YAZILIM TASARIM DESENLERİ VE PRENSİPLERİ C# İLE

Software Design Patterns And Principles

Gang Of Four Design Patterns

SOLID Design Principles

ÖNSÖZ

Bu dokümanda tasarım desenleri ve tasarım prensipleri konuları anlatılmaktadır.
Gangs of Four (GOF) olarak tanımlanan 23 desen ve SOLID olarak tanımlanan 5 prensip incelenmiştir. Örnek uygulamalarda C# dili kullanılmıştır.
Tasarım desenleri konusunda dofactory.com referans noktası olarak kullanılmıştır.
Bu özenle hazırlanmış bir eğitim dökümanı değildir. Konularla ilgili Türkçe kaynak az bulunduğundan ve birilerinin işine yarar düşüncesiyle geçmiş dönemlerdeki kişisel notlarımın toparlanmış, doküman haline getirilmiş bir versiyonudur. Bu konulara ilgi duyan kişiler için giriş ve kavramları anlamaları için yeterli bir düzeydedir. İlk etapta işlerini kolaylaştıracaktır. İlerletmek isteyen kişiler daha fazla sayıda kullanım örneklerini inceleyebilir ve pekiştirmek adına kodlarında bu yapıları kullanabilir.

Mehmet Taşköprü
mtaskopru@gmail.com

İÇERİK

Creational Patterns

Structural Patterns

Behavioral Patterns

Class Design Principles - SOLID

Design Pattern

Türkçe'ye çevrilmiş anlamı Tasarım deseni | örüntüsü | şablonu şeklinde, farklı kaynaklarda kullanılmaktadır.
Yazılım geliştirme sürecinde düşüncelerimizi koda dönüştürmeden önce yazılım mimarisini nasıl tasarlayacağımızı kurgularız. OOP ile kurduğumuz bu soyut dünyada yazdığımız kodları bir kalıba sokacak, standartlaştıracak bir kavram arayışı içerisine gireriz.
Tasarım desenleri çok sık karşılaşılan birbirine benzer yada aynı problemleri çözmek için geliştirilmiş, çözümü konusunda doğruluğu sabitlenmiş en basit en efektif çözüm yolları sunar.
Problem ve çözümü, mimari yapının tasarımı sırasında ortaya çıkan tasarımsal sorunlar olarak algılanmalı algoritmalar ile karıştırılmamalıdır.
Tasarım desenleri sayesinde mimari herkesce bilinen ortak bir yapıya sahip olur. Kod okunabilirliğini ve insanların yapıyı anlamalarını kolaylaştırır. Tasarım desenleri programcıların karşılaştıkları ortak problemler ve bu problemlerin çözümünü temsil eden paylaştıkları ortak deneyimler olarak ifade edilebilir. 1987 yılında OOPSLA konferansı yapılana kadar bu konuda herhangi bir çalışma ortaya çıkmamış. Bu konferans sonrasında bazı makale ve sunumlar yayınlanmıştır.
1994 yılında Erich Gamma, Ralph Johnson, John Vlissides ve Richard Helm tarafından Design Patterns: Elements of Reusable Object-Oriented Software kitabı yayınlanmış ve bu bir başlangıç noktası oluşturmuştur. 3 gurupta 23 tasarım deseni yayınlamışlardır.
Bu 4 kişilik yazar kadrosu (Gang Of Four) Dörtlü Çete/Silahşör olarak anılır.

Tasarım Desenleri 3 ana grup altında incelenir.

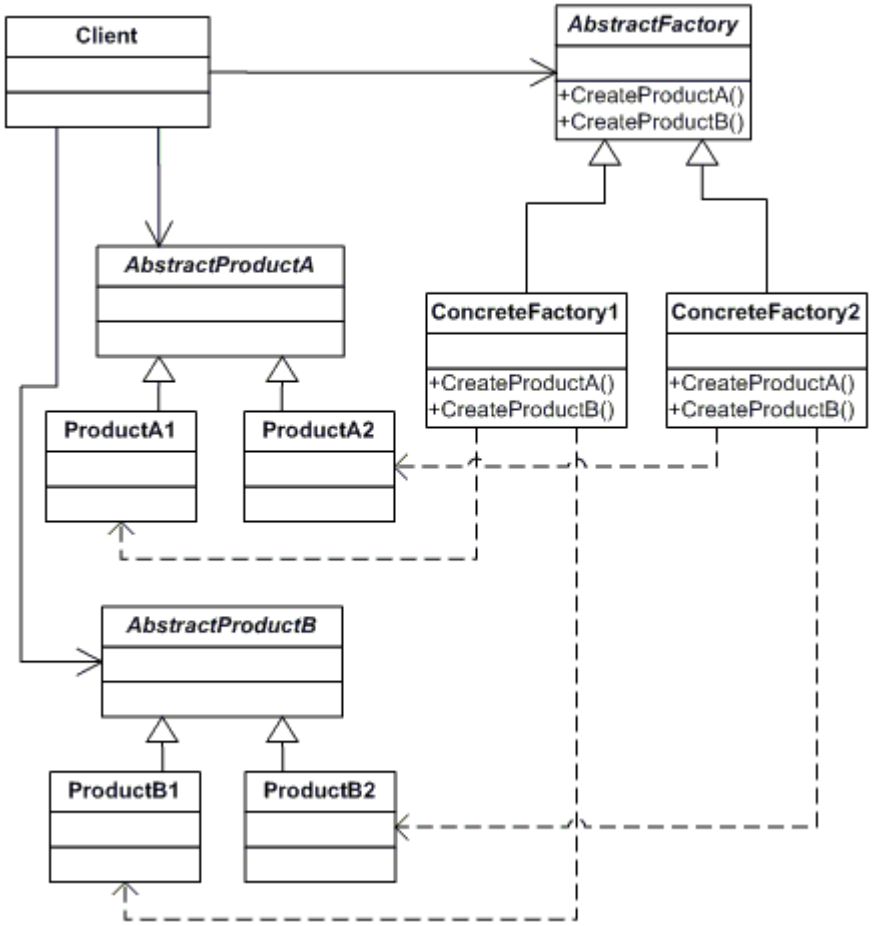
1 - Creational Patterns : Bir yada daha fazla nesnenin oluşturulması ve yönetilmesi ile ilgili desenler.
2 - Behavioral Patterns : Belirli bir işi yerine getirmek için çeşitli sınıfların nasıl birlikte davranış göstereceğini belirleyen desenler.
3 - Structural Patterns : Nesnelerin birbirleri ile olan ilişkileri temel alınarak tasarlanan desenler.

Tasarım desenleri programlama dilinden bağımsızdır. Yapı aynı fakat programlama dili seçimine göre yazılış şekilleri farklıdır.

Abstract Factory

Creational Pattern Gurubunda yer alır.

Soyut fabrika üretim yapılarının oluşturulması ve aralarında ilişki bulunan nesnelerin soyut fabrika yapıları aracılığı ile üretimi ele alınmaktadır. İstemciler tiplerine göre farklı fabrikaları kullanıp seçebilir. İstemcinin ihtiyacı farklı fabrikalar tarafından ele alınıp işlenerek ürünler oluşturulur.



AbstractFactory
ConcreteFactory tipinin şablonu. Soyut fabrika.

ConcreteFactory
Asıl Fabrika

AbstractProduct
Product tipinin şablonu. Soyut ürün.

Product
Asıl Ürün

Client
İstemci uygulamanın ulaştığı ara katman. Soyut Fabrika nesnesi üzerinden ürün üretir. Nesnenin üretiminden AbstractFactory arayüzünü uygulayan ConcreteFactory tipi sorumludur.

Ürün AbstractProduct arayüzü uygulayan Product tipidir.

Üretici ise client uygulamadır.

Soyut Fabrika tipi (Abstract Factory) 2 metod içermektedir. Bunlar A ve B tipli ürünlerin üretimini sağlayan Create metodlarıdır.Asıl Fabrikalar (ConcreteFactory) Abstract Factory tipini uygularlar. Ve bu tipi uygulamaları ile A ve B tipli Create metodlarını içlerinde bulundurmaları gerekmektedir. Bu metotlar soyut ürün tipinden (AbstractProduct) geri dönüş değerine sahiptir. 2 Soyut ürün yapısı AbstractProductA ve AbstractProductB oluşturulmuş bu yapılar diğer sınıflar tarafından uygulanmıştır. Son olarak çalıştırma aşamasında Soyut Fabrikalar oluşturulmuş ve client üzerine parametre geçilmiş ve çalıştırılmıştır.

```
namespace ConsoleApplication1
{
    // Abstract Factory
    abstract class AbstractFactory
    {
        public abstract AbstractProductA CreateProductA();
        public abstract AbstractProductB CreateProductB();
    }

    // ConcreteFactory1
    class ConcreteFactory1 : AbstractFactory
    {
        public override AbstractProductA CreateProductA()
        {
            return new ProductA1();
        }
        public override AbstractProductB CreateProductB()
        {
            return new ProductB1();
        }
    }

    // ConcreteFactory2
    class ConcreteFactory2 : AbstractFactory
    {
        public override AbstractProductA CreateProductA()
        {
            return new ProductA2();
        }
        public override AbstractProductB CreateProductB()
        {
            return new ProductB2();
        }
    }

    // AbstractProductA
    abstract class AbstractProductA
    {
    }

    // AbstractProductB
    abstract class AbstractProductB
    {
        public abstract void Interact(AbstractProductA a);
    }

    // ProductA1
    class ProductA1 : AbstractProductA
    {
```

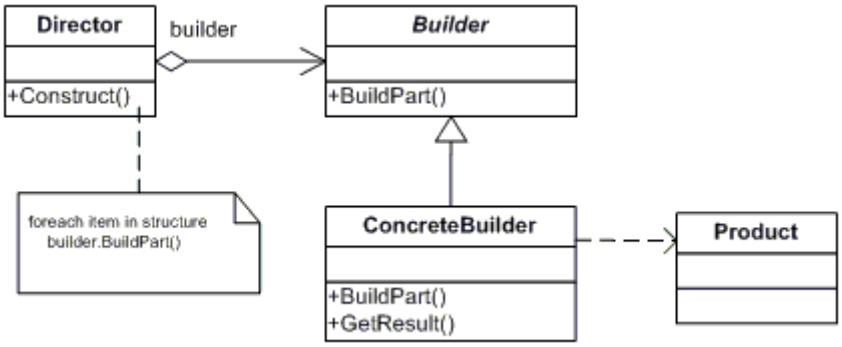
```
    }

    // ProductB1
    class ProductB1 : AbstractProductB
    {
        public override void Interact(AbstractProductA a)
        {
            Console.WriteLine(this.GetType().Name +
              " interacts with " + a.GetType().Name);
        }
    }

    // ProductA2
    class ProductA2 : AbstractProductA
    {
    }

    // ProductB2
    class ProductB2 : AbstractProductB
    {
        public override void Interact(AbstractProductA a)
        {
            Console.WriteLine(this.GetType().Name +
              " interacts with " + a.GetType().Name);
        }
    }

    // Client
    class Client
    {
        private AbstractProductA _abstractProductA;
        private AbstractProductB _abstractProductB;

        public Client(AbstractFactory factory)
        {
            _abstractProductB = factory.CreateProductB();
            _abstractProductA = factory.CreateProductA();
        }

        public void Run()
        {
            _abstractProductB.Interact(_abstractProductA);
        }
    }

    class Program
    {
        public static void Main()
        {
            AbstractFactory factory1 = new ConcreteFactory1();
            Client client1 = new Client(factory1);
            client1.Run();

            AbstractFactory factory2 = new ConcreteFactory2();
            Client client2 = new Client(factory2);
            client2.Run();

            Console.ReadKey();
        }
    }
}
```

Builder Design Pattern

Builder Design Pattern Creational Patterns gurubu içerisinde yer alır. Üretimi bir çok parçadan oluşmuş, çok parçalı karmaşık nesnelerin parçalarının adım adım bir araya getirilerek oluşturulmasını sağlayan bir tasarım desenidir.

Nesne üretecek istemci, üretilecek nesnenin tipini belirleyerek nesneyi elde eder.

Nesne üretimi ile ilgilenmez.



Director
Builder arayüzünü uygulayarak nesne örneklemesi yapan yapı.

Builder
Nesne oluşturulma işlemi şablonu için abstract class veya interface.

ConcreteBuilder:
Nesnenin temel özelliklerini oluşturacak yapı. Ayrıca nesnenin dışa açılmasını sağlayan yapılar da içerir.

Product
Nesne

Örneğimizde Bir bilgisayar üretimi aşamalarını konumuz ile ilişkilendirmeye çalışacağız.

```
namespace ConsoleApplication1
{
    //Builder
    public abstract class ComputerBuilder
    {
        protected Product computer;

        public Product Computer
        {
            get { return computer; }
        }

        public abstract void UretimHattinaAl();
        public abstract void CihazMontaj();
        public abstract void IsletimSistemiKurulum();
    }

    //Product
    public class Product
    {
```

```
        public string Marka { get; set; }
        public string Model { get; set; }
        public string Anakart { get; set; }
        public string Cpu { get; set; }
        public string Ram { get; set; }
        public string HDD { get; set; }
        public string Kasa { get; set; }
        public string Klavye { get; set; }
        public string Mouse { get; set; }
        public string Monitor { get; set; }
        public string OS { get; set; }

        public override string ToString()
        {
            return Marka + " Marka " + Model + " Model bilgisayar
üretildi.";
        }
    }

    //Director
    public class Director
    {
        public void BilgisayarUret(ComputerBuilder computerBuilder)
        {
            computerBuilder.UretimHattinaAl();
            computerBuilder.CihazMontaj();
            computerBuilder.IsletimSistemiKurulum();
        }
    }

    //ConcreteBuilder A
    public class AConcreteBuilder : ComputerBuilder
    {
        public AConcreteBuilder()
        {
            computer = new Product();
        }

        public override void UretimHattinaAl()
        {
            computer.Marka = "A";
            computer.Model = "Home 1000";
        }

        public override void CihazMontaj()
        {
            computer.Anakart = "Gigabyte";
            computer.Cpu = "Intel";
            computer.HDD = "Samsung 300GB";
            computer.Kasa = "AOpen";
            computer.Klavye = "OEM";
            computer.Mouse = "OEM";
            computer.Monitor = "Philips";
            computer.Ram = "Kingston 2GB";
        }

        public override void IsletimSistemiKurulum()
        {
            computer.OS = "Windows 7";
        }
    }
```

```
    //ConcreteBuilder B
    public class BConcreteBuilder : ComputerBuilder
    {
        public BConcreteBuilder()
        {
            computer = new Product();
        }

        public override void UretimHattinaAl()
        {
            computer.Marka = "B";
            computer.Model = "Server 200";
        }

        public override void CihazMontaj()
        {
            computer.Anakart = "Asus";
            computer.Cpu = "Amd";
            computer.HDD = "Maxtor 200GB";
            computer.Kasa = "AOpen";
            computer.Klavye = "OEM";
            computer.Mouse = "OEM";
            computer.Monitor = "Samsung";
            computer.Ram = "Kingston 4GB";
        }

        public override void IsletimSistemiKurulum()
        {
            computer.OS = "Linux";
        }
    }

    //Client
    class Program
    {
        static void Main(string[] args)
        {
            ComputerBuilder computerBuilder;
            Director director = new Director();

            computerBuilder = new AConcreteBuilder();
            director.BilgisayarUret(computerBuilder);
            Console.WriteLine(computerBuilder.Computer.ToString());

            computerBuilder = new BConcreteBuilder();
            director.BilgisayarUret(computerBuilder);
            Console.WriteLine(computerBuilder.Computer.ToString());

            Console.Read();
        }
    }
}
```

Ekran çıktısı

```
A Marka Home 1000 Model bilgisayar üretildi.
B Marka Server 200 Model bilgisayar üretildi.
```

Client uygulama AConcreteBuilder ve BConcreteBuilder tipleri ile üretilecek bilgisayarın modelini seçmiş, istediği tipte bilgisayar ürettirmiş ve üretim aşaması ile ilgilenmemiş soyutlanmıştır. Builder pattern ile Abstract Factory pattern karıştırılmaktadır. Aralarındaki fark Builder pattern üretimi karmaşık çok parçalı nesnelerle ilgilenmektedir. Abstract Factory üretimin karmaşık yada basit olması ile ilgilenmez.

