sarımsaklı ekmek metod belirteçlerinin oluşturulduğu IBread arayüzü de BreadProvider tarafından imlement edilmiş GetChesse adıyla ek metod da oluşturulmuştur.

```
public class RestaurantFacade
{
    private IPizza _PizzaProvider;
    private IBread _BreadProvider;
    public RestaurantFacade()
    {
        _PizzaProvider = new PizzaProvider();
        _BreadProvider = new BreadProvider();
    }
    public string GetNonVegPizza() => _PizzaProvider.GetNonVegPizza();
    public string GetVegPizza() => _PizzaProvider.GetVegPizza();
    public string GetGarlicBread() => _BreadProvider.GetGarlicBread();
    public string GetCheesyGarlicBread() => _BreadProvider.GetCheesyGarlicBread();
}
```

```
public IActionResult Index()
{
    RestaurantFacade facadeForClient = new RestaurantFacade();
    ViewBag.durum1 = facadeForClient.GetNonVegPizza();
    ViewBag.durum2 = facadeForClient.GetVegPizza();
    ViewBag.durum3 = facadeForClient.GetGarlicBread();
    ViewBag.durum4 = facadeForClient.GetCheesyGarlicBread();
    return View();
}
```

RestaurantFacade sınıfı IPizza ve IBread arayüzlerini başlatıcı metodunda örneklendirmiş ve erişilen metodların döndürdüğü değer burada tanımlanan metodlara taşınmıştır.

Index action'da da Facade sınıfımız örneklenerek ilgili metodlar ViewBag'lere atanarak Index view'ımıza gönderilmiştir.





```
<h1>@ViewBag.durum1</h1>  
<h2>@ViewBag.durum2</h2>  
<h3>@ViewBag.durum3</h3>  
<h4>@ViewBag.durum4</h4>
```

Halic\_Seminer10\_FacadePattern [Home](#) [About](#) [Contact](#)

Getting Non Veg PizzaGetting Non Veg Pizza Toppings.

Getting Veg Pizza

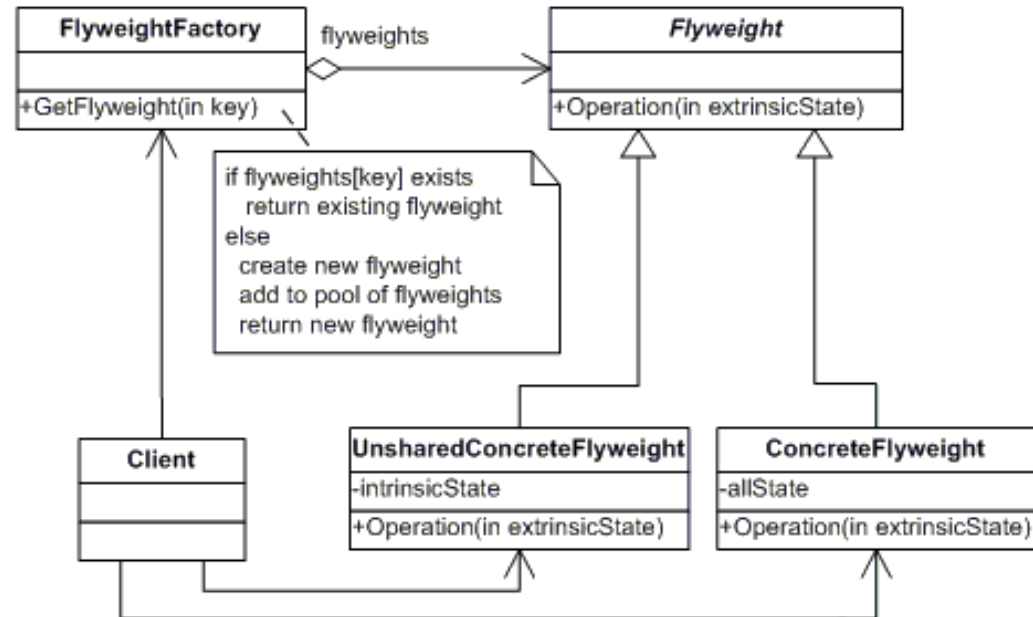
Getting Garlic Bread.

Getting Cheesy Garlic Bread.Getting Cheese.

© 2021 - Halic\_Seminer10\_FacadePattern

Durumlara denk gelen metodlar tersine incelendiğinde, bir Facade sınıfından farklı iki arayüzü kullanan sınıfların ilgili metodlarına erişebildiğimizi görmekteyiz.

Bu desen; benzer nesneleri tekrar tekrar oluşturmak yerine ortak bellek alanlarını kullanarak bellek kullanımını azaltmayı amaçlamaktadır. Ortak bellek alanları nesne havuzu olarak adlandırılabilir. Desen nesneleri bir kez oluşturur ve onları bir havuza kaydeder.



```
public interface IShape
{
    string Print();
}
```

```
public class Rectangle : IShape
{
    public string Print() => "Printing Rectangle";
}
```

```
public class Circle : IShape
{
    public string Print() => "Printing Circle";
}
```

Bir şekil arayüzü ve bu arayüzü implement eden Kare ve Daire sınıfları oluşturarak çağrıldığında metin döndürmek için tanımlanan Print() adlı metodun içi dolduruldu.

```
public class ShapeObjectFactory
{
    Dictionary<string, IShape> shapes = new Dictionary<string, IShape>();

    public int TotalObjectsCreated
    {
        get { return shapes.Count; }
    }

    public IShape GetShape(string ShapeName)
    {
        IShape shape = null;
        if (shapes.ContainsKey(ShapeName))
        {
            shape = shapes[ShapeName];
        }
        else
        {
            switch (ShapeName)
            {
                case "Rectangle":
                    shape = new Rectangle();
                    shapes.Add("Rectangle", shape);
                    break;
                case "Circle":
                    shape = new Circle();
                    shapes.Add("Circle", shape);
                    break;
                default:
                    throw new Exception("Factory cannot create the object specified");
            }
        }
        return shape;
    }
}
```

Dictionary yapısı C#'ta key, value yapısı ile çalışır. Key tekrar edilemez; ancak value değerleri aynı olabilir.

ShapeObjectFactory sınıfında da şekiller adında dictionary oluşturulmuştur.

TotalObjectCreated çağrıldığında sözlükte bulunan şekil sayısı dönmektedir.

GetShape metodunda ise çağrılan şekil mevcutsa döndürülmekte mevcut değilse de hata fırlatılmaktadır.

```
public IActionResult Index()
{
    ShapeObjectFactory sof = new ShapeObjectFactory();

    IShape shape = sof.GetShape("Rectangle");
    ViewBag.shape1 = shape.Print();

    shape = sof.GetShape("Rectangle");
    ViewBag.shape2 = shape.Print();

    shape = sof.GetShape("Circle");
    ViewBag.shape3 = shape.Print();

    shape = sof.GetShape("Rectangle");
    ViewBag.shape4 = shape.Print();

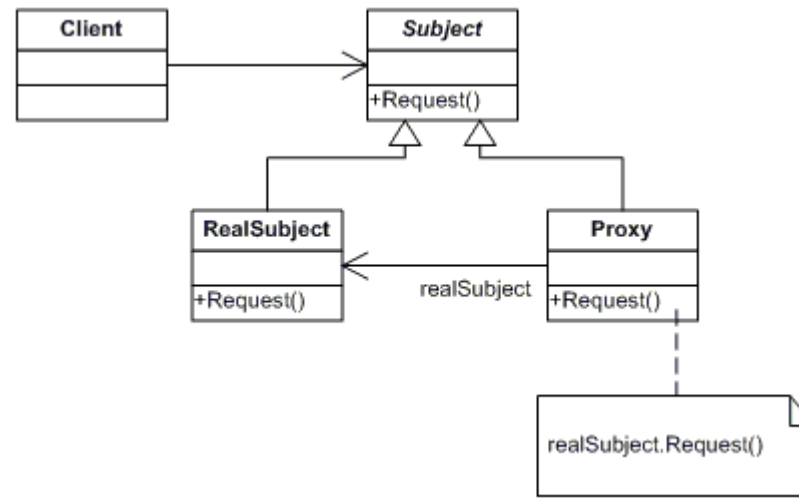
    shape = sof.GetShape("Circle");
    ViewBag.shape5 = shape.Print();

    return View();
}
```

```
<h1>@ViewBag.shape1</h1>
<h2>@ViewBag.shape2</h2>
<h3>@ViewBag.shape3</h3>
<h4>@ViewBag.shape4</h4>
<h5>@ViewBag.shape5</h5>
```

Index action'ında ise ShapeObjectFactory sınıfı örneklendirilmiş GetShape sınıfına sırasıyla Rectangle, Restangle, Circle, Rectangle, Circle yazılmış ilgili şekiller mevcut olduğundan web sayfamızda da ilgili metinler görüntülenmiştir.

Adından da anlaşılacağı üzere bir nesnenin temsili olarak hareket eder ve temelde orijinal nesneyi kullanmak için erişim noktası oluşturmaktadır. Nesneyle istemci arasına yeni bir katman koyarak nesnenin kontrollü bir şekilde paylaşılması sağlanır. Oluşturulması basit ve sık kullanılan desenlerdendir.



```
public interface IMath
{
    double Add(double x, double y);
    double Sub(double x, double y);
    double Mul(double x, double y);
    double Div(double x, double y);
}
```

```
public class Math : IMath
{
    public double Add(double x, double y) => x + y;

    public double Div(double x, double y) => x / y;

    public double Mul(double x, double y) => x * y;

    public double Sub(double x, double y) => x - y;
}
```

```
public class MathProxy : IMath
{
    private Math _math = new Math();
    public double Add(double x, double y) => _math.Add(x, y);
    public double Div(double x, double y) => _math.Div(x, y);
    public double Mul(double x, double y) => _math.Mul(x, y);
    public double Sub(double x, double y) => _math.Sub(x, y);
}
```

O örnekte 4 temel matematiksel işlemin yapılması için IMath adında arayüz oluşturuldu ve bu arayüzü Math sınıfı implement ederek ilgili metodları doldurdu.

MathProxy sınıfı ise IMath arayüzünü implement ederek metodlarını Math sınıfı üzerinden gerçekleştirdi. Bir nevi köprü oluşturdu.