Factory Method Pattern

Creational Pattern Gurubu içerisinde yer alır.

İstemcilerin talep ettikleri nesneleri oluşturmak için bir fabrika metodundan faydalınılır.

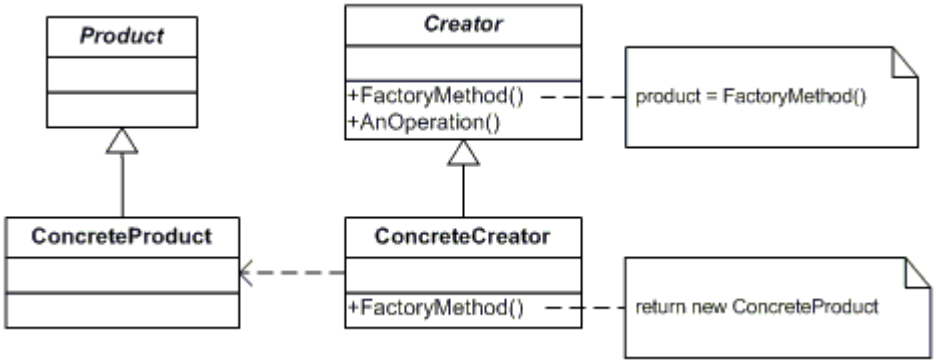
Nesnenin nasıl yaratılacağı kalıtım yoluyla alt sınıflara bırakılır.

İstemci kullanacağı asıl ürün nesnesinin nasıl üretileceği konusunda bir bilgiye sahip değildir.

Nesne nasıl üretilir ne aşamalardan geçer şimdiki durum ve gelecekteki durum bunlarla ilgilenmez sadece nesne talep eder.

Nesnenin üretim aşamasını client'dan saklanmış olur. Nesne üretim aşamasında meydana gelen bir değişiklik istemci uygulamadan izole edilir ve istemci uygulama bu değişimden etkilenmez.

Bu metodlar nesne üreten metotlar olduğundan metod çağrıldığında bir sınıftan türetilmiş bir nesne geri döndürmesi gerekir.



1.Kullanım

Client creator tipinden bir nesnenin factory metodu ile Concrete Product sınıfından bir nesne üretir.

Product

İstemcinin talep ettiği nesnenin tipi. Üretilecek ürünlerin şablonu. Interface yada abstract class

Concrete Product

Asıl nesne. Product arayüzünü uygular.

Creator

Nesne üretimi ile ilgili işlemi yerine getiren, Client uygulamanın taleplerini karşılayan ve uygun ürünü sunan yapı. Nesne üreten Factory Metot bu yapı içerisinde yer alıyor.

Client

İstemci uygulama.

2.Kullanım

Client creator tipinden nesne üreterek concrete creator tipleri içerisindeki factory metotları çağırarak nesne elde eder.

Diğer yapılar aynı kalıp şu iki yapı değişiklik gösterir.

Creator
ConcreteCreator tiplerinin uygulayacağı şablon. Abstract Class yada Interface.

Concrete Creator
Nesne üretimi ile ilgili işlemi yerine getiren, Client uygulamanın taleplerini karşılayan ve uygun ürünü sunan yapı. Factory Method bu yapı içerisinde yer alır.

```
namespace ConsoleApplication1
{
    public enum Ulkeler
    { Turkiye, Ispanya, Amerika }

    // Product
    public abstract class Product
    {
        public abstract void Konus();
    }

    // ConcreteProduct
    public class ConcreteProduct1 : Product
    {
        public override void Konus()
        {
            Console.WriteLine("Türkiye için üretilen nesne Merhaba Dünya
diyor...");
        }
    }

    // ConcreteProduct
    public class ConcreteProduct2 : Product
    {
        public override void Konus()
        {
            Console.WriteLine("İspanya için üretilen nesne Hola Mundo
diyor...");
        }
    }

    // ConcreteProduct
    public class ConcreteProduct3 : Product
    {
        public override void Konus()
        {
            Console.WriteLine("Amerika için üretilen nesne Hello World
diyor...");
        }
    }

    public class Creator
    {
        public Product FactoryMethod(Ulkeler ulke)
        {
            Product product = null;

            switch (ulke)
            {
                case Ulkeler.Turkiye:
                    product = new ConcreteProduct1();
                    break;
```

```
                case Ulkeler.Ispanya:
                    product = new ConcreteProduct2();
                    break;
                case Ulkeler.Amerika:
                    product = new ConcreteProduct3();
                    break;
                default:
                    break;
            }

            return product;
        }
    }

    // Client
    class Program
    {
        static void Main()
        {
            Creator creator = new Creator();

            Product product1 = creator.FactoryMethod(Ulkeler.Turkiye);
            Product product2 = creator.FactoryMethod(Ulkeler.Ispanya);
            Product product3 = creator.FactoryMethod(Ulkeler.Amerika);

            product1.Konus();
            product2.Konus();
            product3.Konus();

            Console.ReadKey();
        }
    }
}
```

2. Kullanım Örneği

```
namespace ConsoleApplication1
{
    public enum Ulkeler
    { Turkiye, Ispanya, Amerika}

    // Product
    public abstract class Product
    {
        public abstract void Konus();
    }

    // ConcreteProduct
    public class ConcreteProduct1 : Product
    {
        public override void Konus()
        {
            Console.WriteLine("Türkiye için üretilen nesne Merhaba Dünya
diyor...");
        }
    }

    // ConcreteProduct
    public class ConcreteProduct2 : Product
    {
```

```
        public override void Konus()
        {
            Console.WriteLine("İspanya için üretilen nesne Hola Mundo
diyor...");
        }
    }

    // ConcreteProduct
    public class ConcreteProduct3 : Product
    {
        public override void Konus()
        {
            Console.WriteLine("Amerika için üretilen nesne Hello World
diyor...");
        }
    }

    public abstract class Creator
    {
        public abstract Product FactoryMethod();
    }

    public class ConcreteCreator1 : Creator
    {
        public override Product FactoryMethod()
        {
            return new ConcreteProduct1();
        }
    }

    public class ConcreteCreator2 : Creator
    {
        public override Product FactoryMethod()
        {
            return new ConcreteProduct2();
        }
    }

    public class ConcreteCreator3 : Creator
    {
        public override Product FactoryMethod()
        {
            return new ConcreteProduct3();
        }
    }

    // Client
    class Program
    {
        static void Main()
        {
            Creator[] creators = {
               new ConcreteCreator1(),
               new ConcreteCreator2(),
               new ConcreteCreator3()};

            foreach (Creator creator in creators)
            {
                Product product = creator.FactoryMethod();
                product.Konus();
            }
```

```
            Console.ReadKey();
        }
    }
}
```

Ekran Çıktısı (Her iki kullanım örneği için)

Türkiye için üretilen nesne Merhaba Dünya diyor...
İspanya için üretilen nesne Hola Mundo diyor...
Amerika için üretilen nesne Hello World diyor...

Prototype Design Pattern

Hatırlayacağınız gibi Creational Patterns bir yada daha fazla nesnenin oluşturulması ve yönetilmesi ile ilgili patternlerden oluşan gurubu temsil ediyordu.

Prototype Pattern yeni bir nesne oluşturmak yerine varolan oluşturulmuş bir nesneden klonlama yöntemi ile yeni bir nesne örneği elde etmek için kullanılıyor.

Kullanım nedenleri

Nesnenin diğer oluşturulacak nesnelere model olması amacıyla prototipinin oluşturulması ve bu prototipten oluşturulacak nesnelere örnek teşkil etmesi, hazır yapı sunması.

Üretim için kalıp oluşturması, diğer üretimler için bu kalıbın kullanılması.

Bir nesnenin oluşturulma maliyetinin yüksek olması nedeniyle defalarca oluşturulacak aynı nesneler için bir model nesne oluşturmak ve diğer oluşturulacak nesneleri yeniden yaratmadan model nesneden klonlamak bu yöntemle nesne üretim maliyetini düşürmek.

Örneğin yeni bir nesne oluşturulmak isteniyor. Çeşitli hesaplama ve değerlendirmeler sonucu nesnenin içerisi dolduruluyor.

Yeni bir nesne oluşturma zaman maliyeti 1 sn sürüyor ve bu nesne sık tekrarlanan kullanıma sahip.

20 kullanımlık bir talep geldi. Üretim için harcanan zaman 20 x 1 = 20 sn

bunu klonlamak nesne başı 0.5ms x 20 = 100 mili sn yani saniyenin 10 da biri toplamda harcanacak süre. (Tahmini rakam)

Prototype Pattern ile oluşturulan nesneler klonlandığı nesnenin bire bir özelliklerini taşımakla beraber genişletebilir bir yapıya da sahiptirler.

Ne tür örnekler verebiliriz ?

Çizim programında belirli bir şeklinin çizim alanına defalarca eklenmesi.

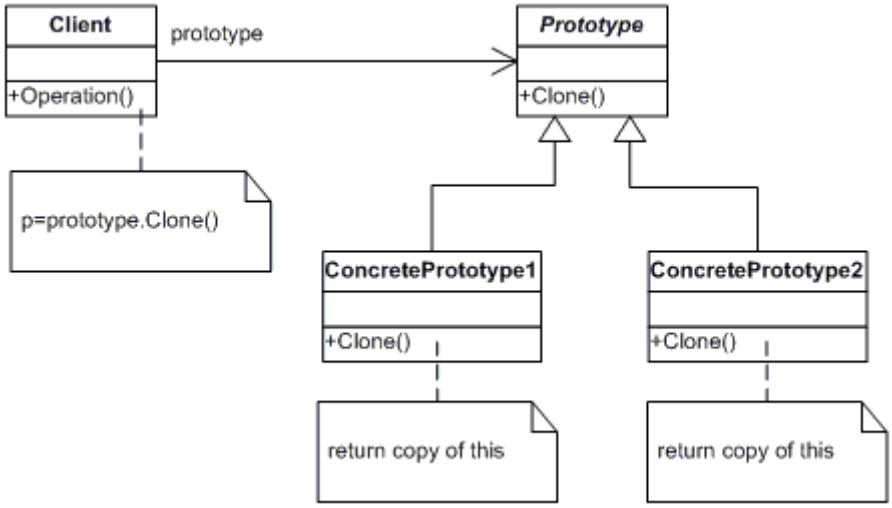
Oyunlarda bir silahtan defalarca mermi atılması ve her bir merminin bir nesne ile temsili gibi..

Biz örneğimizde Bilgisayar ve yazıcı arasında gerçekleşen bir senaryoyu konumuza uyarlamaya çalışacağız.

Yazdırma işleminde yazıcıya ait bilgilerin sistemden alındığı bir nesnemiz olacak.

Her yazdırma işleminde bu nesne gibi bir nesneye ihtiyaç duyulacak.

Sık tekrarlanan bir kullanım ihtiyacı olduğundan ya her seferinde yeni bir nesne oluşturulacak yada zaten oluşturulmuş bir nesne kendinden sonra kullanılacak nesnelere model oluşturacak.



Prototype
Bir abstract class veya interface olacak. İçerisinde Clone isimli metot bulunacak.

Concrete Prototype
İşleme konu olan nesnemiz. (Prototype yapısını uyguladığı için Clone metodunu uygulamak, ezmek zorunda)

(Prototype Pattern uygulanacak sınıf sayısı kadar ConcertePrototype yapısı oluşturulabilir. Biz bir sınıfa uygulayacağız.)

Client
İstemci uygulamamız

(Concrete Prototype) Printer isimli sınıfımız bir printer'ın tanımlandığı, cihaz özelliklerinin sistemden alındığı ve nesnenin doldurulduğu bir yapı. Bu sınıf (Prototype) PrinterPrototype abstract sınıfını uyguluyor. Böylece Clone metodunu kullanması zorunlu hale geliyor. (Client) üzerinde yeni bir printer nesnesi oluşturuluyor. PrintDocument fonksiyonuna yeni bir nesne oluşturmadan varolanı klonlayarak gönderiyor. Böylece Printer class içerisindeki constructor çalışmıyor ve nesne için new işlemi gerçekleşmiyor. Printer sınıfı constructor içerisine işlemin oluşum süresi için 1 sn'lik gecikme ekledim. Eğer her seferinde yeni nesne oluşturulursa yazdırılacak sayfa sayısı kadar 1 sn harcanacak. Varolan klonlanırsa yeni nesne üretilmediğinden Constructor yapısı içerisine hiç girilmeyecek ve bu gecikme hiç olmayacak.

```
namespace ConsoleApplication1
{
    //Prototype
    abstract class PrinterPrototype
    {
        public abstract PrinterPrototype Clone();
    }

    //Concrete Prototype
    class Printer : PrinterPrototype
    {
        public byte Address { get; set; }
        public string Name { get; set; }
        public bool ColorCartridge { get; set; }
```

```
        public bool BlackCartridge { get; set; }
        public int ResolutionDpi { get; set; }
        public string PaperSize { get; set; }

        public Printer(byte address, string name, bool colorCartridge,
            bool blackCartridge, int resolution, string paperSize)
        {
            Address = address;
            Name = name;
            ColorCartridge = colorCartridge;
            BlackCartridge = blackCartridge;
            ResolutionDpi = resolution;
            PaperSize = paperSize;
            System.Threading.Thread.Sleep(1000);
        }

        public override PrinterPrototype Clone()
        {
            // Klonlama işlemi. Gelen nesnenin bir kopyası alınıp sonuç
olarak döndürülüyor.
            return (PrinterPrototype)this.MemberwiseClone();
        }
    }

    //Client
    class Program
    {
        static void Main()
        {
            // Prototype model nesne oluşturuluyor.
            Printer printer = new Printer(0x0F, "Hp", true, true, 600,
"A4");

            for (int i = 0; i < 20; i++)
            {
                PrintDocument((Printer)printer.Clone(), i.ToString());
                Console.WriteLine(Convert.ToString(i + 1) + " nolu sayfa
basıldı");
            }

            Console.WriteLine("Yazdırma tamamlandı...");
            Console.ReadLine();
        }

        static void PrintDocument(Printer printer, string content)
        {
            // Print işlemi yapan metod.
        }
    }
}
```

Singleton Design Pattern

Creatinal Patterns gurubu içerisinde yer alır.