Daha da basite indirgemek istersek OOP'de kapsülleme konusunu aklımıza getirelim. Direkt math sınıfına erişmeden Proxy üzerinden işlemlerimizi gerçekleştirmiş olacağız.

```
public IActionResult Index()
{
    MathProxy proxy = new MathProxy();

    ViewBag.Toplama = proxy.Add(3,5).ToString();
    ViewBag.Cikarma = proxy.Sub(3,5).ToString();
    ViewBag.Carpma = proxy.Mul(3,5).ToString();
    ViewBag.Bolme = proxy.Div(3,5).ToString();

    return View();
}
```

```
<h3>İşlem Sonuçları</h3>
<p>@ViewBag.Cikarma</p>
<p>@ViewBag.Carpma</p>
<p>@ViewBag.Bolme</p>
<p>@ViewBag.Toplama</p>
```

Index action'ımız MathProxy sınıfını örneklemekte ve bu sınıf üzerinden işlemler yaparak ViewBag'lere elde ettiği sonucu atamıştır.

Halic\_Seminer12\_Proxy [Home](#) [About](#) [Contact](#)

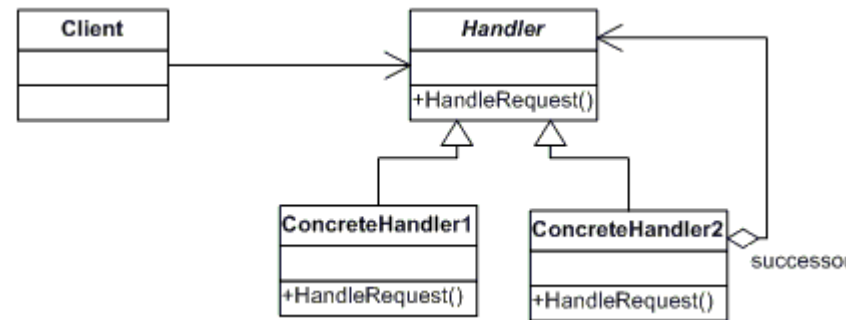
### İşlem Sonuçları

-2  
15  
0,6  
8

© 2021 - Halic\_Seminer12\_Proxy



Bir işlemi yapabilecek birden fazla sınıftan hangisinin yapacağına karar veren tasarım desenidir. If/else kod bloklarının çoğalarak kod karmaşası oluşmasının önüne geçer.





# Chain of Responsibility

```
public class Purchase
{
    private int _number;
    private double _amount;
    private string _purpose;

    public Purchase(int number, double amount, string purpose)
    {
        this._number = number;
        this._amount = amount;
        this._purpose = purpose;
    }

    public int Number
    {
        get { return _number; }
        set { _number = value; }
    }

    public double Amount
    {
        get { return _amount; }
        set { _amount = value; }
    }

    public string Purpose
    {
        get { return _purpose; }
        set { _purpose = value; }
    }
}
```

Purchase sınıfı başlatıcı metodlarında private olan değişkenleri için değer atanmasını istemektedir. Propertyler ile daha sonrasında erişim sağlanarak kapsülleme yapılmıştır.



# Chain of Responsibility

```
public abstract class Approver
{
    protected Approver successor;
    public void SetSuccessor(Approver successor)
    {
        this.successor = successor;
    }
    public abstract string ProcessRequest(Purchase purchase);
}
```

```
public class Director : Approver
{
    public override string ProcessRequest(Purchase purchase)
    {
        if (purchase.Amount < 10000.0)
        {
            return String.Format("{0} approved request# {1}",
                this.GetType().Name, purchase.Number);
        }
        else if (successor != null)
        {
            successor.ProcessRequest(purchase);
        }
        return "";
    }
}
```

Approver soyut sınıfında Approver tipinde successor değişkeni bu değişkene veri atayan SetSuccessor metodu ek olarak bu sınıfı miras alanın zorunlu olarak ezeceği Purchase nesnesi alan ProcessRequest metod şablonu oluşturulmuştur.

Approver sınıfından türeyen Director sınıfı ProcessRequest metodunu ezerek purchase nesnesinin taşıdığı Amount değerine göre karar yapısı ile sonuç döndürmektedir.



# Chain of Responsibility

```
public class President : Approver
{
    public override string ProcessRequest(Purchase purchase)
    {
        if (purchase.Amount < 100000.0)
        {
            return String.Format("{0} approved request# {1}",
                this.GetType().Name, purchase.Number);
        }
        else if (successor != null)
        {
            successor.ProcessRequest(purchase);
        }
        return "";
    }
}
```

```
public class VicePresident : Approver
{
    public override string ProcessRequest(Purchase purchase)
    {
        if (purchase.Amount < 25000.0)
        {
            return String.Format("{0} approved request# {1}",
                this.GetType().Name, purchase.Number);
        }
        else if (successor != null)
        {
            successor.ProcessRequest(purchase);
        }
        return "";
    }
}
```

Approver sınıfından türeyen Director sınıfında olduğu gibi President ve VicePresident sınıflarında da ProcessRequest metodu ezilerek purchase nesnesinin taşıdığı Amount değerine göre karar yapısı ile sonuç döndürmektedir.



# Chain of Responsibility

```
public IActionResult Index()
{
    Approver tutku = new Director();
    Approver ahmet = new VicePresident();
    Approver gokce = new President();

    tutku.SetSuccessor(ahmet);
    ahmet.SetSuccessor(gokce);

    Purchase p = new Purchase(2034, 350.00, "Assets");
    ViewBag.tutku = tutku.ProcessRequest(p);

    p = new Purchase(2035, 32590.10, "Project X");
    ViewBag.tutku2 = tutku.ProcessRequest(p);

    p = new Purchase(2036, 122100.00, "Project X");
    ViewBag.tutku3 = tutku.ProcessRequest(p);

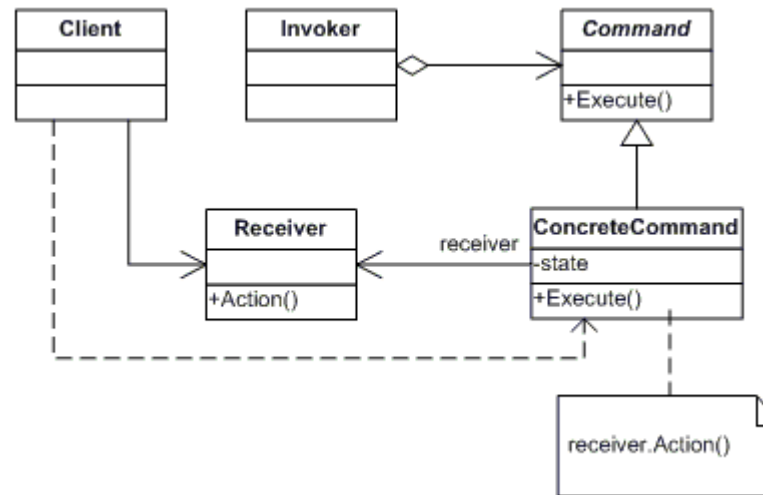
    return View();
}
```

```
<p>@ViewBag.tutku</p>
<p>@ViewBag.tutku2</p>
<p>@ViewBag.tutku3</p>
```

Index aksiyonumuzda ise soyut sınıfımıza bağlı alt sınıflar ile nesneler oluşturulmuş bu nesneler üzerinden Purchase talebinde bulunulmuştur. İlgili sınıfların içinde gerçekleşen karar yapısında şartları sağlayan sınıflar string olarak mesaj döndürmüşlerdir.

Bu örnekte Director sınıfı için gereken Amount değeri sağlandığından tek o nesnenin mesajı ekrana yazdırılmıştır.

Komut tasarım kalıbı, kullanıcı isteklerini gerçekleştiren kod yapısının sarmalanarak nesneler halinde saklanmasına dayanır. İşlemlerin nesne haline getirilip başka bir nesne(invoker) üzerinden tetiklendiği bir tasarım desenidir.



```
public abstract class Command
{
    public abstract string Execute();
    public abstract string UnExecute();
}
```

```
public class CalculatorCommand : Command
{
    private char _operator;
    private int _operand;
    private Calculator _calculator;

    public CalculatorCommand(Calculator calculator,
        char @operator, int operand)
    {
        this._calculator = calculator;
        this._operator = @operator;
        this._operand = operand;
    }

    public char Operator
    {
        set { _operator = value; }
    }

    public int Operand
    {
        set { _operand = value; }
    }

    public override string Execute() => _calculator.Operation(_operator, _operand);

    public override string UnExecute() => _calculator.Operation(Undo(_operator), _operand);

    private char Undo(char @operator)
    {
        switch (@operator)
        {
            case '+': return '-';
            case '-': return '+';
            case '*': return '/';
            case '/': return '*';
            default:
                throw new ArgumentException("@operator");
        }
    }
}
```

Command soyut sınıfı oluşturularak Execute ve UnExecute şablon metodları eklendi.

CalculatorCommand sınıfı Command'dan türetilerek char operator, int operand, Calculator tipinde calculator değişkenler oluşturularak değerler başlatıcı metoddan alındı. Property ile operator ve operand sadece set edilebilir olarak tanımlandı (kapsülleme). Calculator tipinde calculator nesnesine ise sınıf harici erişilemez (private, kapsülleme yok).

Execute metodu \_operand ve \_operatör için operasyon izni verirken UnExecute ise Undo metodunu çağırarak işlemi geri almaktadır.



# Command

```
public class Calculator
{
    private int _curr = 0;

    public string Operation(char @operator, int operand)
    {
        switch (@operator)
        {
            case '+': _curr += operand; break;
            case '-': _curr -= operand; break;
            case '*': _curr *= operand; break;
            case '/': _curr /= operand; break;
        }
        return String.Format("Current value = {0,3} (following {1} {2})",
            _curr, @operator, operand);
    }
}
```

Calculator sınıfı bir önceki sınıfta değinildiği üzere operatör ve operand değişkenlerini alarak işlem gerçekleştirmektedir.



```
public class User
{
    private Calculator _calculator = new Calculator();
    private List<Command> _commands = new List<Command>();
    private int _current = 0;
    public string Redo(int levels)
    {
        string x = "";
        for (int i = 0; i < levels; i++)
        {
            if (_current < _commands.Count - 1)
            {
                Command command = _commands[_current++];
                x+= command.Execute()+"<br />";
            }
        }
        return x;
    }

    public string Undo(int levels)
    {
        string x = "";
        for (int i = 0; i < levels; i++)
        {
            if (_current > 0)
            {
                Command command = _commands[--_current] as Command;
                x+= command.UnExecute() + "<br />";
            }
        }
        return x;
    }

    public string Compute(char @operator, int operand)
    {
        Command command = new CalculatorCommand(
            _calculator, @operator, operand);
        string x = command.Execute();
        _commands.Add(command);
        _current++;

        return x;
    }
}
```

User sınıfında ise calculator nesnesi, commands listesi ve \_current değişkenleri, for döngüsü içinde istenen şart sağlanıyorsa çalışan Execute metodunun yer aldığı Redo metodu, aksi işlemi gerçekleştiren Undo metodu ve operator, operand alan işlemleri Execute metoduna gönderen Compute metodu bulunmaktadır.

```
public IActionResult Index()
{
    User user = new User();

    ViewBag.durum1 = user.Compute('+', 100);
    ViewBag.durum2 = user.Compute('-', 50);
    ViewBag.durum3 = user.Compute('*', 10);
    ViewBag.durum4 = user.Compute('/', 2);

    ViewBag.durum5 = user.Undo(4);

    ViewBag.durum6 = user.Redo(3);

    return View();
}
```

```
<br />
<p>@ViewBag.durum1</p>
<p>@ViewBag.durum2</p>
<p>@ViewBag.durum3</p>
<p>@ViewBag.durum4</p>
<hr />
<p>@Html.Raw(ViewBag.durum5)</p>
<hr />
<p>@Html.Raw(ViewBag.durum6)</p>
```

Index aksiyonunda user nesnesi oluşturularak nesne üzerinden Compute metoduna parametre ile operator ve operand gönderilmektedir.

Undo metodu ile 4 işlem geri yani aksi işlemi,

Redo metodu ile listedeki 3 işlem olağan sırasında gerçekleştirilmektedir.

Halic\_Seminer13\_Command Home About Contact

Current value = 100 (following + 100)

Current value = 50 (following - 50)

Current value = 500 (following \* 10)

Current value = 250 (following / 2)

Sırayla yapılan işlemler...

Current value = 500 (following \* 2)

Current value = 50 (following / 10)

Current value = 100 (following + 50)

Current value = 0 (following - 100)

Tersine yapılan 4 işlem... (Undo(4))

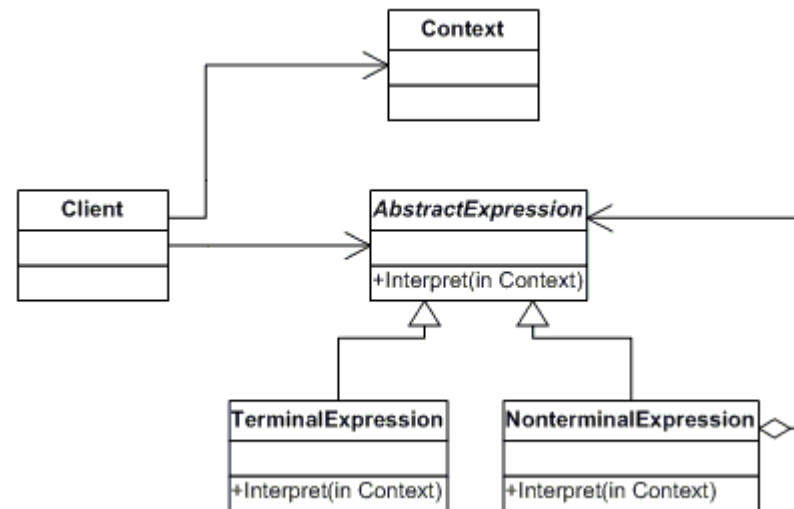
Current value = 100 (following + 100)

Current value = 50 (following - 50)

Current value = 500 (following \* 10)

Sırayla yapılan işlemin ilk 3'ü... (Redo(3))

Belirli bir düzen veya kurala bağlı olan metinlerin sayısal veya mantıksal olarak işlenmesi gereken durumlarda kullanılır. Kısacası roma rakamı gibi bir sayının MCMXXVIII nasıl çevirileceğini kurallar bütünü içerisinde düzenleyen yapıdır.



```
public class Context
{
    private string _input;
    private int _output;

    public Context(string input)
    {
        this._input = input;
    }

    public string Input
    {
        get { return _input; }
        set { _input = value; }
    }

    public int Output
    {
        get { return _output; }
        set { _output = value; }
    }
}
```

Roma rakamlarını (string) sayıya (int) dönüştürecek olan örnekte; Context sınıfı içerisinde girecek olan string tipi ve çıkış yapacak int tipi için değişken oluşturularak property ile kapsülleme işlemi gerçekleştirildi ve input değeri başlatıcı metod ile istendi.

```
public abstract class Expression
{
    public void Interpret(Context context)
    {
        if (context.Input.Length == 0)
            return;

        if (context.Input.StartsWith(Nine()))
        {
            context.Output += (9 * Multiplier());
            context.Input = context.Input.Substring(2);
        }
        else if (context.Input.StartsWith(Four()))
        {
            context.Output += (4 * Multiplier());
            context.Input = context.Input.Substring(2);
        }
        else if (context.Input.StartsWith(Five()))
        {
            context.Output += (5 * Multiplier());
            context.Input = context.Input.Substring(1);
        }

        while (context.Input.StartsWith(One()))
        {
            context.Output += (1 * Multiplier());
            context.Input = context.Input.Substring(1);
        }
    }

    public abstract string One();
    public abstract string Four();
    public abstract string Five();
    public abstract string Nine();
    public abstract int Multiplier();
}
```

Expression adında abstract sınıf oluşturularak, yine bu sınıf içerisinde string dönüş tipli One, string dönüş tipli Five, string dönüş tipli Nine ve int dönüş tipli Multiplier metod şablonları oluşturuldu. Expression sınıfı içerisinde soyut olmayan context parametresi alan Interpret isimli metod yazıldı.

Interpret metodu context parametresinden gelen göre işlemler gerçekleştirerek Context sınıfı içinde bulunan output parametresine elde ettiği sonucu döndürmektedir.

```
public class ThousandExpression : Expression
{
    public override string One() => "M";
    public override string Nine() => " ";
    public override string Four() => " ";
    public override string Five() => " ";
    public override int Multiplier() => 1000;
}
```

```
public class HundredExpression : Expression
{
    public override string One() => "C";
    public override string Four() => "CD";
    public override string Five() => "D";
    public override string Nine() => "CM";
    public override int Multiplier() => 100;
}
```

```
public class TenExpression : Expression
{
    public override string One() => "X";
    public override string Four() => "XL";
    public override string Five() => "L";
    public override string Nine() => "XC";
    public override int Multiplier() => 10;
}
```

```
public class OneExpression : Expression
{
    public override string One() => "I";
    public override string Four() => "IV";
    public override string Five() => "V";
    public override string Nine() => "IX";
    public override int Multiplier() => 1;
}
```

ThousandExpression, HundredExpression, TenExpression sınıfları Expression soyut sınıfından miras aldığından One, Nine, Four, Five, Multiplier metodlarını ezmek zorundadırlar.

ThousandExpression; bin basamaklı işlem yapılması, HundredExpression; yüz basamaklı işlem yapılması, TenExpression; on basamaklı, OneExpression bir basamaklı sayılarla çeviri yapılması için oluşturulan sınıflardır.

```
public IActionResult Index()
{
    string roman = "MCMXXVIII";
    Context context = new Context(roman);

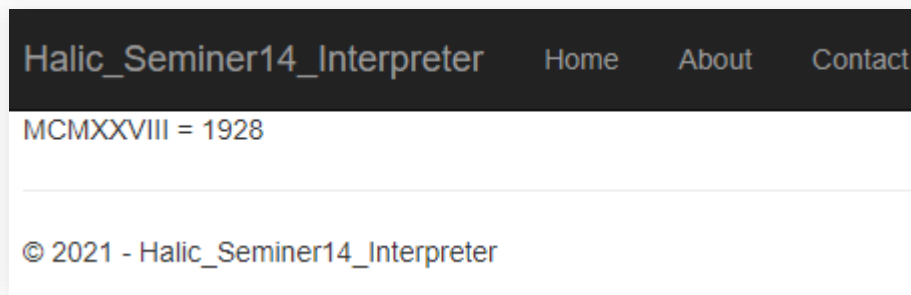
    List<Expression> tree = new List<Expression>();
    tree.Add(new ThousandExpression());
    tree.Add(new HundredExpression());
    tree.Add(new TenExpression());
    tree.Add(new OneExpression());

    foreach (Expression exp in tree)
    {
        exp.Interpret(context);
    }

    ViewBag.sonuc = String.Format("{0} = {1}", roman, context.Output);

    return View();
}
```

<p>@ViewBag.sonuc</p>

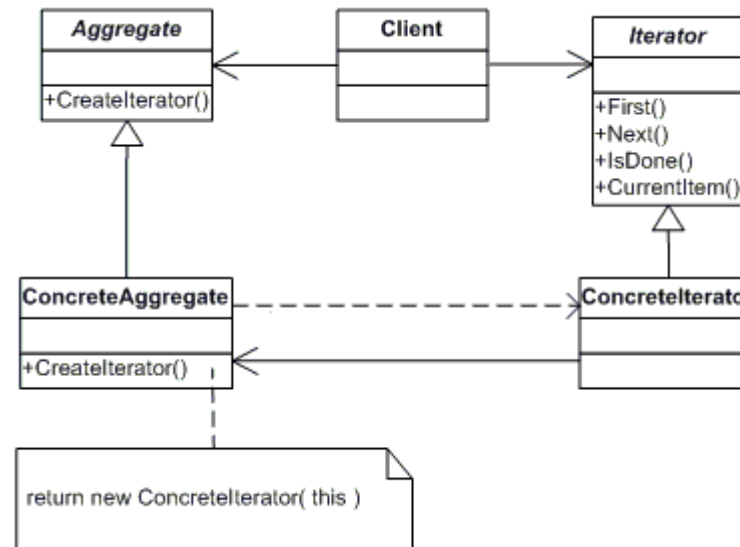


Index aksiyonunda ilk olarak çıktısını görmek istediğimiz roman sayısını değişkene atayarak Context sınıfında başlatıcı metod tarafından istenen input değişkenine gönderdik.

İşlemlerin gerçekleştiği expression listesi için ThousandExpression, HundredExpression, TenExpression, OneExpression sınıflarının nesne örneğini oluşturduk. Foreach ile bu liste içerisinde dönerek Expression içinde yer alan Interpret metoduna işlem yaptırdık ve context.Output'a atanan sonucu View'a ViewBag ile göndererek sonucu ekrana yazdırdık.