Uygulamanın çalışma süresi boyunca bir nesnenin sadece tek bir örneğinin oluşturulmasının garanti altına alınması istenildiğini durumlarda singleton pattern kullanılır.

Konuyu bir donanım aygıtı ile örneklendirelim. Bir DVD-RW cihazımız olsun.

Bu cihaz dvd yazma ve okumayı aynı anda gerçekleştiremez. DVD nesnesinin birden fazla örneğini oluşturmak ve kullanmak karmaşaya yol açacaktır. Bir nesneden yazma komutunu çalıştırmak diğer nesneden okuma işlemi yapmaya çalışmak hatalara neden olacaktır.

Burada ortaya çıkan ihtiyaç/problem için başvuru kaynağımız singleton pattern olacaktır.

Amacımız sistemimize bağlı olan bir DVD cihazını yalnızca bir nesne ile temsil etmek.

```
namespace ConsoleApplication1
{
    public class DVDDevice
    {
        private static DVDDevice dvdDevice = new DVDDevice();

        // 1.Yol DvdDevice nesnesine Property üzerinden ulaşmak
        public static DVDDevice Device
        {
            get
            {
                return dvdDevice;
            }
        }

        //2.Yol DvdDevice nesnesine Method üzerinden ulaşmak
        public static DVDDevice GetDevice()
        {
            return dvdDevice;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //1.Yol
            DVDDevice device = DVDDevice.Device;

            //2.Yol
            DVDDevice device = DVDDevice.GetDevice();

            Console.Read();
        }
    }
}
```

DVDDevice sınıfı belleğe yüklenmesi ile dvdDevice static nesnesi yaratılacak ve o nesneyi kullanmak isteyen yapılar, yukarıdaki yaratılan nesnenin private erişim belirleyicisi ile tanımlanıp dışarıya kapatılması nedeniyle, örneklerde olduğu gibi bir property yada metod üzerinden nesneye ulaşacaklardır. Main metodunda görüldüğü gibi new anahtar sözcüğü kullanılmamıştır. Çünkü nesne burada yaratılmamıştır.

Nesnenin yaratılması DVDDevice sınıfının belleğe yüklendiği sırada değil bizim istediğimiz herhangi bir zamanda olması istendiği durumda yukarıdaki yapıyı şu şekilde değiştireceğiz.

```
namespace ConsoleApplication1
{
    public class DVDDevice
    {
        private static DVDDevice dvdDevice;

        // 1.Yol DvdDevice nesnesine Property üzerinden ulaşmak
        public static DVDDevice Device
        {
            get
            {
                if (dvdDevice == null)
                    dvdDevice = new DVDDevice();

                return dvdDevice;
            }
        }

        //2.Yol DvdDevice nesnesine Method üzerinden ula
        public static DVDDevice GetDevice()
        {
            if (dvdDevice == null)
                dvdDevice = new DVDDevice();

            return dvdDevice;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //1.Yol
            DVDDevice device = DVDDevice.Device;

            //2.Yol
            DVDDevice device = DVDDevice.GetDevice();

            Console.Read();
        }
    }
}
```

İlk örneğimizde

```
private static DVDDevice dvdDevice = new DVDDevice();
```

dvdDevice nesnemiz new operatörü ile DVDDevice sınıfı belleğe yüklendiğinde otomatik olarak oluşturuluyordu. İkinci örneğimizde

```
private static DVDDevice dvdDevice
```

Nesnemiz tanımlandı fakat yaratılmadı.

Nesneye erişim için kullandığımız property ve metod içinde nesnenin durumu sorgulanıyor. Eğer nesne henüz yaratılmadı ise yeni bir nesne yaratıp bu nesneyi döndürüyor. Nesne daha önce yaratıldı ise yaratılan nesneyi döndürüyor.

```
if (dvdDevice == null)
     dvdDevice = new DVDDevice();
return dvdDevice;
```

Multithread bir uygulamada birden fazla thread, singleton olarak tanımlamak istediğimiz nesneyi kullanma ihtiyacı duyarsa yapılacak şey işlem bitene kadar diğer thread'ler için bu bloğu kilitlemek.
Kilit işlemi için global bir nesne yaratıyoruz.

```
private static readonly object LockObject = new object();
```

Ardından kod bloğmuzu bu şekilde değiştiriyoruz.

```
System.Threading.Monitor.Enter(LockObject);
   if (dvdDevice == null)
        dvdDevice = new DVDDevice();
   return dvdDevice
System.Threading.Monitor.Exit(LockObject);
```

Adapter Design Pattern

Structural Patterns Grubu içerisinde yer alır.

Varolan mevcut sisteme yabancı bir yapının adapte edilmesi istenildiği durumlarda kullanılır.

Birbiri ile uyum sağlamayan yapıların araya bir bağlantı adaptörü eklenmesi ile kullanılabilir hale getirilmesi sağlanır.

Örneğin

usb bir klavyenin ucuna takılan ps2 adaptörü ile ps2 girişine bağlanabilmesi.

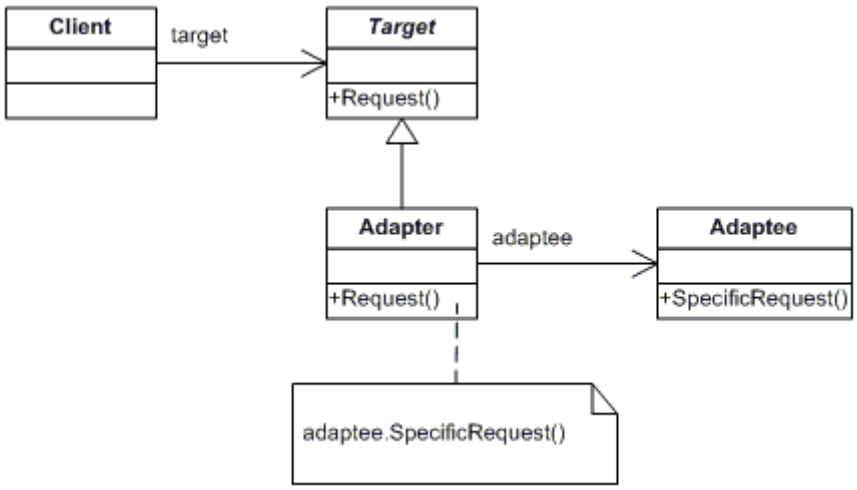
büyük uçlu kulaklık girişlerinin çevirici adaptör ile normal kulaklık olarak kullanılması.

genişliği 1 birim boru ile 3 birim olan iki borunun ortaya eklenen bir ek boru ile birleştirilmesi.

gibi gündelik hayattan örnekler verilebilir.

Adapter pattern ayrıca Wrapper Pattern olarak da isimlendirilir.

Com,ActiveX nesneleri gibi unmanaged kod yapıları ile managed kod yapıları arasında wrapper olarak kullanılır. Bu wrapper adaptör aracılığı ile biz bu nesnelerin özellik ve metotlarına erişebilir ve kullanabiliriz.



Target
Varolan mevcut yapı.

Adapter
Adaptör

Adapte
Adapte edilecek tip nesne (yabancı yapı)

Client
İstemci uygulama

```
namespace ConsoleApplication1
{
    // Target
    class Target
    {
        public virtual void Request()
        {
            Console.WriteLine("Called Target Request()");
        }
    }

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

        public override void Request()
        {
            _adaptee.SpecificRequest();
        }
    }

    //Adaptee
    class Adaptee
    {
        public void SpecificRequest()
        {
            Console.WriteLine("Called SpecificRequest()");
        }
    }

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

Adapter doğrudan adaptee dan türetilerek yada adapter adapte tipinden bir nesne örneğini sarmalayıp tutararak kullanılabilir.

Adaptee farklı platformlarda çalışan bir yapı olabilir, farklı library içinde olabilir, com/activex olabilir.

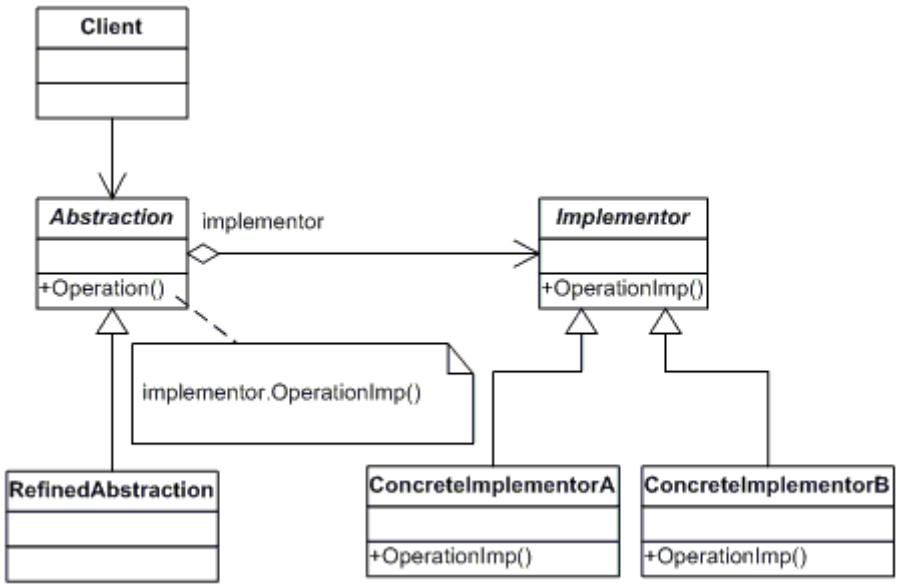
Bridge Design Pattern

Structural Patterns gurubunda yer alır.

Bir sınıfın arayüzünü implementasyonundan ayırmak için kullanılır.

Böylece her iki yapıda birbirinden bağımsız olarak değişebilir.

Bu yapı kendi içerisinde başka bir yapının daha oluşturulmasına izin veren ve bu iç yapının değişikliğe uğramasıyla client'ların hiç bir şekilde etkilenmemesini garanti altına alan bir yapı bizlere sunmaktadır.Abstraction , RefinedAbstraction,Implementor ve ConcreteImplementor yapı taşlarından oluşur.




```
namespace ConsoleApplication1
{
    // Client
    class MainApp
    {
        static void Main()
        {
            Abstraction ab = new RefinedAbstraction();

            ab.Implementor = new ConcreteImplementorA();
            ab.Operation();

            ab.Implementor = new ConcreteImplementorB();
            ab.Operation();

            // Wait for user
            Console.ReadKey();
        }
    }

    // Abstraction
    class Abstraction
    {
        protected Implementor implementor;
```

```
        public Implementor Implementor
        {
            set { implementor = value; }
        }

        public virtual void Operation()
        {
            implementor.Operation();
        }
    }

    abstract class Implementor
    {
        public abstract void Operation();
    }

    // RefinedAbstraction
    class RefinedAbstraction : Abstraction
    {
        public override void Operation()
        {
            implementor.Operation();
        }
    }

    // ConcreteImplementorA
    class ConcreteImplementorA : Implementor
    {
        public override void Operation()
        {
            Console.WriteLine("ConcreteImplementorA Operation");
        }
    }

    // ConcreteImplementorB
    class ConcreteImplementorB : Implementor
    {
        public override void Operation()
        {
            Console.WriteLine("ConcreteImplementorB Operation");
        }
    }
}
```

Ekran Çıktısı

ConcreteImplementorA Operation
ConcreteImplementorB Operation

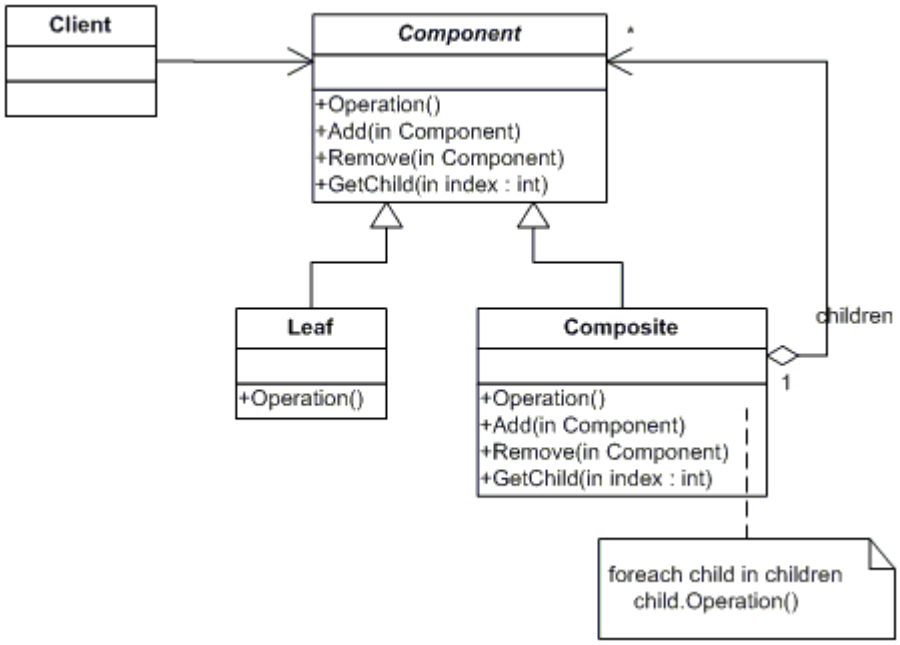
Composite Design Pattern

Composite Design Pattern Structural Patterns gurubunda yer almaktadır.

Bir yapıyı oluşturan nesneler topluluğunda yapıyı oluşturan parçaları ağaç (tree) hiyeraşik yapısına göre oluşturan ve aralarındaki ilişkiler için düzen sağlayan bir tasarım desenidir.

Hiyeraşik Yapı : Bir toplulukta veya bir kuruluşta yer alan kişileri alt-üst ilişkileri, görev ve yetkilerine göre sınıflandıran sistem olarak tanımlanır.

Xml, Html Dom, Tree View kontrolü, Active Directory gibi yapılar kullanım yerleri açısından örnek olarak gösterilebilir.



Client
İstemci uygulama

Component
Composite ve Leaf için tanımlanacak ortak şablon abstract class yada interface olacak.

Leaf
Yapı içerisindeki nesnelerin davranışlarını tanımlayan Leaf nesnelerini sunan yapı.

Composite
İçerisinde Composite ve Leaf nesnelerini tutan yapı.

Biz örneğimizde Windows Form uygulamaları yapısını konumuz ile ilişkilendirmeye çalışacağız.

Bir Composite Root tanımlanacak bu Root altına Leaf tipinden elemanlar eklenecek. Daha sonra Composit tipli nesneler üretilecek. Bunların içerisine Leaf tipli elemanlar eklenecek ve daha sonra Composit tipli oluşturulan container nesneler Root Composite içerisine eklenecek.

Aşağıdaki şekil bu tanımı örneklendiriyor.

+Form (Root) (Composite)
....TextBox (Leaf)
....TextBox (Leaf)
....Button (Leaf)
....Panel (Composite)
.....TextBox (Leaf)
.....TextBox (Leaf)
....Panel (Composite)
.....TextBox (Leaf)
.....TextBox (Leaf)
....Button (Leaf)