Bir veri kümesinin bağlantılarını en aza indirmek için; listede yer alan nesnelerin, sırasıyla uygulamadan soyutlamasını sağlar.



```
interface IAbstractIterator
{
    Item First();
    Item Next();
    bool IsDone { get; }
    Item CurrentItem { get; }
}
```

```
interface IAbstractCollection
{
    Iterator CreateIterator();
}
```

```
public class Item
{
    private string _name;
    public Item(string name)
    {
        this._name = name;
    }
    public string Name
    {
        get { return _name; }
    }
}
```

IAbstractIterator isimli arayüz içerisine Item tipinde First, Item tipinde Next, bool tipinde IsDone, Item tipinde CurrentItem metod imzalarını;

IAbstractCollection isimli arayüz içerisine Iterator tipinde CreateIterator metod imzasını;

Başlatıcı metod dışında herhangi bir şekilde değer atanamayan name değişkeninin bulunduğu Item isimli sınıf oluşturuldu.

```
public class Iterator : IAbstractIterator
{
    private Collection _collection;
    private int _current = 0;
    private int _step = 1;

    public Iterator(Collection collection)
    {
        this._collection = collection;
    }

    public Item First() => _collection[0] as Item;

    public Item Next()
    {
        _current += _step;
        if (!IsDone)
            return _collection[_current] as Item;
        else
            return null;
    }

    public int Step
    {
        get { return _step; }
        set { _step = value; }
    }

    public Item CurrentItem
    {
        get { return _collection[_current] as Item; }
    }

    public bool IsDone
    {
        get { return _current >= _collection.Count; }
    }
}
```

IAbstractIterator arayüzünü implement eden Iterator sınıfında Collection tipinde collection, int tiplerinde \_current ve step değişkenleri oluşturularak başlatıcı metod Collection tipinde collection istemektedir. Arayüzde yer alan metod işaretlerinin hepsi bu sınıf içerisinde kodlanmıştır. Bu doğrultuda koleksiyonumuzdaki ilk eleman First metodunda, sırasına göre gelecek olan eleman Next metodunda, atlanacak aşama step propertysinde, seçili eleman CurrentItem metodundan döndürülmektedir.

```
public class Collection: IAbstractCollection
{
    private ArrayList _items = new ArrayList();

    public Iterator CreateIterator() => new Iterator(this);

    public int Count
    {
        get { return _items.Count; }
    }

    public object this[int index]
    {
        get { return _items[index]; }
        set { _items.Add(value); }
    }
}
```

IAbstractCollection arayüzünü implement eden Collection sınıfı ise CreateIterator metodundan yeni bir Iterator döndürmektedir. Ayrıca içerisinde bulunan dizi listesi üzerinden Count propertysi ile dizi boyutunu döndürmektedir.

```
public IActionResult Index()
{
    Collection collection = new Collection();
    collection[0] = new Item("Item 0");
    collection[1] = new Item("Item 1");
    collection[2] = new Item("Item 2");
    collection[3] = new Item("Item 3");
    collection[4] = new Item("Item 4");
    collection[5] = new Item("Item 5");
    collection[6] = new Item("Item 6");
    collection[7] = new Item("Item 7");
    collection[8] = new Item("Item 8");

    Iterator iterator = collection.CreateIterator();

    iterator.Step = 2;

    List<string> listem = new List<string>(); // Veriyi View'e taşımak için
    ekstra liste oluşturduk.
    for (Item item = iterator.First();
        !iterator.IsDone; item = iterator.Next())
    {
        listem.Add(item.Name);
    }

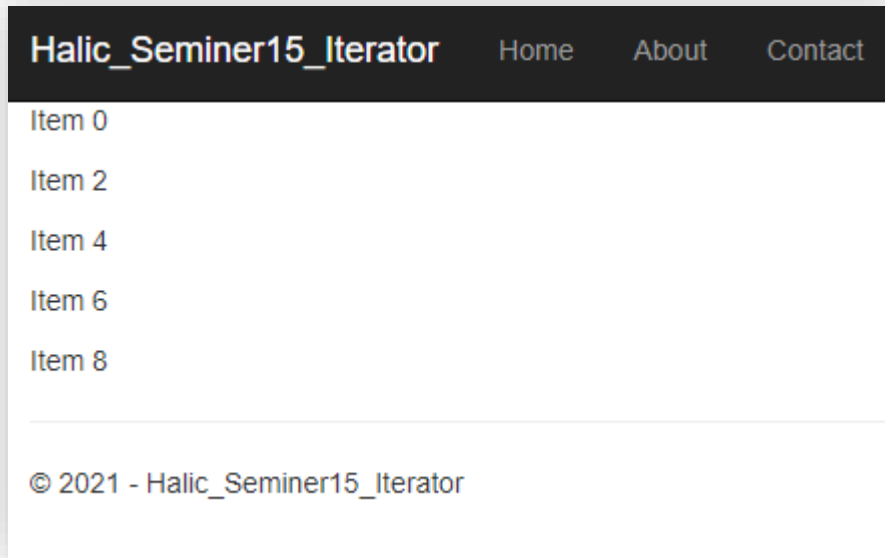
    return View(listem);
}
```

Index aksiyonumuzda ise Collection tipinde collection oluşturarak bu koleksiyon dizisine indis vererek yeni Item nesnelerinin eklenmesini sağladık. Ardından Iterator tipinde iterator nesnesini collection.CreateIterator(); metodu ile oluşturarak Step propertyesine 2 değerini göndererek itemların 0,2,4,6... şeklinde sıralanmasını sağladık. Bu doğrultuda elde edilen listeyi View'a gönderdik.

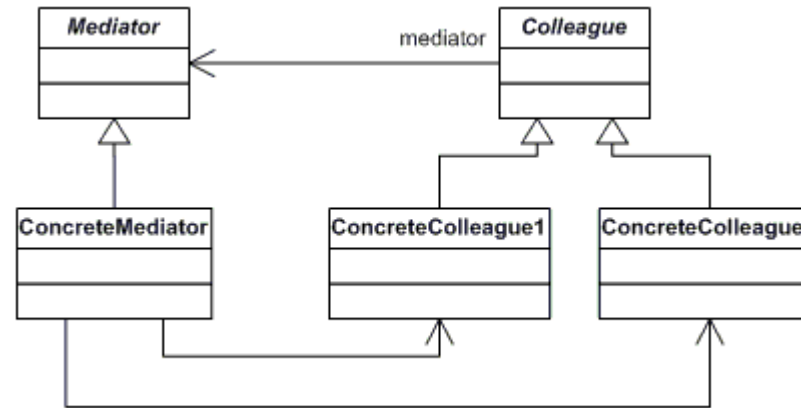
```
@model List<string>

@foreach (string item in Model)
{
    <p>@item</p>
}
```

List'e view'da foreach ile ekrana yazdırılmıştır. Tanımda yer aldığı üzere Iterator tasarım deseninde sıralama gerçekleştirilmektedir.



Mediator tasarım deseni, nesneler arasındaki bağımlılıkları tek bir merkezden yönetimi sağlayarak azaltmaya imkan tanır. Nesnelerin nasıl konuşacaklarını değil, ne konuşacaklarıyla ilgilenen bir desendir.



```
public abstract class AbstractChatroom
{
    public abstract void Register(Participant participant);
    public abstract string Send(string from, string to, string message);
}
```

```
public class Chatroom : AbstractChatroom
{
    private Dictionary<string, Participant> _participants =
        new Dictionary<string, Participant>();
    public override void Register(Participant participant)
    {
        if (!_participants.ContainsValue(participant))
        {
            _participants[participant.Name] = participant;
        }
        participant.Chatroom = this;
    }

    public override string Send(
        string from, string to, string message)
    {
        Participant participant = _participants[to];
        if (participant != null)
        {
            return participant.Receive(from, message);
        }
        return "";
    }
}
```

AbstractChatroom isimli soyut sınıfa değer döndürmeyen Register ve string değer döndüren Send soyut metodlarının imzası eklenmiştir bu soyut sınıftan türetilen Chatroom sınıfı ilgili metodları override etmiştir.

Bu kapsamda key, value taşıyan Dictionary tipli listede ilgili kullanıcı mevcut değilse yeni kullanıcı atamakla görevli olan Register metodu yine kullanıcı mevcutsa mesaj gönderme hakkı veren Send metodu bulunmaktadır.



```
public class Participant
{
    private Chatroom _chatroom;
    private string _name;

    public Participant(string name)
    {
        this._name = name;
    }

    public string Name
    {
        get { return _name; }
    }

    public Chatroom Chatroom
    {
        set { _chatroom = value; }
        get { return _chatroom; }
    }

    public string Send(string to, string message) => _chatroom.Send(_name, to,
message);
    public virtual string Receive(string from, string message) => String.Format("{0}
to {1}: '{2}'", from, Name, message);
}
```

Private erişim belirleyicisi ile Chatroom tipinde \_chatroom değişkeni, string tipinde \_name değişkeninin yer aldığı ve \_name değişkenine değerin sadece başlatıcı metod ile verildiği public erişim belirleyicili Participant sınıfında Chatroom property'si private olan \_chatroom ile get set işlemlerini gerçekleştirmektedir. Send metodu \_chatroom.Send() metoduna ilgili değişkenleri göndermekte, Receive metodu da mesajı döndürmektedir.

```
public class Beatle : Participant
{
    public Beatle(string name) : base(name)
    {
    }
    public override string Receive(string from, string message) => "To a Beatle: " +
base.Receive(from, message);
}
```

```
public class NonBeatle : Participant
{
    public NonBeatle(string name) : base(name)
    {
    }
    public override string Receive(string from, string message) => "To a non-Beatle:
"+base.Receive(from, message);
}
```

Participant sınıfından türeyen Beatle ve NonBeatle sınıflarında bulunan Receive metodu mesaj döndürmektedir.

```
public IActionResult Index()
{
    // Create Chatroom
    Chatroom chatroom = new Chatroom();
    //Create participants
    Participant Tutku = new Beatle("Tutku");
    Participant ZeynepHoca = new Beatle("Zeynep Hoca");
    //Register participants for chatroom
    chatroom.Register(Tutku);
    chatroom.Register(ZeynepHoca);
    // View'a göndermek için bu konuşmayı string listesine taşıyalım.
    List<string> chatlogs = new List<string>();
    //Chatting participants
    chatlogs.Add(Tutku.Send("Zeynep Hoca", "Nasılsınız hocam?"));
    chatlogs.Add(ZeynepHoca.Send("Tutku", "İyiyim Tutku sen nasılsın?"));
    chatlogs.Add(Tutku.Send("Zeynep Hoca", "Teşekkürler hocam.));

    return View(chatlogs);
}
```

```
@model List<string>
<br />
@foreach (string item in Model)
{
    <p>@item</p>
}
```

Ekranımıza göndereceğimiz verilerin kontrol edildiği Home Controllerımızın Index aksiyonunda Chatroom tipinde chatroom nesnesini oluşturduk. Participant Tutku ve Participant ZeynepHoca nesnelerini Beatle sınıfı ile örnekledik. Tutku.Send(hedef,mesaj) metodu ile mesajı gönderdik. Bu mesajlaşmanın tarayıcıda görüntülenmesi amacıyla mesajları chatlogs string listesine atarak view'a gönderdik ve view'da da foreach ile ekrana yazdırdık.