```
namespace ConsoleApplication1
{
    public enum KontrolTip
    {
        Form,
        Panel,
        TextBox,
        Button,
    }

    //Component
    abstract class Kontrol
    {
        public string _id;
        public string _text;
        public KontrolTip _tip;

        public Kontrol(string id, string text, KontrolTip tip)
        {
            _id = id;
            _text = text;
            _tip = tip;
        }

        public abstract void Ekle(Kontrol uye);
        public abstract void Sil(Kontrol uye);
        public abstract void Goster(int satirBosluk);
    }

    //Leaf
    class LeafKontrol : Kontrol
    {
        public LeafKontrol(string id, string text, KontrolTip tip)
            : base(id, text, tip)
        {
            //Leaf için kullanılmaz
        }
        public override void Ekle(Kontrol kontrol)
        {
            //Leaf için kullanılmaz
        }

        public override void Sil(Kontrol kontrol)
        {
            //Leaf için kullanılmaz
        }
```

```
        public override void Goster(int satirBosluk)
        {
            Console.WriteLine(new String('.', satirBosluk) + " " + _id + " " + _text);
        }
    }

    //Composite
    class CompositeKontrol : Kontrol
    {
        //Composite tip içinde tutulacak componentler için generic list
        private List<Kontrol> _kontroller = new List<Kontrol>();

        public CompositeKontrol(string id, string text, KontrolTip tip)
            : base(id, text, tip)
        {
        }

        public override void Ekle(Kontrol uye)
        {
            _kontroller.Add(uye);
        }

        public override void Sil(Kontrol uye)
        {
            _kontroller.Remove(uye);
        }

        public override void Goster(int satirBosluk)
        {
            Console.WriteLine(new String('.', satirBosluk) + "+ " + _id + " " +
_text);

            foreach (Kontrol kontrol in _kontroller)
            {
                kontrol.Goster(satirBosluk + 1);
            }
        }
    }

    //Client
    class Program
    {
        static void Main(string[] args)
        {
            CompositeKontrol cKontrolRoot = new CompositeKontrol("Form1",
"WinApplication 1", KontrolTip.Form);
            cKontrolRoot.Ekle(new LeafKontrol("TextBox1", "Adınızı Giriniz...",
KontrolTip.TextBox));
            cKontrolRoot.Ekle(new LeafKontrol("TextBox2", "Soyadınız Giriniz...",
KontrolTip.TextBox));
            cKontrolRoot.Ekle(new LeafKontrol("Button1", "Kontrol Et",
KontrolTip.Button));

            CompositeKontrol cKontrol1 = new CompositeKontrol("Panel1", "Adres
Bilgileri", KontrolTip.Panel);
            cKontrol1.Ekle(new LeafKontrol("TextBox4", "Adres Satır 1",
KontrolTip.TextBox));
            cKontrol1.Ekle(new LeafKontrol("TextBox5", "Adres Satır 2",
KontrolTip.TextBox));
            cKontrolRoot.Ekle(cKontrol1);

            CompositeKontrol cKontrol2 = new CompositeKontrol("Panel2", "Telefon
Bilgileri", KontrolTip.Panel);
```

```
            cKontrol2.Ekle(new LeafKontrol("TextBox6", "Telefon 1",
KontrolTip.TextBox));
            cKontrol2.Ekle(new LeafKontrol("TextBox7", "Telefon 2",
KontrolTip.TextBox));
            cKontrolRoot.Ekle(cKontrol2);

            cKontrolRoot.Ekle(new LeafKontrol("Button2", "Gönder",
KontrolTip.Button));

            cKontrolRoot.Goster(1);

            Console.Read();
        }
    }
}
```

Ekran Çıktısı

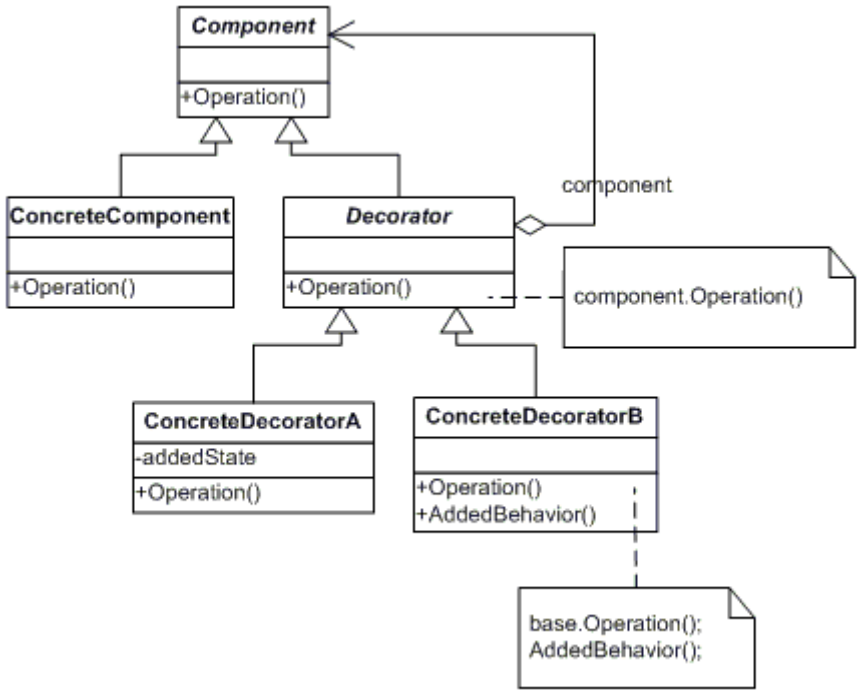
.+ Form1 WinApplication 1
.. TextBox1 Adınızı Giriniz...
.. TextBox2 Soyadınız Giriniz...
.. Button1 Kontrol Et
..+ Panel1 Adres Bilgileri
... TextBox4 Adres Satır 1
... TextBox5 Adres Satır 2
..+ Panel2 Telefon Bilgileri
... TextBox6 Telefon 1
... TextBox7 Telefon 2
.. Button2 Gönder

Decorator Design Pattern

Structural Patterns gurubunda yer alır. Hatırlaycağınız gibi Structural Patterns nesnelerin birbirleri ile olan ilişkileri temel alınarak tasarlanan desenlerdi.

Çalışma zamanında (runtime) bir nesnenin yapısını bozmadan değiştirilmesi yeni yapıların (sorumluluklar) eklenmesi

yada varolan yapıların çıkartılması işlemleri için kullanılır.



Component

Bileşenin şablonu. Yeni yapıların, sorumlulukların ekleneceği nesnemizin arayüzü.

Interface yada abstract class olarak tanımlanacak.

ConcreteComponent

Bileşenimiz. Compenent yapısını uygulayan yapıların dinamik olarak eklendiği asıl bileşen sınıfı.

Decorator

Component yapısını uygulayan ayrıca Component tipinden bir nesne referansı içeren sınıf.

ConcreteDecorator

Bileşene yeni yapı, sorumlulukları kazandıran bileşeni genişleten sınıftır.

Biz örneğimizde bir dosya yöneticisini konumuz ile ilişkilendirmeye çalışacağız.

Bir dosya yöneticimiz olacak, dosya yöneticimizin ad, kaynak, hedef özellikleri ve sadece kopyalama işlemi yapan bir metodu olacak. Biz decorator tasarım kalıbı aracılığı ile dosya yöneticimize Aç, Sil, KısaYol Oluştur gibi ek özellikler sorumluluklar kazandıracağız.

```
namespace ConsoleApplication1
{
    // Component
    abstract class ComponentBase
    {
        public string Ad;
        public string Kaynak;
        public string Hedef;
        public abstract void Kopyala();
    }

    // ConcreteComponent
    class ConcreteComponent : ComponentBase
    {
        public ConcreteComponent(string kaynak, string hedef, string ad)
        {
            base.Ad = ad;
            base.Kaynak = kaynak;
            base.Hedef = hedef;
        }

        public override void Kopyala()
        {
            Console.WriteLine("{0} dosya yöneticisi {1} kaynaktan {2}
hedefe kopyalama işlemi yaptı.", Ad, Kaynak, Hedef);
        }
    }

    // Decorator
    abstract class DecoratorBase : ComponentBase
    {
        protected ComponentBase DosyaYonetici;
        public DecoratorBase(ComponentBase dosyaYonetici)
        {
            DosyaYonetici = dosyaYonetici;
        }
        public override void Kopyala()
        {
            if (DosyaYonetici != null)
                DosyaYonetici.Kopyala();
        }
    }

    // ConcreteDecorator
    class ConcreteDecorator : DecoratorBase
    {
        public ConcreteDecorator(ComponentBase dosyaYonetici)
            : base(dosyaYonetici)
        {
        }

        public void Ac()
        {
            Console.WriteLine("{0} dosya yoneticisi {1} dosyasını açtı.",
base.DosyaYonetici.Ad, DosyaYonetici.Kaynak);
        }

        public void Sil()
        {
            Console.WriteLine("{0} dosya yoneticisi {1} dosyasını sildi.",
```

```
base.DosyaYonetici.Ad, base.DosyaYonetici.Kaynak);
        }

        public void KisaYolOlustur()
        {
            Console.WriteLine("{0} dosya yoneticisi {1} dosyası için {2}
konumunda kısayol oluşturdu.", base.DosyaYonetici.Ad,
base.DosyaYonetici.Kaynak, base.DosyaYonetici.Hedef);
        }

        public override void Kopyala()
        {
            base.Kopyala();
        }
    }

    // Client
    class Program
    {
        static void Main()
        {
            ConcreteComponent dosyaYonetici = new
ConcreteComponent(@"c:\a.txt", @"d:\a.txt", "D1");
            dosyaYonetici.Kopyala();
            ConcreteDecorator dekoratorDosyaYonetici = new
ConcreteDecorator(dosyaYonetici);
            dekoratorDosyaYonetici.Sil();
            dekoratorDosyaYonetici.Ac();
            dekoratorDosyaYonetici.KisaYolOlustur();
            dekoratorDosyaYonetici.Kopyala();
            Console.Read();
        }
    }
}
```

Ekran Çıktısı

D1 dosya yöneticisi c:\a.txt kaynaktan d:\a.txt hedefe kopyalama işlemi yaptı.
D1 dosya yoneticisi c:\a.txt dosyasını sildi.
D1 dosya yoneticisi c:\a.txt dosyasını açtı.
D1 dosya yoneticisi c:\a.txt dosyası için d:\a.txt konumunda kısayol oluşturdu.
D1 dosya yöneticisi c:\a.txt kaynaktan d:\a.txt hedefe kopyalama işlemi yaptı.

Facade Design Pattern

Structural Desing Pattern Gurubunda yer alır.

Türkçe karşılığı Önyüz, Cephe anlamına gelir.

Bir sistemin parçalarını oluşturan sınıfları istemci uygulamadan soyutlamak ve kullanım kolaylığı sağlamak için kullanılabilecek bir tasarım desenidir.

Karmaşık bir yapıyı kolay kullanılabilir anlaşılabilir hale getirmek için kullanılır.

İçinde pek çok yapıyı fonksiyonu barındıran bir kod kümesi (library) içinden kullanacaklarını seçmek ve bunları bir önyüz(facade) ile kullanmak.

Farklı görevleri yerine getiren sınfılarımız olsun.

İstemci

Facade

Class1: TextLog işlemleri

Class2: EventLog işlemleri

Class3: DBLog işlemleri

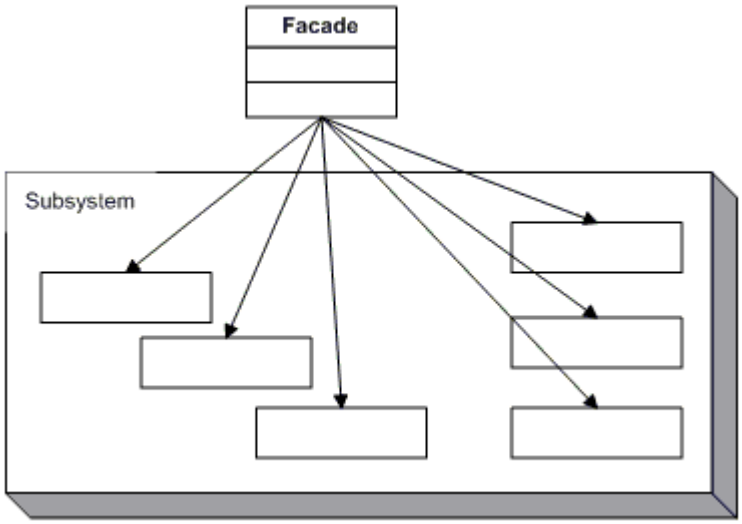
Class4: EmailLog işlemleri

Facade tipi istemci ile bu sınıflar arasında yer alır.

İstemci alt sistemlere (sınfılara) doğrudan ulaşmak yerine Facade aracılığı ile bu class'ları kullanabilir.

İstemci bu sınıflardan soyutlanmış ve nasıl çalıştıkları ile ilgilenmemektedir. Bu sınıflar farklı library içerisinde yer alıyor da olabilir.

Facade tipi uygulamadan çıkarıldığında mevcut sistem bundan etkilenmemektedir. Çünkü Sınıfların tümü bağımsız ve tek başına da kullanılabilir bir yapıdadır.