# Mediator

Halic\_Seminer16\_Mediator   Home   About   Contact

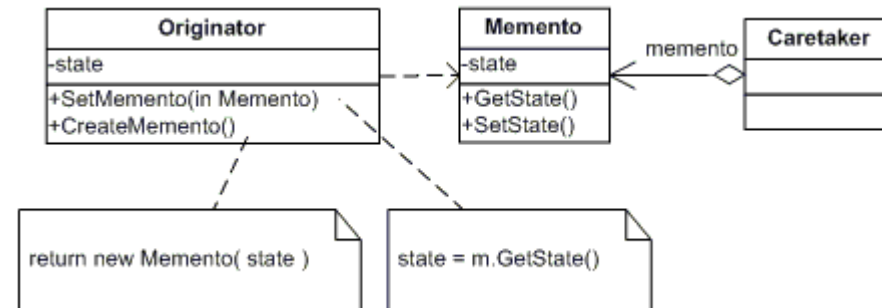
To a Beatle: Tutku to Zeynep Hoca: 'Nasılsınız hocam?'

To a Beatle: Zeynep Hoca to Tutku: 'İyiyim Tutku sen nasılsın?'

To a Beatle: Tutku to Zeynep Hoca: 'Teşekkürler hocam.'

© 2021 - Halic\_Seminer16\_Mediator

Adından da anlaşılacağı üzere memento tasarım deseni; elimizdeki mevcut nesnenin herhangi bir T anındaki durumunu kayda alarak, sonradan oluşabilecek değişiklikler üzerine tekrardan o kaydı elde etmemizi sağlayan bir desendir.



```
public class Memento
{
    private string _name;
    private string _phone;
    private double _budget;

    public Memento(string name, string phone, double budget)
    {
        this._name = name;
        this._phone = phone;
        this._budget = budget;
    }
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
    public string Phone
    {
        get { return _phone; }
        set { _phone = value; }
    }
    public double Budget
    {
        get { return _budget; }
        set { _budget = value; }
    }
}
```

\_name, \_phone, \_budget adında değişkenlerin bulunduğu ve bu değişkenlere erişimlerin property ile sağlandığı, başlatıcı metod ile il değerlerin atandığı Memento sınıfı oluşturulmuştur.



# Memento

```
public class ProspectMemory
{
    private Memento _memento;

    public Memento Memento
    {
        set { _memento = value; }
        get { return _memento; }
    }
}
```

PropectMemory sınıfı, Memento tipinde \_memento değişkenine sahip olup Memento tipinde Memento adına sahip property ile \_memento değişkenine erişim sağlanmaktadır.

```
public class SalesProspect
{
    private string _name;
    private string _phone;
    private double _budget;

    public string Name
    {
        get { return _name; }
        set
        {
            _name = value;
        }
    }

    public string Phone
    {
        get { return _phone; }
        set
        {
            _phone = value;
        }
    }

    public double Budget
    {
        get { return _budget; }
        set
        {
            _budget = value;
        }
    }

    public Memento SaveMemento() => new Memento(_name, _phone, _budget);
    public string RestoreMemento(Memento memento)
    {
        this.Name = memento.Name;
        this.Phone = memento.Phone;
        this.Budget = memento.Budget;
        return "Restoring state " + memento.Name + " " + memento.Phone + " " + memento.Budget;
    }
}
```

SalesProspect sınıfında \_name, \_phone, \_budget değişkenlerine erişim bu sınıfta da propertyler ile sağlanmış olup kapsülleme kullanılmıştır. SaveMemento metodu ile yeni Memento nesnesi oluşturulmaktadır.

RestoreMemento metodu ile de parametre ile gelen Memento tipinde memento değişkenine ait veriler SalesProspect içerisinde bulunan Name, Phone, Budget propertylerine atanmaktadır ardından string tipinde değer döndürülmektedir.



```
public IActionResult Index()
{
    SalesProspect s = new SalesProspect();
    s.Name = "Noel van Halen";
    s.Phone = "(412) 256-0990";
    s.Budget = 25000.0;

    // Store internal state
    ProspectMemory m = new ProspectMemory();
    m.Memento = s.SaveMemento();

    // Continue changing originator
    s.Name = "Leo Welch";
    s.Phone = "(310) 209-7111";
    s.Budget = 100000.0;

    // Restore saved state
    ViewBag.durum=s.RestoreMemento(m.Memento);
    return View();
}
```

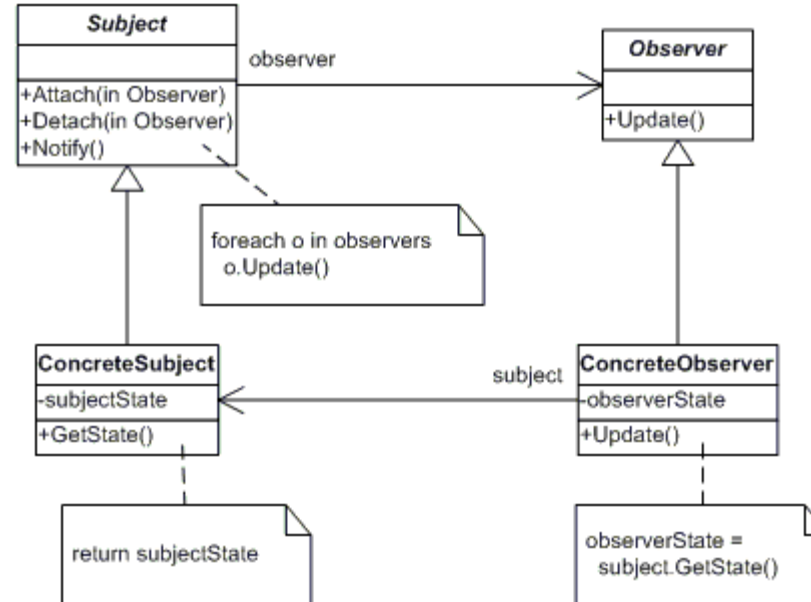
```
<br />
<p>@ViewBag.durum</p>
```

SalesProspect tipinde s nesnesi oluşturularak bu nesneye ait propertylere değerler gönderilmiştir. ProspectMemory tipinde m nesnesi oluşturularak Memento propertysi için s.SaveMemento(); metodundan veri gönderilmiştir. SaveMemento metodu new Memento(name,phone,budget); şeklindeydi. Bu örnekte mevcut s nesnesine tekrar yeni değerler atadık ardından s.RestoreMemento metodu içerisine m.Memento verisini gönderdik ve ilk verdiğimiz değer View'da karşımıza çıkmıştır.

Halic\_Seminer17\_Momento [Home](#) [About](#) [Contact](#)

Restoring state Noel van Halen (412) 256-0990 25000

Bir nesnede meydana gelen değişikliği içinde bulunduğu listedeki tüm elemanlara bildiren bir tasarım desendir. Beğendiğiniz bir ürüne indirim geldiğinde e-posta kutunuza bildirim gönderilmesi örnek olarak verilebilir.



```
public interface IInvestor
{
    string Update(Stock stock);
}
```

```
public class Investor : IInvestor
{
    private string _name;
    private Stock _stock;

    public Investor(string name)
    {
        this._name = name;
    }

    public string Update(Stock stock) => String.Format("Notified {0} of {1}'s  
"+"change to {2:C}", _name, stock.Symbol, stock.Price);

    public Stock Stock
    {
        get { return _stock; }
        set { _stock = value; }
    }
}
```

Observer adından da anlaşılacağı üzere gözlemci bir yapıdır. Bu örnekte Stock tipinde stock verisini alan string geri dönüş tipli Update metod imzalı IInvestor arayüzü kodlanmıştır.

String veri tipli \_name ve Stock tipli \_stock değişkenlerine sahip \_name için sadece Constructor metodu ile değer atanan ve IInvestor arayüzünü implement eden bu sebeple Update metodunun bulunduğu Investor sınıfı oluşturulmuştur.

```
public abstract class Stock
{
    private string _symbol;
    private double _price;
    private List<IInvestor> _investors = new List<IInvestor>();

    public Stock(string symbol, double price)
    {
        this._symbol = symbol;
        this._price = price;
    }

    public void Attach(IInvestor investor)
    {
        _investors.Add(investor);
    }

    public void Detach(IInvestor investor)
    {
        _investors.Remove(investor);
    }

    public string Notify()
    {
        foreach (IInvestor investor in _investors)
        {
            return investor.Update(this)+" <br />";
        }

        return "";
    }

    public double Price
    {
        get { return _price; }
        set
        {
            if (_price != value)
            {
                _price = value;
                Notify();
            }
        }
    }

    public string Symbol
    {
        get { return _symbol; }
    }
}
```

Stock sınıfında string tipinde `_symbol`, double tipinde `_price` değişkenleri ile `IInvestor` tipinde `_investors` listesi bulunmaktadır.

`_symbol` ve `_price` değişkenleri başlatıcı metodla ilk değerlerini almaktadır.

`IInvestor` tipinde `investor` alan `Attach` değişkeni `_investors` listesine `investor`'u eklemektedir. `Detach` metodu ise gelen `investor`u, `_investors` listesinden çıkarmaktadır.

`Notify` metodu `_investors` listesinde `foreach` ile gezen koda sahiptir öğelerin aldığı değerlerin `Update` metodu ile string değer döndürmesini sağlar. `Price` property'si ise set işleminde mevcut fiyatla yeni fiyat arasında fark varsa `Notify` metodunu çalıştırır. `Notify`, string döndüğünden sisteme bildirim verme görevini üstlenir.

`Symbol` property'si ise sadece `_symbol` değerini döndürür. Set işlemi kapalıdır.

```
public class IBM:Stock
{
    public IBM(string symbol, double price): base(symbol, price)
    {
    }
}
```

```
public IActionResult Index()
{
    // Create IBM stock and attach investors
    IBM ibm = new IBM("IBM", 120.00);
    ibm.Attach(new Investor("Sorros"));
    ibm.Attach(new Investor("Berkshire"));

    // Fluctuating prices will notify investors

    ibm.Price = 120.10;
    //ViewBag.durum = ibm.Notify();
    ibm.Price = 121.00;
    //ViewBag.durum = ibm.Notify();
    ibm.Price = 120.50;
    //ViewBag.durum = ibm.Notify();
    ibm.Price = 120.75;
    ViewBag.durum=ibm.Notify();

    return View();
}
```

Stock sınıfından türetilen IBM sınıfında ise ata sınıfından gelen değişkenlere veri atayan başlatıcı metod bulunmaktadır.

Index aksiyonunda IBM tipinde ibm nesne örneği alındı. IBM ve 120.00 değerleri başlatıcı metoda verildi ardından Attach metoduna Sorros ve Berkshire adlarını taşıyan Investorler gönderildi. Bu investorler, Stock sınıfında yer alan listeye eklenmiş olduar. Ardından Price property'si ile set edilen her fiyatta Notify ile dönüş alındı. Bu durum ViewBag ile index view'ine gönderildi.

```
<br />  
<p>@Html.Raw(ViewBag.durum)</p>
```

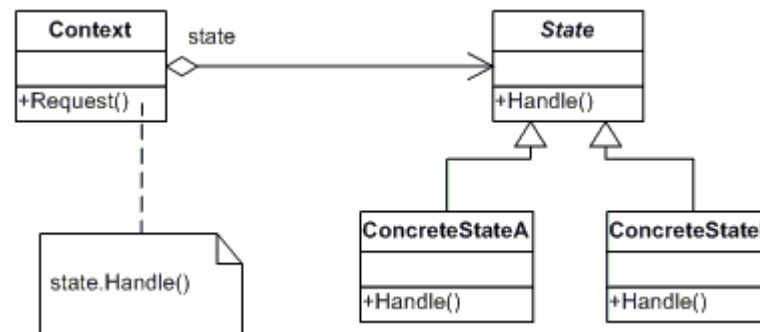
Halic\_Seminer18\_Observer [Home](#) [About](#) [Contact](#)

Notified Sorros of IBM's change to ₺120,75

© 2021 - Halic\_Seminer18\_Observer

ViewBag ile gelen bildirim ekrana yazdırıldı. Özetle bu örnekte fiyat değişikliğini ilgili yerlere iletme söz konusudur. Örneğin bir e-ticaret sitesinde favorilerinize eklediğiniz üründe fiyat değişikliği oluşursa bizlere bu tasarım ile e-posta, sms veya telefon bildirimi yapılabilir.

Durum modeli, bir nesnenin dahili durumu değiştiğinde davranışını da değiştirmesine izin veren davranışsal bir yazılım tasarım modelidir.



```
public abstract class State
{
    protected Account account;
    protected double balance;
    protected double interest;
    protected double lowerLimit;
    protected double upperLimit;
    public Account Account
    {
        get { return account; }
        set { account = value; }
    }
    public double Balance
    {
        get { return balance; }
        set { balance = value; }
    }

    public abstract void Deposit(double amount);
    public abstract void Withdraw(double amount);
    public abstract void PayInterest();
}
```

State soyut sınıfında protected erişim belirleyicisi ile Account tipinde account, double tiplerinde balance, interest, lowerLimit, upperLimit değişkenleri oluşturulmuştur. account ve balance değişkenleri için property ile kapsülleme yapılmıştır. Deposit, Withdraw, PayInterest ile metod imzaları oluşturulmuştur.



```
public class Account
{
    private State _state;
    private string _owner;
    public Account(string owner)
    {
        this._owner = owner;
        this._state = new SilverState(0.0, this);
    }
    public double Balance
    {
        get { return _state.Balance; }
    }
    public State State
    {
        get { return _state; }
        set { _state = value; }
    }
    public string Deposit(double amount)
    {
        _state.Deposit(amount);
        return String.Format("Deposited {0:C} --- Balance = {1:C} Status = {2}", amount, this.Balance,
this.State.GetType().Name);
    }
    public string Withdraw(double amount)
    {
        _state.Withdraw(amount);
        return String.Format("Withdrew {0:C} --- Balance = {1:C} Status = {2}", amount, this.Balance,
this.State.GetType().Name);
    }
    public string PayInterest()
    {
        _state.PayInterest();
        return String.Format("Interest Paid --- --- Balance = {0:C} Status = {1}", this.Balance,
this.State.GetType().Name);
    }
}
```

Account sınıfında State tipinde \_state, string tipinde \_owner değişkenleri oluşturularak başlatıcı metoddan \_owner değişkeni için string tipinde değer beklemektedir.

Balance propertysi State soyut sınıfından türetilen State alt sınıflarında tanımlanmıştır ve burada bulunan property sadece Balance değerini döndürmektedir. State State propertysi ise get set işlemlerine açıktır. Deposit metodu gelen double tipindeki veriyi \_state.Deposit metoduna göndermektedir. Withdraw ve PayInterest metodları da Deposit metodu ile aynı davranışı sergilemektedir.

```
public class RedState : State
{
    private double _serviceFee;
    public RedState(State state)
    {
        this.balance = state.Balance;
        this.account = state.Account;
        Initialize();
    }

    private void Initialize()
    {
        interest = 0.0;
        lowerLimit = -100.0;
        upperLimit = 0.0;
        _serviceFee = 15.00;
    }

    public override void Deposit(double amount)
    {
        balance += amount;
        StateChangeCheck();
    }

    public override void Withdraw(double amount)
    {
        amount = amount - _serviceFee;
    }

    public override void PayInterest()
    {
        // No interest is paid
    }

    private void StateChangeCheck()
    {
        if (balance > upperLimit)
        {
            account.State = new SilverState(this);
        }
    }
}
```

State soyut sınıfından türetilen RedState, SilverState, GoldState sınıflarında temelde Initialize metodunda değişiklik söz konusudur.

RedState metodu double tipinde \_serviceFee değişkeni tanımlamıştır. Başlatıcı metodunda State tipinde state parametresi ile veri talep etmektedir. Gelen state ile balance, account değişkenlerine veri aktarım Initialize metodunu çağırılmaktadır. Initialize metodunda ata sınıfından gelen interest, lowerLimit, upperLimit ve kendisinde bulunan \_serviceFee değişkeni için değer atama işlemi yapılmaktadır.

Deposit metodu ise balance değişkeninin değerine parametre ile gelen amount değerini eklemektedir ardından StateChanceCheck metodunu çağırılmaktadır. Withdraw metodu parametre ile aldığı amount değerine, yine amountun \_serviceFee değeri ile farkını eklemektedir. PayInterest metodunda herhangi bir işlem yapılmamaktadır. StateChangeCheck metodunda ise balance değeri upperLimit değerinden yüksek ise accountta bulunan State popertysine SilverState nesnesi oluşturularak gönderilir. Çünkü Silver bir üst sınıftır sebebi ise upperLimit'i daha yüksektir.

```
public class SilverState : State
{
    public SilverState(State state) :
        this(state.Balance, state.Account)
    {
    }

    public SilverState(double balance, Account account)
    {
        this.balance = balance;
        this.account = account;
        Initialize();
    }

    private void Initialize()
    {
        interest = 0.0;
        lowerLimit = 0.0;
        upperLimit = 1000.0;
    }

    public override void Deposit(double amount)
    {
        balance += amount;
        StateChangeCheck();
    }

    public override void Withdraw(double amount)
    {
        balance -= amount;
        StateChangeCheck();
    }

    public override void PayInterest()
    {
        balance += interest * balance;
        StateChangeCheck();
    }

    private void StateChangeCheck()
    {
        if (balance < lowerLimit)
        {
            account.State = new RedState(this);
        }
        else if (balance > upperLimit)
        {
            account.State = new GoldState(this);
        }
    }
}
```

RedState sınıfında anlatıldığı üzere SilverState sınıfında bulunan Initialize metodunda upperLimit 1000.0'dır, RedState sınıfında ise bu değer 0.0'dır.

Bu sınıfta bulunan StateChangeCheck metouda da balance değeri lowerLimit değerinden düşük ise state propertyisine RedState nesnesi oluşturularak gönderilmektedir. balance değeri upperLimit'ten yüksek ise daha yüksek değeri taşıyan GoldState sınıfı oluşturularak state propertyisi güncellenmektedir.

Yine PayInterest metodu RedState sınıfındaki aksine bu sınıfta işlem yapmaktadır.

```
public class GoldState : State
{
    public GoldState(State state)
        : this(state.Balance, state.Account)
    {
    }

    public GoldState(double balance, Account account)
    {
        this.balance = balance;
        this.account = account;
        Initialize();
    }

    private void Initialize()
    {
        interest = 0.05;
        lowerLimit = 1000.0;
        upperLimit = 1000000.0;
    }

    public override void Deposit(double amount)
    {
        balance += amount;
        StateChangeCheck();
    }

    public override void Withdraw(double amount)
    {
        balance -= amount;
        StateChangeCheck();
    }

    public override void PayInterest()
    {
        balance += interest * balance;
        StateChangeCheck();
    }

    private void StateChangeCheck()
    {
        if (balance < 0.0)
        {
            account.State = new RedState(this);
        }
        else if (balance < lowerLimit)
        {
            account.State = new SilverState(this);
        }
    }
}
```

GoldState daha önce işlenen RedState ve SilverState sınıflarında yer alan Initialize metodunun taşıdığı lowerLimit, upperLimit değerlerine göre üst seviyede olan bir sınıftır.

Bu sınıfta tanımlanan StateChangeCheck metodunun içeriğinde ise balance değeri 0'dan küçük ise state propertyesine RedState örneği oluşturularak gönderilir, balance değeri lowerLimit'ten küçük ise SilverState örneği alınarak state propertyesine gönderilir.

```
public IActionResult Index()
{
    // Open a new account
    Account account = new Account("Jim Johnson");

    // Apply financial transactions
    account.Deposit(500.0);
    account.Deposit(300.0);
    account.Deposit(550.0);
    account.PayInterest();
    account.Withdraw(2000.00);
    ViewBag.durum = account.Withdraw(1100.00);

    return View();
}
```

```
<br />
<p>@ViewBag.durum</p>
```

Index aksiyonumuzda ise Account tipinde account nesnesi oluşturularak gönderilen isim parametresine Jim Johnson string değişkenini gönderdik. Deposit metodlarına değerler atadık deposite gönderilen değerler balance değerine eklenip StateChangeCheck metodunda değerlendirilmekteydi.