```
Örnek uygulama
namespace ConsoleApplication1
{
    // Altsistem 1
    class AltSistem1
    {
        public void Method1()
        {
            Console.WriteLine("Alt Sistem 1 Method1 ");
        }
    }

    // Altsistem 2
    class AltSistem2
    {
        public void Method2()
        {
            Console.WriteLine("Alt Sistem 2 Method 2");
        }
    }

    // Facade class
    class Facade
    {
        private AltSistem1 _altSistem1;
        private AltSistem2 _altSistem2;

        public Facade()
        {
            _altSistem1 = new AltSistem1();
            _altSistem2 = new AltSistem2();

        }

        public void Method1()
        {
            Console.WriteLine("Method1() Çalıştırıldı");
            _altSistem1.Method1();
        }

        public void Method2()
        {
            Console.WriteLine("Method2() Çalıştırıldı");
            _altSistem2.Method2();
        }
    }

    class Program
    {
        public static void Main()
        {
            Facade facade = new Facade();
            facade.Method1();
            facade.Method2();
            Console.ReadKey();
        }
    }

}
```

Ekran Çıktısı

```
Method1() Çalıştırıldı
Alt Sistem 1 Method1
Method2() Çalıştırıldı
Alt Sistem 2 Method 2
```

Görüldüğü gibi Facade desenini kodlarımızda bilinçli olarak kullanmasak da kodlama aşamasında mantıksal olarak yaptığımız bir ayrım olarak da belki daha öncede kullanmışızdır.

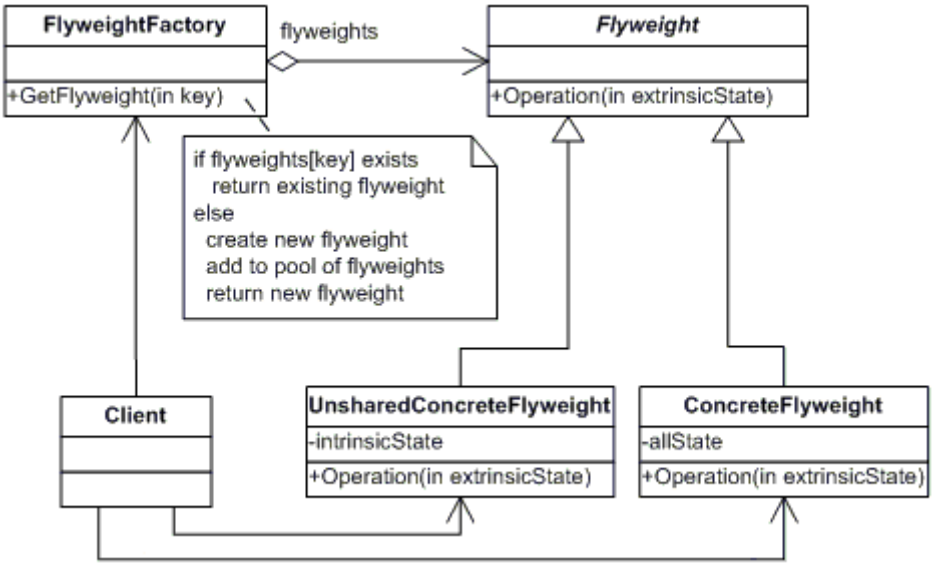
Flyweight Design Pattern

Structural Patterns gurubu içerisinde yer alır.

Bellek tüketimini optimize ederek belleği gereksiz işlemlerden arındırp daha verimli kullanmak için tasarlanmış bir tasarım desenidir.

Nesnelerin bellek tüketimleri nesne sayısı ile doğru ortantılı olarak artmaktadır. FlyWeight tasarım deseni yapıca aynı nesneleri bellekte

çokca oluşturmak yerine her bir nesnenin bir kopyasını oluşturmak ve oluşturulan nesneleri ortak bir noktada tutup paylaştırma işlemini yerine getirir.



Flyweight
Nesnenin ortak özelliklerini tutan interface veya abstract class.

ConcreteFlyweight
Flyweight şablonunu uygulayan farklı Nesneler (örneğimizdeki her bir harf)

FlyweightFactory
Nesneleri ortak bir noktada tutan ve paylaşımını sağlayan yapı.

Client
İstemci uygulama

İncelediğim neredeyse tüm kaynaklarda Flyweight pattern kullanımı konusunda
kelime işlemci uygulamaları örneği verilmiş. Bir yazıda geçen harfler bir nesne olarak temsil edilmiş ve bellekte her harf basımında yeni nesne üretmek yerine her harf için bellekte bir nesne üretmek ve yeniden talep halinde ortak noktadan çağırıp kullanmak örneklendirilmiş.

Örneğin 5000 satırılık sadece harften oluşan bir yazı için bellekte 5000 nesne üretmek yerine alfabeyi temsilen 29 nesne tutulmuş. Her bir nesne bellekte 1 byte kaplayacağı varsayılırsa 5000 byte'lık bellek tüketimi 29 byte'a düşürülmüş hafifletilmiş oluyor.

```
namespace ConsoleApplicaiton1
{
```

```
class Program
{
    static void Main()
    {
        string document = "AAZZBBZB";
        char[] chars = document.ToCharArray();

        CharacterFactory factory = new CharacterFactory();

        // extrinsic state
        int pointSize = 10;

        // For each character use a flyweight object
        foreach (char c in chars)
        {
            pointSize++;
            Character character = factory.GetCharacter(c);
            character.Display(pointSize);
        }

        Console.ReadKey();
    }
}

//FlyweightFactory class
class CharacterFactory
{
    private Dictionary<char, Character> _characters =
      new Dictionary<char, Character>();

    public Character GetCharacter(char key)
    {
        Character character = null;
        if (_characters.ContainsKey(key))
        {
            character = _characters[key];
        }
        else
        {
            switch (key)
            {
                case 'A': character = new CharacterA(); break;
                case 'B': character = new CharacterB(); break;
                //...
                case 'Z': character = new CharacterZ(); break;
            }
            _characters.Add(key, character);
        }
        return character;
    }
}

//Flyweight
abstract class Character
{
    protected char symbol;
    protected int width;
    protected int height;
    protected int ascent;
    protected int descent;
    protected int pointSize;
```

```
        public abstract void Display(int pointSize);
    }


    // ConcreteFlyweight
    class CharacterA : Character
    {
        // Constructor
        public CharacterA()
        {
            this.symbol = 'A';
            this.height = 100;
            this.width = 120;
            this.ascent = 70;
            this.descent = 0;
        }

        public override void Display(int pointSize)
        {
            this.pointSize = pointSize;
            Console.WriteLine(this.symbol +
              " (pointsize " + this.pointSize + ")");
        }
    }

    //ConcreteFlyweight
    class CharacterB : Character
    {
        public CharacterB()
        {
            this.symbol = 'B';
            this.height = 100;
            this.width = 140;
            this.ascent = 72;
            this.descent = 0;
        }

        public override void Display(int pointSize)
        {
            this.pointSize = pointSize;
            Console.WriteLine(this.symbol +
              " (pointsize " + this.pointSize + ")");
        }
    }

    //ConcreteFlyweight
    class CharacterZ : Character
    {
        public CharacterZ()
        {
            this.symbol = 'Z';
            this.height = 100;
            this.width = 100;
            this.ascent = 68;
            this.descent = 0;
        }

        public override void Display(int pointSize)
        {
            this.pointSize = pointSize;
            Console.WriteLine(this.symbol +
              " (pointsize " + this.pointSize + ")");
```

```
        }
    }
}
```

Ekran Çıktısı

```
A (pointsize 11)
A (pointsize 12)
Z (pointsize 13)
Z (pointsize 14)
B (pointsize 15)
B (pointsize 16)
Z (pointsize 17)
B (pointsize 18)
```

Her bir harfi temsilen ConcreteFlyweight yapıları oluşturulmuş ve bu yapılar Flyweight abstract class'ını uygulamış ve bu sayede ortak nesne yapısı
sağlamışlardır. FlyweightFactory yapısı Flyweight arayüzünü uygulayan nesneleri tutan ve paylaştıran bir yapıda çalışmaktadır. Eğer nesne daha önce
üretildi ise üretilen nesneyi, eğer üretilmedi ise yeni ürettiği nesneyi dönmektedir.
Extrinsic state nesnelerin harici durumudur. Örnekte her bir karakterin Point Size değeri farklıdır.
Yada konumları farklı olabilir. Örneğin 3.satır 5.sütun gibi..

Proxy Design Pattern

Structural Patterns gurubu içerisinde yer alır.

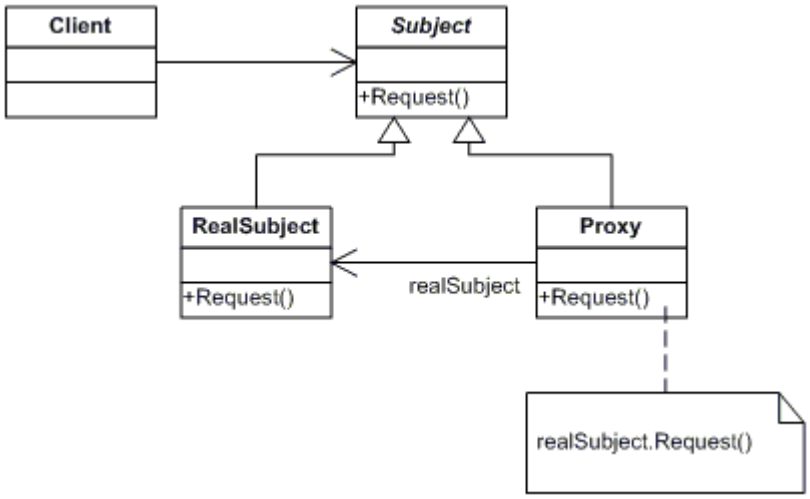
istemcinin kullanacağı bir nesneye kontrol ve erişim için vekillik sağlanması gerektiği durumlarda kullanılır.

Proxy tasarım deseni performansa açısından oluşturulması zaman alan karmaşık bir nesneyi basit bir nesne ile temsil etmektedir.

Performans, Nesnenin farklı network yapısında bulunması ve yetkilendirme ön hazırlık gerektiği durumlarda kullanılır.

İstemci ile nesne arasında kalan bir yapı olarak düşünülebilir.

Network internet üzerinden taşınan bellek kullanımı ve kapladığı alan yüksek nesneler video resim gibi yapılar örneklerde incelenmiş.



Subject
interface abstract class

Real Subject
arayüzünü uygulayan asıl nesne

Proxy
realsubject kullanan bünyesinde içeren class

```
namespace ConsoleApplication1
{
    // Client
    class Program
    {
        static void Main()
        {
            Proxy proxy1 = new Proxy();
            Proxy proxy2 = new Proxy();

            proxy1.Request();
            proxy2.Request();
            proxy2.Request();
```

```
            Console.Read();
        }
    }

    // Subject
    abstract class Subject
    {
        public abstract void Request();
    }

    // RealSubject
    class RealSubject : Subject
    {
        public override void Request()
        {
            Console.WriteLine("Called RealSubject.Request()");
        }
    }

    // Proxy
    class Proxy : Subject
    {
        RealSubject realSubject;

        public override void Request()
        {
            if (realSubject == null)
            {
                realSubject = new RealSubject();
            }

            realSubject.Request();
        }
    }
}
```

Chain of Responsibility

Behavioral Patterns gurubu içerisinde yer alır.

Sorumluluk Zinciri anlamına gelir.

İstemcinin gönderdiği mesaj veya isteğin sanal bir zincir üzerinde gezdirilmesi ve

işlemin icrası konusunda asıl sorumlu olan zincir halkasının ilgili işleri yapması olarak tanımlanabilir.

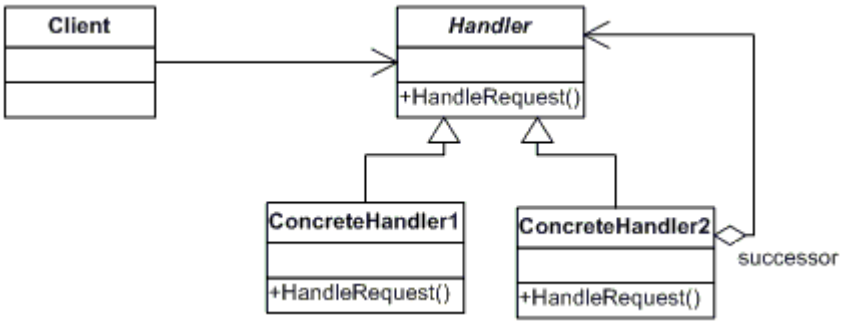
Bu zincirin her bir halkası bir nesne olarak temsil edilir ve tümü birbirinden bağımsız bir şekilde çalışabilir.

Birbirlerine Zayıf bir şekilde (Loosly Coupled) bağlıdırlar.

Zincir halkaları gelen istekleri işlemek ve cevap vermek zorundadır.

Zincirde yer alan tüm zincir halkaları (işleyiciler) ayrı bir sorumluluğa sahiptir.

Zincire bırakılan tüm mesaj yada istek zincirin tüm halkaları tarafından işlenebilir ve diğer halkalara yönlendirilebilir.



Handler
Abstract Class veya Interface olarak tasarlanacak. İşleyici nesenlerin (zincir halkalarının) şablonunu oluşturacak

ConcreteHandler
Handler arayüzünü uygulayan İşleyici nesneler. (Zincirin asıl halkaları)

Client
İstemci Uygulama

Successor
Zincirdeki bir sonraki halkayı temsil ediyor.

İstemcinin gönderdiği talep işleyici zincir halkaları tarafından değerlendirilir. Eğer talep işlem gören zincir halkasının sorumluluk

alanında ise talebi gerçekleştirir. Eğer sorumluluk alanında değilse bir sonraki üst zincir halkasına talebi gönderir.

İşlem aşaması talebin sorumlu olduğu zincire ulaşması ve işlemin sonlanmasına kadar devam eder.

Pattern kullanım alanı konusunda verilebilecek örnekler

Exception yapıları, Loglama, Mail spam filtreleri, HttpHandler yapısı,

Biz örneğimizde bir satın alma işleminde ödeme yetkisini konu ile ilişkilendirmeye çalışacağız.

Memur 1000 TL (max)

Sef 5000 TL (max)

Müdür sınırsız

yetkili olmaları durumunda ödeyebilecekleri tutarlar.