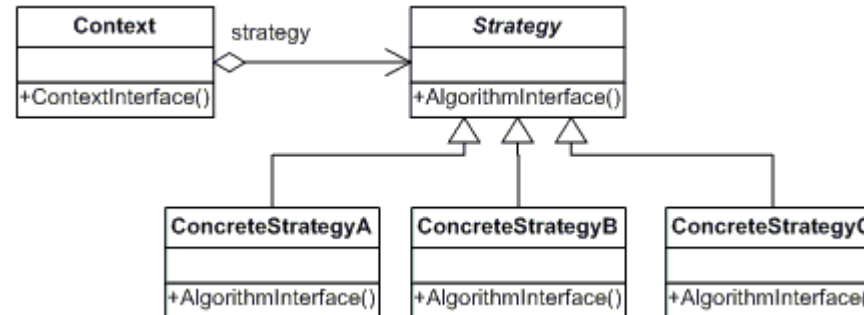
Gerçekleştirilen işlemler sonucunda elde edilen değer ViewBag'e atanarak View'da ekrana yazdırılmıştır.

Halic\_Seminer19\_State   Home   About   Contact

Withdrew ₺1.100,00 --- Balance = -₺582,50 Status = RedState

© 2021 - Halic\_Seminer19\_State

Çalışma zamanında bir algoritma seçilmesini sağlayan davranışsal bir yazılım tasarım modelidir(Runtime esnasında bir objenin davranışını değiştirmemize imkan verir). Kod, tek bir algoritmayı doğrudan uygulamak yerine, bir algoritma ailesinde kullanılacak çalışma zamanı talimatlarını alır. Özetle bir işlem için bir algoritma değil birden çok algoritma kullanıldığında bu desen tercih edilir.



```
public abstract class SortStrategy
{
    public abstract string Sort(List<string> list);
}
```

```
public class QuickSort : SortStrategy
{
    public override string Sort(List<string> list)
    {
        list.Sort();
        return "QuickSorted list ";
    }
}
```

```
public class ShellSort : SortStrategy
{
    // ShellSorted algoritması ile sıralama yapıp döndürebiliriz.
    public override string Sort(List<string> list) => "ShellSorted list ";
}
```

```
public class MergeSort : SortStrategy
{
    //MergeSorted ile sıralama yapıp döndürebiliriz.
    public override string Sort(List<string> list) => "MergeSorted List";
}
```

Bu örnekte sıralama algoritmaları kullanılmıştır ancak listeler için algoritmalar yazılmamıştır tasarım deseninin örneğinin anlaşılması açısından kullanılan Algoritma bilgisi string olarak ekrana yazılmıştır.

Bu kapsamda SortStrategy soyut sınıfında string değer dönen Sort metod imzası oluşturulmuştur. SortStrategy sınıflarından QuickSort, ShellSort, MergeSort sınıfları türetilerek Sort metodlarına dönüş tipi olarak kendi Sort adlarını yazmışlardır.

Sadece QuickSort sınıfında hazır Sort methodu mevcuttur.

```
public class SortedList
{
    private List<string> _list = new List<string>();
    private SortStrategy _sortstrategy;

    public void SetSortStrategy(SortStrategy sortstrategy)
    {
        this._sortstrategy = sortstrategy;
    }

    public void Add(string name)
    {
        _list.Add(name);
    }

    public string Sort()
    {
        string metin = "";
        metin = _sortstrategy.Sort(_list);
        foreach (string name in _list)
        {
            metin += "<br /> " + name;
        }
        return metin;
    }
}
```

SortedList sınıfında string listesi ve SortStrategy tipinde \_sortstrategy değişkenine sahiptir. SetSortStrategy metodu ile \_sortstrategy değişkenine veri atamaktadır. Add metodu ile gelen string değerleri string listesine eklemektedir. Sort metodu ile de \_sortstrategy değişkenine atanmış olan stratejiye göre (Quick, Shell, Merge) sıralama yaparak listeden elde ettiği string değerleri döndürmektedir. Biz bu örnekte ilgili listeyi ekrana yazabilmek adına tek bir string değişkeni içerisine bu listeyi ekledik.



```
public IActionResult Index()
{
    SortedList studentRecords = new SortedList();

    studentRecords.Add("Tutku");
    studentRecords.Add("Ahmet");
    studentRecords.Add("Gökçe");
    studentRecords.Add("Neşe");
    studentRecords.Add("Onur");

    studentRecords.SetSortStrategy(new QuickSort());
    ViewBag.durum1=studentRecords.Sort();

    studentRecords.SetSortStrategy(new ShellSort());
    ViewBag.durum2 = studentRecords.Sort();

    studentRecords.SetSortStrategy(new MergeSort());
    ViewBag.durum3 = studentRecords.Sort();

    return View();
}
```

Index aksiyonunda SortedList tipinde studentRecords adında liste tanımlayarak Add metoduyla Tutku, Ahmet, Gökçe, Neşe, Onur isimlerini ekledik. SetSortStrategy metodu ile 3 sıralama algoritması için de nesne örneği oluşturarak elde edilen değeri ViewBag'lere atadık. (ViewBag'ler string değer taşırlar. Bu yüzden de listemizi önceki sayfalarda anlatıldığı üzere string değişkenine atamıştık.) ViewBag'ler Viewlarımızda rahatça görüntülenmektedir.

```
<div style="margin-top:10px;">  
  <p>@Html.Raw(ViewBag.durum1)</p>  
  <p>@Html.Raw(ViewBag.durum2)</p>  
  <p>@Html.Raw(ViewBag.durum3)</p>  
</div>
```

Index.cshtml dosyasında Html içeren ViewBaglerin görüntülenmesi için @Html.Raw() metoduna parametre olarak göndererek ekrana taşınan listeyi yazdırdık.

Halic\_Seminer20\_Strategy   Home   About   Contact

QuickSorted list

Ahmet  
Gökçe  
Neşe  
Onur  
Tutku

ShellSorted list

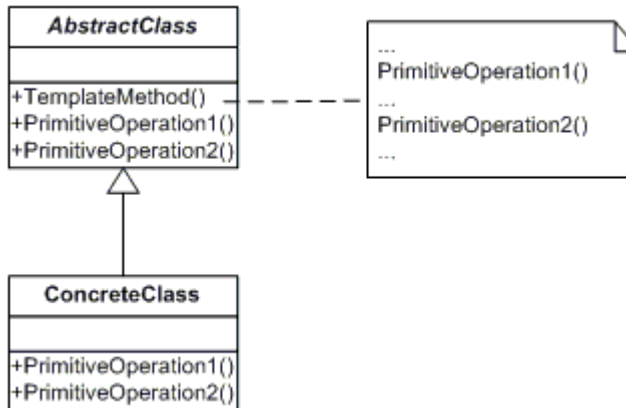
Ahmet  
Gökçe  
Neşe  
Onur  
Tutku

MergeSorted List

Ahmet  
Gökçe  
Neşe  
Onur  
Tutku

Sonuç olarak bu örnekte bir işlem için birden çok algoritma kullanan yapı oluşturuldu.

Şablon metot kalıbı bir işlem için gerekli adımları soyut olarak tanımlar ve bir şablon metot ile algoritmanın nasıl çalışacağını belirler. Alt sınıflar algoritma için gerekli bir ya da birden fazla metodu kendi bünyelerinde gerçekleyerek kullanılan algoritmanın kendi istekleri doğrultusunda çalışmasını sağlarlar. Böylece hem kod tekrarının önüne geçilerek kodun tekrar kullanılabilirliği hem de algoritma iskeletinde yapılacak bir düzenlemenin tek bir yerden yapılması sağlanır.



```
public abstract class AbstractAlisveris
{
    protected string UrunAdi;
    protected OdemeTipi odemeTipi;
    string Baslat() => "Alışveriş Başladı";
    string Bitir() => $"Alışveriş Bitti. {UrunAdi} {odemeTipi} yöntemiyle
alınmıştır.";
    abstract public void Urun();
    abstract public void OdemeSekli();
    public string TemplateMethod()
    {
        Baslat();
        Urun();
        OdemeSekli();
        Bitir();
        return Baslat() + " ... " + Bitir();
    }
}
```

```
public class Buzdolabi : AbstractAlisveris
{
    public override void OdemeSekli()
    {
        odemeTipi = OdemeTipi.Taksit;
    }

    public override void Urun()
    {
        UrunAdi = "Buzdolabı";
    }
}
```

OdemeTipi Pesin ve Taksit olarak tanımlanan enumdan sonra soyut AbstractAlisveris sınıfı oluşturuldu. Bu sınıfta string tipinde UrunAdi, OdemeTipi tipinde odemeTipi değişkenleri oluşturularak Baslat() metodu ile alışveriş başladı, Bitir Metodu ile de alışveriş bitti mesajı verilerek ürün adı ve ödeme tipi döndürüldü. Urun, OdemeSekli metod imzaları oluşturulurken. TemplateMethod metodunda Baslat, Urun, OdemeSekli, Bitir metodları çalıştırılarak string değer döndürüldü.

Buzdolabı AbstractAlisveris soyut sınıfından türetildi ve Ürün adı ile Ödeme şekli bilgisi ilgili metodlara girildi.

```
public class Televizyon : AbstractAlisveris
{
    public override void OdemeSekli()
    {
        odemeTipi = OdemeTipi.Pesit;
    }

    public override void Urun()
    {
        UrunAdi = "Televizyon";
    }
}
```

```
public IActionResult Index()
{
    AbstractAlisveris alisveris1 = new Televizyon();
    ViewBag.Alisveris1=alisveris1.TemplateMethod();

    AbstractAlisveris aliveris2 = new Buzdolabi();
    ViewBag.Alisveris2 = aliveris2.TemplateMethod();

    return View();
}
```

```
<div style="margin-top:10px;">
    <p>@ViewBag.Alisveris1</p>
    <p>@ViewBag.Alisveris2</p>
</div>
```

Televizyon sınıfında da Buzdolabi sınıfından farklı olarak Urun adı ve Ödeme şekli bilgisi girildi. Index aksiyonunda AbstractAlisveris tipinde alisveris1 nesnesi Televizyon sınıfı ile örnekledi yine AbstractAlisveris tipinde alisveris2 nesnesi Buzdolabi sınıfı ile örnekledi.