```
namespace ConsoleApplication1
{
    // Handler
    abstract class CORHandler
    {
        protected CORHandler successor;

        public void Successor(CORHandler successor)
        {
            this.successor = successor;
        }

        public abstract void Istek(OdemeYetkisi odemeYetkisi);
    }

    // ConcreteHandler
    class Memur : CORHandler
    {
        public override void Istek(OdemeYetkisi odemeYetkisi)
        {
            if (odemeYetkisi.Yetki == true && odemeYetkisi.Miktar < 1000)
            {
                Console.WriteLine("{0} Ödeme onaylandı.",
this.GetType().Name);
            }
            else if (successor != null)
            {
                successor.Istek(odemeYetkisi);
            }
        }
    }

    // ConcreteHandler
    class Sef : CORHandler
    {
        public override void Istek(OdemeYetkisi odemeYetkisi)
        {
            if (odemeYetkisi.Yetki == true && odemeYetkisi.Miktar < 5000)
            {
                Console.WriteLine("{0} Ödeme onaylandı.",
this.GetType().Name);
            }
            else if (successor != null)
            {
                successor.Istek(odemeYetkisi);
            }
        }
    }

    // ConcreteHandler
    class Mudur : CORHandler
    {
        public override void Istek(OdemeYetkisi odemeYetkisi)
```

```
        {
            if (odemeYetkisi.Yetki == true)
            {
                Console.WriteLine("{0} Ödeme onaylandı.",
this.GetType().Name);
            }
            else if (successor != null)
            {
                successor.Istek(odemeYetkisi);
            }
        }
    }

    class OdemeYetkisi
    {
        public int Miktar { get; set; }
        public bool Yetki { get; set; }
    }

    class Program
    {
        static void Main()
        {
            CORHandler memur = new Memur();
            CORHandler sef = new Sef();
            CORHandler mudur = new Mudur();

            memur.Successor(sef);
            sef.Successor(mudur);

            OdemeYetkisi odeme = new OdemeYetkisi() { Miktar = 800, Yetki
= true };
            memur.Istek(odeme);

            odeme = new OdemeYetkisi() { Miktar = 4000, Yetki = true };
            memur.Istek(odeme);

            odeme = new OdemeYetkisi() { Miktar = 2000, Yetki = true };
            sef.Istek(odeme);

            odeme = new OdemeYetkisi() { Miktar = 6000, Yetki = true };
            sef.Istek(odeme);

            odeme = new OdemeYetkisi() { Miktar = 10000, Yetki = true };
            mudur.Istek(odeme);

            Console.ReadKey();
        }
    }


}
```

Ekran Çıktısı

```
Memur Ödeme onaylandı.
Sef Ödeme onaylandı.
Sef Ödeme onaylandı.
Mudur Ödeme onaylandı.
Mudur Ödeme onaylandı.
```

Command Design Pattern

Behavioral Design Pattern gurubu içerisinde yer alır.

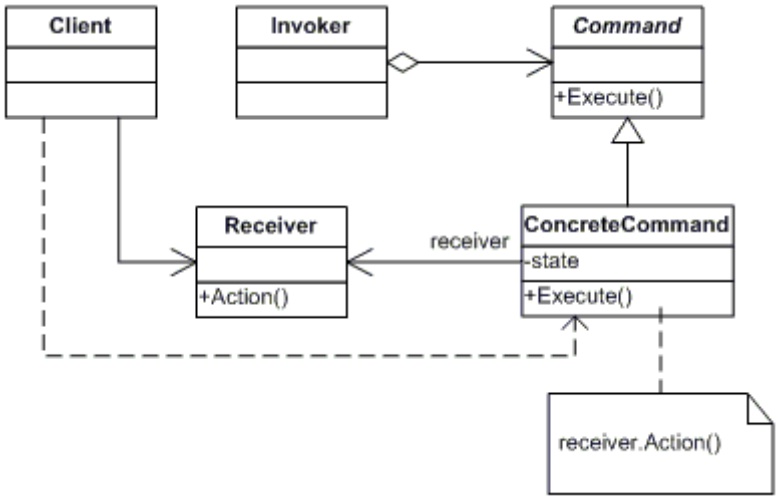
Nesneyi tanımadığımız, nesnenin gerçekleştireceği işlem hakkında hiç bişey bilmediğimiz durumlar karşında kullanabileceğimiz bir patterndir.

Talep edilen işlemin kodları sarmalanarak bir nesne haline getirilerek alıcı nesne aracılığı ile işlemin icrasını gerçekleştirebiliriz.

Gündelik hayatta cep telefonlarımızı sıkça kullanıyoruz.

Telefonun elektronik yapısı ve üzerindeki tuşların altında işletilen işlemler hakkında hiç bir bilgimiz yok. Biz aramak istediğimiz numara ve arama işlemini başlatmak için komutlar göndererek istediğimiz işlemi icra ediyoruz.

Arama yapmak için hangi komutlara hangi sırada basacağımızı biliyoruz. Gerisini bilmek ve işletmek telefonun işi.



Command
Komut için bir interface yada abstract class

Concrete Command
Command arayüzü uygulayan komut yapısı

Client :
İstemci

Invoker
İşlemi çağıran yapı

Receiver
İşlem yapacak metot gövdeleri.

```
namespace ConsoleApplication1
{
    class Program
    {
        // Command
        abstract class Command
        {
            protected Receiver receiver;

            public Command(Receiver receiver)
            {
                this.receiver = receiver;
            }

            public abstract void Execute();
        }

        // Invoker' class
        class Invoker
        {
            private Command _command;

            public void SetCommand(Command command)
            {
                this._command = command;
            }

            public void ExecuteCommand()
            {
                _command.Execute();
            }
        }

        // ConcreteCommand1
        class ConcreteCommand1 : Command
        {
            private string strParameter;
            public ConcreteCommand1(Receiver receiver, string parameter)
                : base(receiver)
            {
                this.strParameter = parameter;
            }

            public override void Execute()
            {
                receiver.Action(this.strParameter);
            }
        }

        // ConcreteCommand2
        class ConcreteCommand2 : Command
        {
            public ConcreteCommand2(Receiver receiver)
                : base(receiver)
            {
            }

            public override void Execute()
            {
                receiver.Action2();
            }
        }
```

```
        // Receiver
        class Receiver
        {
            public void Action(string parameter)
            {
                Console.WriteLine("Called Receiver.Action With Parameter :
" + parameter);
            }

            public void Action2()
            {
                Console.WriteLine("Called Receiver.Action 2");
            }
        }

        // Client
        static void Main()
        {
            Receiver receiver = new Receiver();
            Command command = new ConcreteCommand1(receiver, "ABCDEF");
            Command command2 = new ConcreteCommand2(receiver);
            Invoker invoker = new Invoker();

            invoker.SetCommand(command);
            invoker.ExecuteCommand();

            invoker.SetCommand(command2);
            invoker.ExecuteCommand();

            Console.ReadKey();
        }
    }
}
```

Ekran Çıktısı

Called Receiver.Action With Parameter : ABCDEF
Called Receiver.Action 2

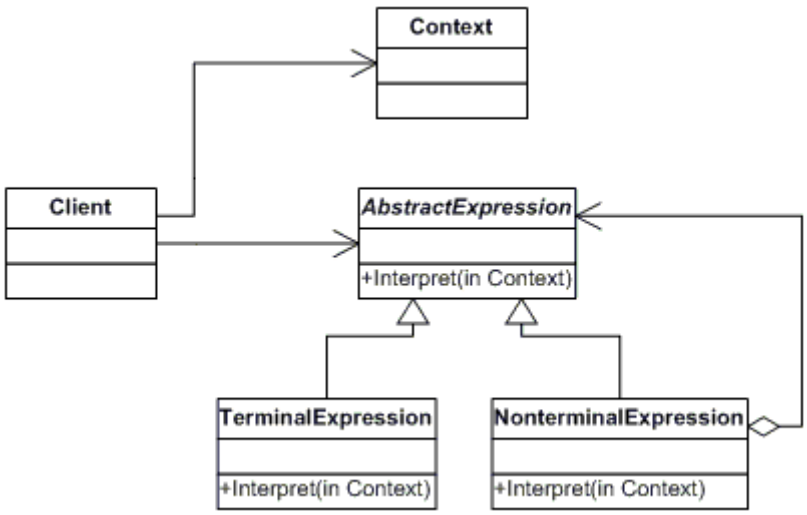
Interpreter Design Pattern

Behavioral Design Patterns Gurubu içerisinde yer alır.

Yorumlayıcı Kalıbı

Mantıksal kalıpların çözümlenmesi ve bir bütün içerisinde yer almasını sağlamak amacıyla kullanılmaktadır. Dil bilgisi kuralları gibi kalıplar içerisinde yer alan ifadelerin yorumlanması amacıyla kullanılması tercih edilmektedir.

Kısacası roma rakamı gibi bir sayının MCMXXVIII nasıl çevirleceğini kurallar bütünü içerisinde düzenleyen yapıdır.



AbstractExpression :
Yorumlama işlemini yapan yapının arayüzü. Interface veya Abstract Class

TerminalExpression
İşleme konu olan değerler.

NonterminalExpression
Diğer değerler

Context :
Yorumlanacak içerik

Client
İstemci uygulama

```
namespace ConsoleApplication1
{
    class Program
    {
        static void Main()
        {
            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);
            }

            Console.WriteLine("{0} = {1}",
              roman, context.Output);

            Console.ReadKey();
        }
    }

    //Context
    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; }
        }
    }

    // AbstractExpression
    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();
    }

    // TerminalExpression
    class ThousandExpression : Expression
    {
        public override string One() { return "M"; }
        public override string Four() { return " "; }
        public override string Five() { return " "; }
        public override string Nine() { return " "; }
        public override int Multiplier() { return 1000; }
    }

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

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

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

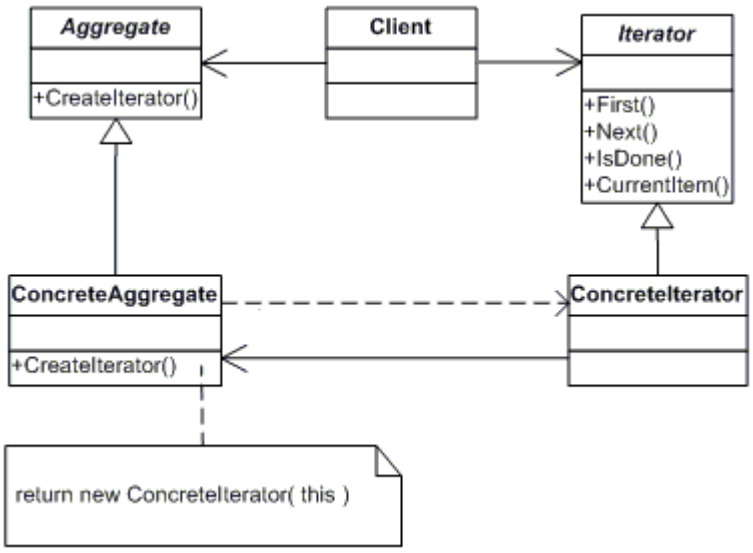
Iterator Design Pattern

Iterator Design Pattern Behavioral Patterns gurubunda yer alır.

Bir nesne kümesi içindeki elemanlara erişim ve elemanlar arasında dolaşma ihtiyaçları için kullanılır.

Iterator Design Patterni .Net içerisinde bilinçli olarak kullanmasak da arkaplanda çalışan yapılar aracılığı ile (örneğin IEnumerable arayüzü)

kodlarımız içinde sıkça kullanırız.



Aggregate

Kolleksiyonun dolaşım işlemleri için Concrete Iterator tipli metodun uygulanması için sunduğu

Interface yada Abstract Class.

Kolleksiyon olarak tanımlanacak sınıflar bu yapıyı uygulamalıdır.

ConcreteAggreagate

Kolleksiyonların tutulduğu sınıf.

Iterator: Kolleksiyon içindeki elemanlara erişim için gerekli yapının şablonu Interface yada Abstract Class.

ConcreteIterator

Kolleksiyon içindeki elemanlara erişim için gerekli asıl iterator yapısı.

Örneğimizde oluşturulan bir kolleksiyon içerisinde iterator pattern uygulanacak ve kolleksiyon elemanları arasında dolaşım sağlanacaktır.

DoFactory üzerindeki hazır örnek alınmıştır.

```
namespace ConsoleApplication1
{
    class MainApp
    {
        static void Main()
        {
            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 = new Iterator(collection);

            iterator.Step = 2;

            Console.WriteLine("Iterating over collection:");

            for (Item item = iterator.First();
                !iterator.IsDone; item = iterator.Next())
            {
                Console.WriteLine(item.Name);
            }

            Console.ReadKey();
        }
    }

    //Item
    class Item
    {
        private string _name;

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

        public string Name
        {
            get { return _name; }
        }
    }

    //Aggregate
    interface IAbstractCollection
    {
        Iterator CreateIterator();
    }

    //ConcreteAggregate
    class Collection : IAbstractCollection
    {
        private ArrayList _items = new ArrayList();

        public Iterator CreateIterator()
        {
```

```
            return new Iterator(this);
        }

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

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


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


    //ConcreteIterator
    class Iterator : IAbstractIterator
    {
        private Collection _collection;
        private int _current = 0;
        private int _step = 1;

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

        public Item First()
        {
            _current = 0;
            return _collection[_current] 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; }
        }
    }
}
```

Ekran Çıktısı

```
Iterating over collection:
Item 0
Item 2
Item 4
Item 6
Item 8
```

iterator.Step = 2;

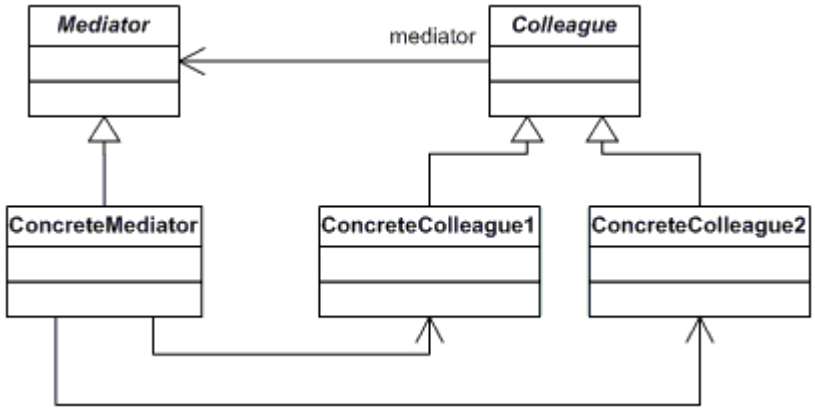
kodu ile koleksiyon üzerinde 2'şer adımla dolaşılmış ve kolleksiyon içindeki elemanlar alınmıştır.

.Ayrıca Net framework içindeki hazır yapıları IEnumerable, IEnumerator gibi kavramları inceleyebilirsiniz.

Mediator Design Pattern

Mediator Design Pattern nesnelerin aralarındaki iletişimin tek bir noktadan sağlanması ve koordine edilmesi gerektiği durumlarda kullanılır.

Nesneler birbirleri ile doğrudan konuşmak yerine merkezi bir yapı aracılığı ile haberleşirler bu sayede nesneler arasında bağımlılık azalır. Nesneler birbirlerinin kim olduklarını bilmeden merkez aracılığı ile haberleşebilirler.



Pattern kullanım örneğinde dofactory üzerindeki bir chat uygulamasını konumuz ile ilişkilendirmeye çalışacağız.

Örneğimizde farklı guruplar içinde yer alan kullanıcılar birbirleri ile merkez aracılığı ile konuşmaktadır.

```
namespace ConsoleApplication1
{

    //Mediator
    abstract class Mediator
    {
        public abstract void KayitOl(Katilimci katilimci);
        public abstract void MesajGonder(string kimden, string kime,
string mesaj);
    }

    //ConcreteMediator
    class ConcreteMediator : Mediator
    {
        private Dictionary<string, Katilimci> _katilimcilar = new
Dictionary<string, Katilimci>();

        public override void KayitOl(Katilimci katilimci)
        {
            if (!_katilimcilar.ContainsValue(katilimci))
            {
                _katilimcilar[katilimci.Ad] = katilimci;
            }
            katilimci.concreteMediator = this;
        }

        public override void MesajGonder(string kimden, string kime,
string mesaj)
        {
```

```
            Katilimci katilimci = _katilimcilar[kime];
            if (katilimci != null)
            {
                katilimci.MesajAlici(kimden, mesaj);
            }
        }
    }

    // Colleague
    class Katilimci
    {
        private ConcreteMediator _concreteMediator;
        private string _ad;

        public Katilimci(string ad)
        {
            this._ad = ad;
        }

        public string Ad
        {
            get { return _ad; }
        }

        public ConcreteMediator concreteMediator
        {
            set { _concreteMediator = value; }
            get { return _concreteMediator; }
        }

        public void MesajGonder(string kime, string mesaj)
        {
            _concreteMediator.MesajGonder(_ad, kime, mesaj);
        }

        public virtual void MesajAlici(string kimden, string mesaj)
        {
            Console.WriteLine("{0} to {1}: '{2}'", kimden, Ad, mesaj);
        }
    }


    //ConcreteColleague
    class ConcreteColleague1 : Katilimci
    {
        public ConcreteColleague1(string ad)
            : base(ad)
        {
        }

        public override void MesajAlici(string kimden, string mesaj)
        {
            Console.Write("Kime -> ConcreteColleague1: ");
            base.MesajAlici(kimden, mesaj);
        }
    }

    //ConcreteColleague
    class ConcreteColleague2 : Katilimci
    {
        public ConcreteColleague2(string ad)
            : base(ad)
```

```
        {
        }
        public override void MesajAlici(string kimden, string mesaj)
        {
            Console.Write("Kime -> ConcreteColleague2: ");
            base.MesajAlici(kimden, mesaj);
        }
    }


    //Client
    class Program
    {
        static void Main(string[] args)
        {
            ConcreteMediator sohbetOdasi = new ConcreteMediator();

            Katilimci Can = new ConcreteColleague1("Can");
            Katilimci Canan = new ConcreteColleague1("Canan");
            Katilimci Baris = new ConcreteColleague2("Barış");
            Katilimci Ahmet = new ConcreteColleague2("Ahmet");
            Katilimci Selvi = new ConcreteColleague1("Selvi");


            sohbetOdasi.KayitOl(Can);
            sohbetOdasi.KayitOl(Canan);
            sohbetOdasi.KayitOl(Baris);
            sohbetOdasi.KayitOl(Ahmet);
            sohbetOdasi.KayitOl(Selvi);

            // Chatting participants
            Selvi.MesajGonder("Can", "Selam Can");
            Canan.MesajGonder("Can", "Selam Can Yemeğe Çıkacakmısın ?");
            Baris.MesajGonder("Ahmet", "İstediğim evraklar hazır mı ?");
            Canan.MesajGonder("Ahmet", "Toplantı saat 3'te ");
            Ahmet.MesajGonder("Barış", "İstediğin evraklar hazır");


            Console.Read();
        }
    }
}
```

Ekran Çıktısı

```
Kime -> ConcreteColleague1: Selvi to Can: 'Selam Can'
Kime -> ConcreteColleague1: Canan to Can: 'Selam Can Yemeğe Çıkacakmısın ?'
Kime -> ConcreteColleague2: Barış to Ahmet: 'İstediğim evraklar hazır mı ?'
Kime -> ConcreteColleague2: Canan to Ahmet: 'Toplantı saat 3'te '
Kime -> ConcreteColleague2: Ahmet to Barış: 'İstediğin evraklar hazır'
```

ConcreteColleague farklı gurupları network'leri temsil etmektedir bu guruplar içlerinde kullanıcıları barındırmaktadır.

Memento Design Pattern

Memento Design Pattern bir nesnenin o anki durumunun saklanması ve durumunda meydana gelen değişimler sonrası, durumunun saklandığı haline geri döndürülmesi
işlemini düzenleyen yapıdır. Behavioral Patterns gurubu içerisinde yer alır.

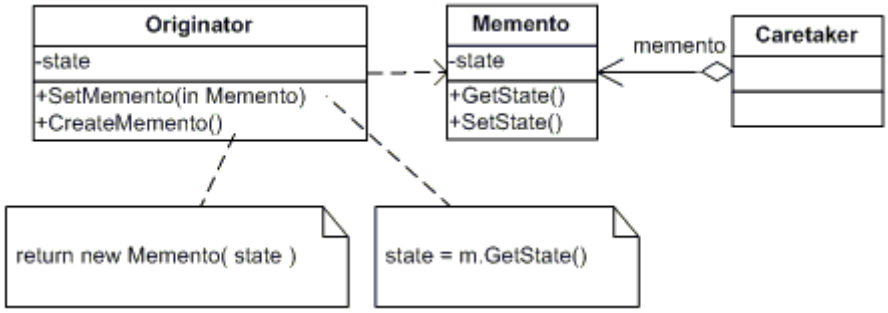
Kullanım ihtiyacı konusunda ne tür örnekler verebiliriz ?

İşletim sistemi veya Programlarda Geri Al Fonksiyonları

Grafik/Çizim programlarında history yapıları ile yapılan işlemlerin geriye döndürülmesi.

Oyun bölümlerinde ilerlenen seviyenin kayıt edilmesi. İstenildiğinde geçilen seviyelere geri dönüş yapılması.

gibi...



Orginator
Durumu saklanacak ve geri alınacak yapı. (Nesne)

Momento
Orginator nesnesinin anlık görüntülerini tutan yapı.

CareTaker
Güvenli saklayıcı. Momento nesnesinin içeriğine müdahele etmeden güvenli bir şekilde saklayan yapı.

Biz örneğimizde Windows üzerinde bir desktop görüntü teması seçme işlemini konumuz ile ilişkilendirmeye çalışacağız.

Önce Rainly Day Temasını seçeceğiz ardından Windows Classic temasını seçeceğiz daha sonra Windows Classic temasını beğenmeyip Rainly Day temasına geri döneceğiz.

```
namespace ConsoleApplication1
{
    //Orginator
    public class WindowsThemes
    {
        public string ThemeName { get; set; }

        public MementoWindowsThemes Save()
        {
            MementoWindowsThemes memento = new MementoWindowsThemes();
            memento.ThemeName = this.ThemeName;
            return memento;
        }
```

```
        public void Undo(MementoWindowsThemes memento)
        {
            this.ThemeName = memento.ThemeName;
        }
    }

    //Memento
    public class MementoWindowsThemes
    {
        public string ThemeName { get; set; }
    }

    //Care Taker
    public class CareTakerWindowsThemes
    {
        public MementoWindowsThemes Theme { get; set; }
    }

    class Program
    {
        static void Main()
        {
            WindowsThemes theme = new WindowsThemes();
            CareTakerWindowsThemes careTaker = new
CareTakerWindowsThemes();

            theme.ThemeName = "Rainly Day";
            careTaker.Theme = theme.Save();
            Console.WriteLine(theme.ThemeName);

            theme.ThemeName = "Windows Classic";
            Console.WriteLine(theme.ThemeName);

            theme.Undo(careTaker.Theme);
            Console.WriteLine(theme.ThemeName);

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

Observer Design Pattern

Observer Design Pattern Varolan bir nesnenin durumunda meydana gelebilecek değişimler ve bu değişimle ilgilenen diğer nesneler arasındaki ilişkileri inceler.

Behavioral Patterns gurubu içerisinde yer alır. Baş aktörler Nesne ve Gözlemleyicilerdir.

Nesne ile Gözlemleyiciler arasında bire çok ilişki vardır. Bir nesne ve bir yada birden çok gözlemleyici.

Kullanım ihtiyacı konusunda ne tür örnekler verebiliriz ?

Yayıncı / Abone yapıları

Bir ürünün stok, fiyat, miktar durumlarında meydana gelebilecek değişimler ve bu değişimleri takip eden gözlemleyiciler.

Haber kaynaklarındaki değişimler son dakika haberleri ve bunları takip eden gözlemleyiciler.

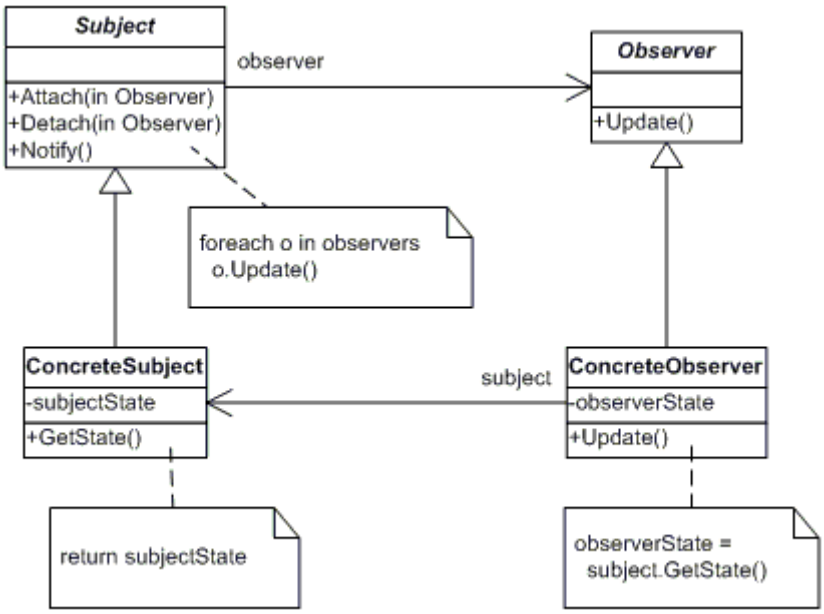
Blog yada RSS kanalları üzerinden içerik nesnesi durumunda meydana gelen bir değişim

yeni içerik ekleme, içerik güncelleme ve bunu takip eden gözlemleyiciler.

MVC ve Event Yapıları gibi...

Observer Design Pattern çok sık bir kullanım oranına sahiptir.

Biz örneğimizde hava durumu ve hava durumundan haberdar olmak isteyen gözlemleyicileri konumuz ile ilişkilendirmeye çalışacağız.



Subject
Durumu gözlemlenecek nesnemiz.

Observer
ConcreteObserver'lar tarafından uygulanacak bir interface veya abstract class. Concrete Observer nesnelerinin ortak şablonu.

ConcreteObserver
Subject durumunda bir değişim olduğunda durumdan haberdar edilecek gözlemleyiciler.

Subject içerisinde Attach ile yeni ConcreteObserver kaydı, Detach ile kayıtlı ConcreteObserver kayıt silinme işlemi,
Notify ile tüm kayıtlı ConcreteObserver'lara nesne durumunda meydana gelen değişim haberi ve değişen nesnenin bir kopyası gönderimi sağlanmaktadır.

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

namespace ConsoleApplication1
{
    //Subject
    public class HavaDurumuRaporu
    {
        private DateTime sonGuncelleme;
        public DateTime SonGuncelleme
        {
            get
            {
                return sonGuncelleme;
            }
            set
            {
                sonGuncelleme = value;
                GozlemleyicileriUyar();
            }
        }

        public string Durum { get; set; }
        public int Sicaklik { get; set; }
        public int Ruzgar { get; set; }
        public string RuzgarYonu { get; set; }

        private List<IGozlemleyici> gozlemleyiciler;

        public HavaDurumuRaporu()
        {
            gozlemleyiciler = new List<IGozlemleyici>();
        }

        public void GozlemleyiciEkle(IGozlemleyici gozlemleyici)
        {
            gozlemleyiciler.Add(gozlemleyici);
        }

        public void GozlemleyiciSil(IGozlemleyici gozlemleyici)
        {
            gozlemleyiciler.Remove(gozlemleyici);
        }

        private void GozlemleyicileriUyar()
        {
            foreach (IGozlemleyici gozlemleyici in gozlemleyiciler)
            {
                gozlemleyici.Guncelle(this);
            }
        }
```

```
    }

    //Observer
    public interface IGozlemleyici
    {
        void Guncelle(HavaDurumuRaporu havaDurumu);
    }

    //Concrete Observer
    public class Gozlemleyici : IGozlemleyici
    {
        public void Guncelle(HavaDurumuRaporu havaDurumuRaporu)
        {
            Console.WriteLine("Yeni hava durumu yayınlandı yayın
tarihi/saati " + havaDurumuRaporu.SonGuncelleme.ToString());
        }

        public string Kimlik { get; set; }
    }

    class Program
    {
        static void Main()
        {

            HavaDurumuRaporu rapor = new HavaDurumuRaporu();

            Gozlemleyici g1 = new Gozlemleyici { Kimlik = "ABC Denizcilik"
};
            rapor.GozlemleyiciEkle(g1);

            Gozlemleyici g2 = new Gozlemleyici { Kimlik = "DEF Yat Limanı"
};
            rapor.GozlemleyiciEkle(g2);

            rapor.Durum = "Parçalı Bulutlu";
            rapor.Ruzgar = 3;
            rapor.RuzgarYonu = "Lodos";
            rapor.Sicaklik = 25;
            Console.WriteLine("Nesne Hava Durumu Yayınlıyor...");
            rapor.SonGuncelleme = DateTime.Now;

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

Ekran Çıktısı

```
Nesne Hava Durumu Yayınlıyor...
Yeni hava durumu yayınlandı yayın tarihi/saati 20.03.2010 11:53:20
Yeni hava durumu yayınlandı yayın tarihi/saati 20.03.2010 11:53:20
```

İlk olarak nesnemizi (Subject) tanımladık. HavaDurumRaporu içerisine hava durumunda yer alan property'leri ekledik. Bu özelliklerden SonGuncelleme property set bloğunda GozlemleyicileriUyar() metodunu çağırdık. Nesnenin Son Guncelleme zamanında bir değişim olduğunda gözlemleyicilere bildiren yapı. Değişim olduğunda GozlemleyicileriUyar() metodunu çağırarak tüm gözlemleyicileri

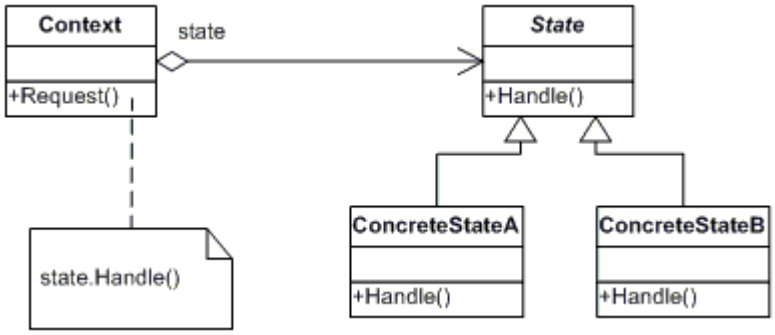
dolaşıp Guncelle metotlarını çağırmaktadır. Yukarıdaki UML şemasındaki Notify fonksiyonu. Gozlemleyicileri gozlemleyiciler generic list yapısında tutup GozlemciEkle ve GozlemciSil metodları ile ekleme çıkarma yaptık. UML şemasındaki Attach, Detach fonksiyonları.
Client yapısında bir HavaDurumuRaporu nesnesi oluşturuldu. 2 gözlemleyici eklendi. Nesnenin SonGuncelleme zamanı yani durumu değişti. Tanımlı tüm Concrete Observer yapılarının Guncelle() metodu çağrılarak bu değişimden haberdar edildi.

State Design Pattern

State Design Pattern Behavioral patterns gurubunda yer alır.

Nesnenin durumunda oluşabilecek bir değişim nesnenin davranışlarını etkiliyor, davranışlar nesne durumuna göre değişim gösteriyorsa

nesnenin davranışlarını nesnenin duruma göre değiştirmesi için state tasarım deseni kullanılır.



Context
Durumu izlenecek nesne.

State
Nesnenin tüm durumlar için arayüzü ortak şablonu. Abstract Class veya Interface

ConcreteState
Contex yapısının durumu davranışları.

Biz örneğimizde bir Printer cihazını ve duruma göre davranışlarını incelemeye çalışacağız.