TemplateMethod() metodundan gelen string ViewBag'lere atandı ve View'a gönderilerek aşağıdaki sonuç elde edildi.

[Halic\\_Seminer21\\_Template](#) [Home](#) [About](#) [Contact](#)

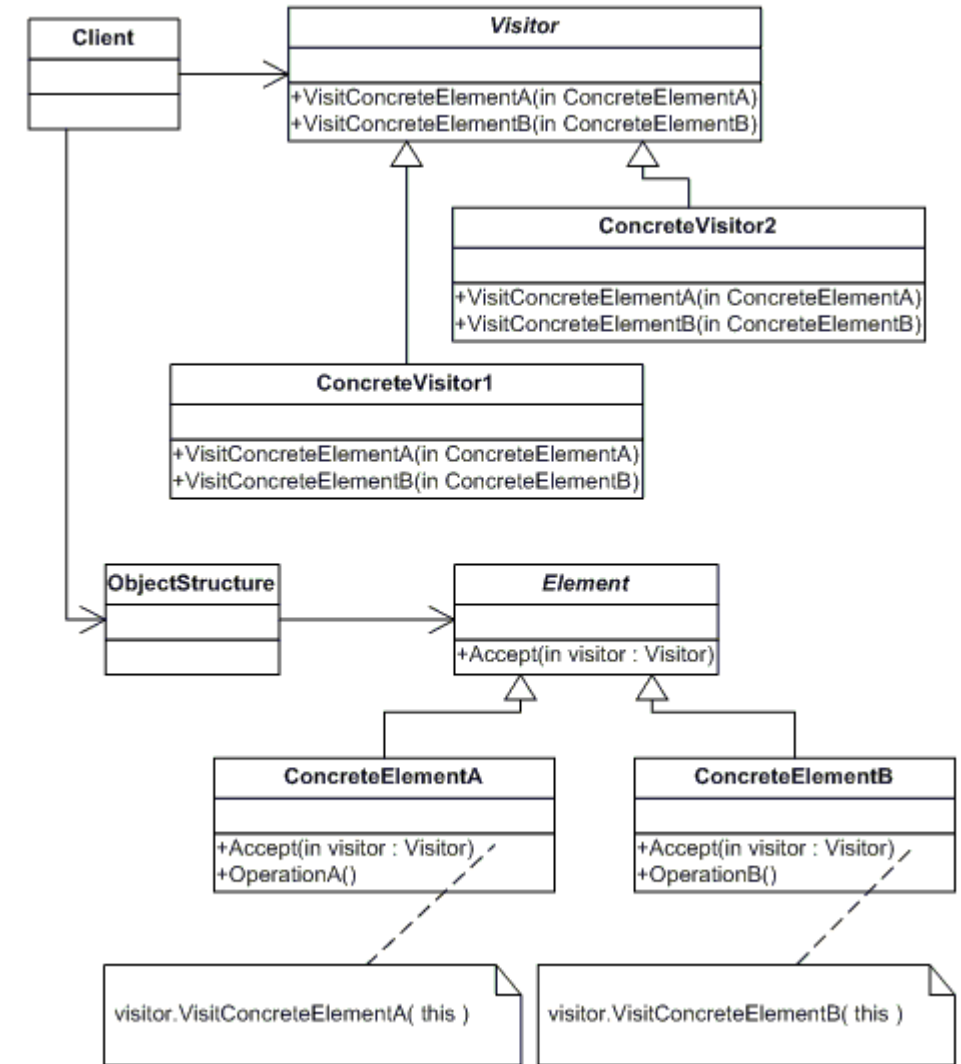
Alışveriş Başladı ... Alışveriş Bitti. Televizyon Pesit yöntemiyle alınmıştır.

Alışveriş Başladı ... Alışveriş Bitti. Buzdolabi Taksit yöntemiyle alınmıştır.

© 2021 - Halic\_Seminer21\_Template

Visitor design pattern sınıflara, sınıfların içerisinde değişiklik yapmadan fonksiyonellik ekleme imkanı sunan desendir.

Bir sınıf hiyerarşisinde yer alan sınıflar üzerinde değişiklik yapmadan, bu sınıflar yeni metodların eklenmesini kolaylaştırır. İstenilen metod bir visitor sınıfında implemente edilir.



```
public interface IVisitor
{
    string Visit(Element element);
}
```

```
public abstract class Element
{
    public abstract string Accept(IVisitor visitor);
}
```

```
public class Employee : Element
{
    private string _name;
    private double _income;
    private int _vacationDays;

    public Employee(string name, double income,int vacationDays)
    {
        this._name = name;
        this._income = income;
        this._vacationDays = vacationDays;
    }

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    public double Income
    {
        get { return _income; }
        set { _income = value; }
    }

    public int VacationDays
    {
        get { return _vacationDays; }
        set { _vacationDays = value; }
    }

    public override string Accept(IVisitor visitor) => visitor.Visit(this);
}
```

IVisitor arayüzünde string geri dönüş tipli Element tipinde element parametresine sahip Visit metod imzası bulunmaktadır. Element soyut sınıfında da IVisitor tipinde visitor alan Accept metodu oluşturulmuştur.

Employee sınıfı Element soyut sınıfından türetilmiş, string tipinde \_name, double tipinde \_income, int tipinde \_vacationDays değişkenlere sahiptir. Bu değişkenlere ilk veri başlatıcı metod ile atanmaktadır; ayrıca bu değişkenlere property ile dışarıdan erişim mümkündür. (Kapsülleme) Override edilen Accept metodu ise Employee sınıf içeriğini Visit metoduna döndürmektedir.

```
public class Employees
{
    private List<Employee> _employees = new List<Employee>();

    public void Attach(Employee employee)
    {
        _employees.Add(employee);
    }

    public void Detach(Employee employee)
    {
        _employees.Remove(employee);
    }

    public string Accept(IVisitor visitor)
    {
        string metin = "";
        foreach (Employee e in _employees)
        {
            metin+="  
>" + e.Accept(visitor);
        }
        return metin;
    }
}
```

Employees sınıfı Employee tipinde `_employees` adında bir listeye sahiptir. Listeye employee eklemek için `Attach`, listeden employee silmek için `Detach` metodları bulunmaktadır. `Accept` metodu ise `IVisitor` tipinde visitor almaktadır. `Accept` metodu içerisinde bulunan `_employees` listesi üzerinde `foreach` ile dönmektedir. Gelen visitor nesnesini döndüğü employee içinde bulunan `Accept` metoduna göndermektedir ve dönen değeri string tipli metin değişkenine eklemektedir.



```
public class IncomeVisitor : IVisitor
{
    public string Visit(Element element)
    {
        Employee employee = element as Employee;
        // Provide 10% pay raise
        employee.Income *= 1.10;
        return String.Format("{0} {1}'s new income: {2:C}", employee.GetType().Name,
employee.Name, employee.Income);
    }
}
```

```
public class VacationVisitor : IVisitor
{
    public string Visit(Element element)
    {
        Employee employee = element as Employee;
        // Provide 3 extra vacation days
        employee.VacationDays += 3;
        return String.Format("{0} {1}'s new vacation days: {2}",
employee.GetType().Name, employee.Name, employee.VacationDays);
    }
}
```

InComeVisitor ve Vacation sınıfları IVisitor arayüzünü implement etmişlerdir. Oluşturulması zorunlu olan InComeVisitor sınıfındaki Visit metodunda Employee tipinde employee adındaki nesne için parametre ile gelen element as ile Employee tipine benzetilmiştir. Buradan hareketle employee nesnesinin InCome propertyisine yeni maaş bilgisi set edilmiştir. VacationVisitor sınıfında ise Visit metodunda aynı employee yapısı oluşturulmuş employee nesnesinde bulunan VacationDays propertyisine +3 gün eklenmiştir.

Visit metodları elde ettiği son bilgiyi string formatında döndürmektedir.

```
class Clerk : Employee
{
    public Clerk(): base("Hank", 25000.0, 14){}
}

class Director : Employee
{
    public Director(): base("Elly", 35000.0, 16){}
}

class President : Employee
{
    public President() : base("Sandy", 45000.0, 21){}
}
```

```
public IActionResult Index()
{
    Employees e = new Employees();
    e.Attach(new Clerk());
    e.Attach(new Director());
    e.Attach(new President());

    // Employees are 'visited'
    ViewBag.durum1 = e.Accept(new IncomeVisitor());
    ViewBag.durum2 = e.Accept(new VacationVisitor());

    return View();
}
```

Employee sınıfı name, income, vacationdays parametrelerini alan başlatıcı metoda sahipti bu sebeple bu sınıftan türeyen Clerk, Director, President sınıfları başlatıcı metodlarında ilgili parametreleri kullanmışlardır.

Index aksiyonunda iste Employees tipinde e nesnesi oluşturulmuştur. e.Attach() metodu ile Employee sınıfından türemiş olan Clerk, Director, President sınıfları Employees sınıfındaki listeye eklenmiştir. Ardından e.Accept metoduna InComeVisitor (+%10 maaş) ve VacationVisitor (+3 gün tatil/izin) sınıfları gönderilmiştir.



```
<div style="margin-top:10px">  
  <p>@Html.Raw(ViewBag.durum1)</p>  
  <p>@Html.Raw(ViewBag.durum2)</p>  
</div>
```

Bu kapsamda Clerck, Director, President sınıflarında oluşturulan bireylerin maaşları %10 artırılmış, tatil günlerine de 3 gün eklenmiştir.

Halic\_Seminer22\_Visitor

[Home](#)

[About](#)

[Contact](#)

Clerk Hank's new income: ₺27.500,00

Director Elly's new income: ₺38.500,00

President Sandy's new income: ₺49.500,00

Clerk Hank's new vacation days: 17

Director Elly's new vacation days: 19

President Sandy's new vacation days: 24

© 2021 - Halic\_Seminer22\_Visitor



# Kaynaklar

- DoFactory .Net Design Patterns <https://www.dofactory.com/net/design-patterns>
- Onur Dayıbaşı - Tasarım Örüntüleri (Design Patterns) Medium Blog Yazısı <https://medium.com/design-patterns/design-patterns-tasar%C4%B1m-%C3%B6r%C3%BCnt%C3%BCleri-3b0f67ae6488>
- C#Corner <https://www.c-sharpcorner.com/search.aspx?q=design%20patterns>
- Gençay YILDIZ Blog Sitesi – Design Patterns Yazıları <https://www.gencayyildiz.com/blog/category/design-pattern/>