```
namespace ConsoleApplication1
{
    //State
    abstract class PrinterState
    {
        public abstract void GetState(Printer context);
    }

    //Context
    class Printer
    {
        private PrinterState printerState;
        public PrinterState State
        {
            get
            {
                return printerState;
            }
            set
            {
                printerState = value;
                printerState.GetState(this);
```

```
            }
        }

        public Printer()
        {
            State = new StatePrinterLoading();
        }

        public void PrintDocument(string documentPath)
        {
            Console.WriteLine("Yazdırılacak döküman geldi");
            State = new StatePrinterBusy();
            Console.WriteLine(documentPath + " yazdırıldı.");
            State = new StatePrinterReady();
        }
    }

    //Concrete State
    class StatePrinterLoading : PrinterState
    {
        public StatePrinterLoading()
        {
            Console.WriteLine("Printer Loading");
        }

        public override void GetState(Printer context)
        {
            Console.WriteLine("Printer Yüklendi Kullanıma Hazır");
            context.State = new StatePrinterReady();
        }
    }

    //Concrete State
    class StatePrinterReady : PrinterState
    {
        public StatePrinterReady()
        {
            Console.WriteLine("Printer Ready");
        }
        public override void GetState(Printer context)
        {
            Console.WriteLine("Yeni Yazdırma İşlemi İsteği İçin
Bekliyor.");
        }
    }

    //Concrete State
    class StatePrinterBusy : PrinterState
    {
        public StatePrinterBusy()
        {
            Console.WriteLine("Printer Busy");
        }
        public override void GetState(Printer context)
        {
            Console.WriteLine("Printer Meşgul Yazdırma İşlemi Yapıyor.");
        }
    }

    //Client
    class Program
    {
```

```
        static void Main(string[] args)
        {
            Console.WriteLine("Kitap.pdf dosyası yazdırılacak");
            Printer printer = new Printer();
            printer.PrintDocument("Kitap.pdf");
            Console.Read();
        }
    }
}
```

Ekran Çıktısı

```
Kitap.pdf dosyası yazdırılacak
Printer Loading
Printer Yüklendi Kullanıma Hazır
Printer Ready
Yeni Yazdırma İşlemi İsteği İçin Bekliyor.
Yazdırılacak döküman geldi
Printer Busy
Printer Meşgul Yazdırma İşlemi Yapıyor.
Kitap.pdf yazdırıldı.
Printer Ready
Yeni Yazdırma İşlemi İsteği İçin Bekliyor.
```

Strategy Design Pattern

Behavioral Patterns gurubunda yer almaktadır. Hatırlayacağınız gibi Behavioral Patterns gurubu belirli bir işi yerine getirmek için çeşitli sınıfların nasıl birlikte davranış göstereceğini belirleyen desenlerden oluşuyordu.

Genelde bir işlem ve bu işlemin bir yapılış şekli vardır.

İhtiyaçlar sonucu bir yapılış şekli birden fazla ve değişken bir yapıda olabilir.

Yapılış şeklinin seçim kararı tasarım zamanı (design time) yada çalışma zamanı (run time) da veriliyor olabilir.

Strategy Design Pattern

Bir işlemin birden fazla yapılış şekli

ve bu yapılış şekillerinin seçim kararı

ve bunların uygulama sırasındaki davranışları

konusunu işleyen tasarım desenidir.

Konuyu örneklendirirsek

Bir mektup yazdınız ve bunu göndereceksiniz.

İşlemimiz

Mektubu PTT ile göndermek.

Bir yapılış şekli

Normal hizmet tarifesinden adrese gönder.(ucuz ve uzun süren bir işlem)

Birden fazla yapılış şekli

Acil gitmesi gerektiği durumda (APS İadeli Taahhütlü Gönderi) ile gönder. (pahalı ve kısa süren bir işlem)

Alıcının internet erişimi varsa Elektronik Posta ile gönder. (ucuz ve en kısa süren bir işlem)

gibi... bir çok seçenek.

Yapılış şekli seçim kararının verilmesi.

Design Time

Mektubu yazdınız normal hizmet tarifesi üzerinden göndermek kararı aldınız ve yola çıktınız.

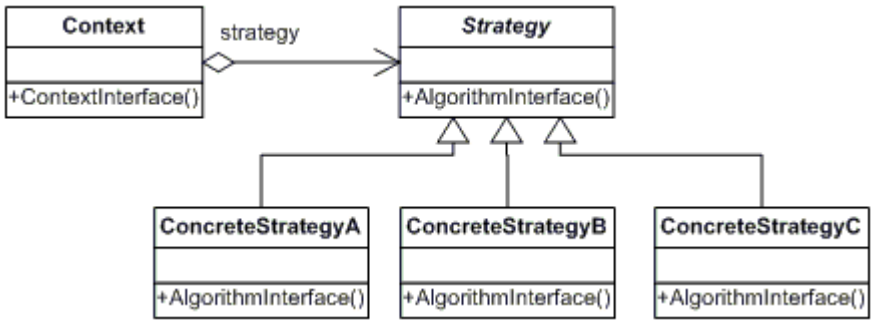
Run Time

Yola çıktıktan sonra telefon geldi ve konunun acil olduğunu en kısa sürede ulaşması gerektiği alıca tarafından size söylendi.

Seçim Kararı

Telefonda alıcıya internet erişimi ve eposta adresi olup olunmandığı soruldu. Alıcı hesabı olmadığını belirtti.

APS ile göndermeye karar verdiniz.



Context
İçerik nesnemiz. İstediği ConcreteStrategy tipini seçerek kullanabiliyor.

ConcreteStrategy
Birden fazla yapılış şekli. Örneğimizde postamızı gönderebileceğimiz farklı kanallar.

Strategy
ConcreteStrategy yapıları için ortak bir şablon sunan Abstract Class yada Interface'lerdir.

İleride oluşacak farklı gönderim kanalları bu yapı uygulanarak geliştirilecek ve kullanacaktır.

```
namespace ConsoleApplication1
{
    abstract class Strategy
    {
        public abstract void Send(string NameSurname, string Address,
string Telephone);
    }

    class Context
    {
        private Strategy _strategy;

        public string NameSurname { get; set; }
        public string Address { get; set; }
        public string Telephone { get; set; }

        public Context(Strategy strategy)
        {
            this._strategy = strategy;
        }
        public void Send()
        {
            _strategy.Send(NameSurname, Address, Telephone);
        }
    }

    class ConcreteStrategy1 : Strategy
    {
        public override void Send(string NameSurname, string Address,
string Telephone)
        {
            Console.WriteLine("Normal Hizmet Tarifesi üzerinden
gönderildi.");
        }
    }

    class ConcreteStrategy2 : Strategy
    {
```

```
        public override void Send(string NameSurname, string Address,
string Telephone)
        {
            Console.WriteLine("APS İadeli Taahhütlü Gönderi tarifesi
üzerinden gönderildi.");
        }
    }

    class ConcreteStrategy3 : Strategy
    {
        public override void Send(string NameSurname, string Address,
string Telephone)
        {
            Console.WriteLine("Elektronik Posta ile gönderildi.");
        }
    }

    class Program
    {
        static void Main()
        {
            Context context;

            // Normal hizmet tarifesi üzerinden göndermek üzere yola
çıktık.
            context = new Context(new ConcreteStrategy1());

            bool isReceivePhoneCalls = true;
            bool isExistAccount = false;

            if (isReceivePhoneCalls == true) // Çağrı geldi
            {
                if (isExistAccount == false) // İnternet hesabı yoksa ?
                    context = new Context(new ConcreteStrategy2()); // APS
                else
                    context = new Context(new ConcreteStrategy3());//
EPOSTA
            }

            //Alıcı bilgilerini dolduruyp yolluyoruz.
            context.NameSurname = "Cin Ali";
            context.Address = "İstanbul";
            context.Telephone = "212-9999999";
            context.Send();

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

Template Method Design Pattern

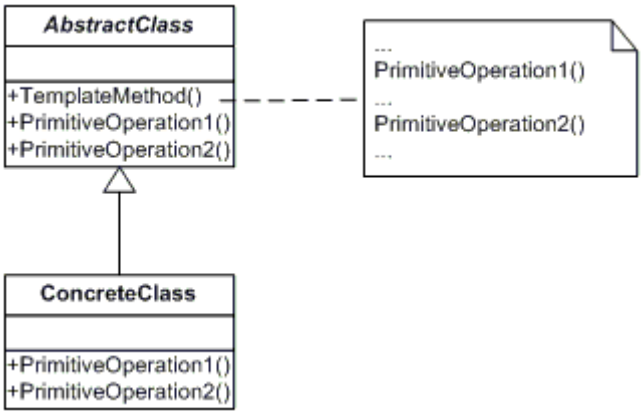
Behavioral Patterns Gurubu içerisinde yer alır.

Bir algoritmanın bütününü oluşturan farklı işlemler soyut bir yapıda altsınıf olarak tanımlanır.

Bu altsınıflar algoritma bütünlüğü için gerekli işlemleri kendi içerisinde implemente ederler.

Bu yolla algoritma altsınıflar tarafından işlenerek altsınıfların istekleri doğrultusunda çalışır.

Algoritmanın değişmeyen kısımları bir kere tanımlanır. Değişecek kısımların tanımlanması alt sınıflara bırakılır. Böylece Ana kalıp değişmeksizin, bazı ara adımlar değiştirilebilir.




```
namespace ConsoleApplication1
{
    // Abstract Class
    abstract class AbstractClass
    {
        public abstract void PrimitiveOperation1();
        public abstract void PrimitiveOperation2();

        // Template method
        public void TemplateMethod()
        {
            PrimitiveOperation1();
            PrimitiveOperation2();
            Console.WriteLine("");
        }
    }

    // ConcreteClass
    class ConcreteClassA : AbstractClass
    {
        public override void PrimitiveOperation1()
        {
            Console.WriteLine("ConcreteClassA.PrimitiveOperation1()");
        }
        public override void PrimitiveOperation2()
        {
            Console.WriteLine("ConcreteClassA.PrimitiveOperation2()");
        }
    }

    // ConcreteClass
    class ConcreteClassB : AbstractClass
```

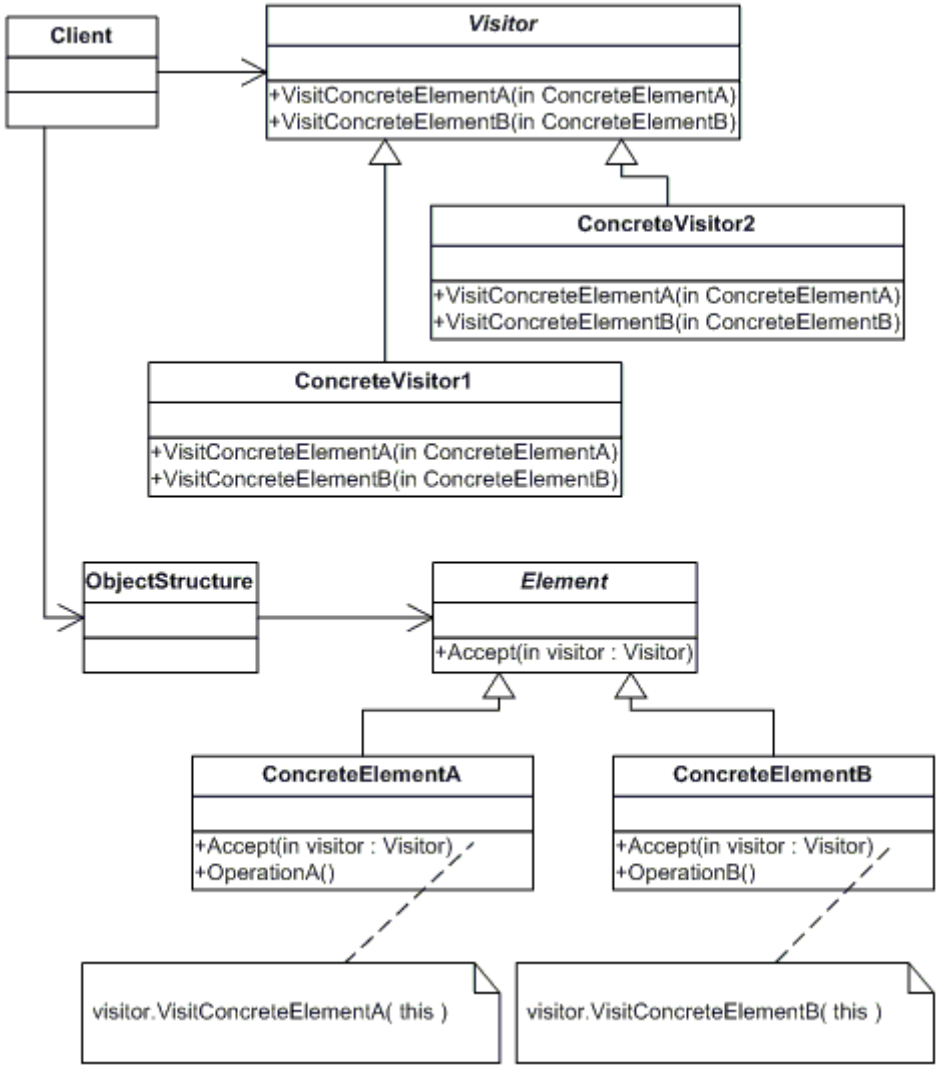
```
    {
        public override void PrimitiveOperation1()
        {
            Console.WriteLine("ConcreteClassB.PrimitiveOperation1()");
        }
        public override void PrimitiveOperation2()
        {
            Console.WriteLine("ConcreteClassB.PrimitiveOperation2()");
        }
    }
    // Client
    class Program
    {
        static void Main()
        {
            AbstractClass aA = new ConcreteClassA();
            aA.TemplateMethod();

            AbstractClass aB = new ConcreteClassB();
            aB.TemplateMethod();

            // Wait for user
            Console.ReadKey();
        }
    }

}

// Örnek 2
namespace ConsoleApplication1
{
    abstract class KrediHesapAlgoritma
    {
        public abstract void Yontem1();
        public abstract void Yontem2();

        public void Hesapla(int tutar, int vade)
        {
            if (tutar < 400)
                Yontem1();
            else
                Yontem2();
            Console.WriteLine("");
        }
    }

    class Konut : KrediHesapAlgoritma
    {
        public override void Yontem1()
        {
            Console.WriteLine("% 0,5 oran ile");
        }
        public override void Yontem2()
        {
            Console.WriteLine("% 2 oran ile");
        }
    }

    class Tasit : KrediHesapAlgoritma
    {
        public override void Yontem1()
        {
```

```
            Console.WriteLine("% 35 oran ile");
        }
        public override void Yontem2()
        {
            Console.WriteLine("% 32 oran ile");
        }
    }


    class Program
    {
        static void Main()
        {
            KrediHesapAlgoritma konut = new Konut();
            konut.Hesapla(500, 10);

            KrediHesapAlgoritma tasit = new Tasit();
            tasit.Hesapla(300, 10);

            Console.ReadKey();
        }
    }

}
```

Visitor

Behavioral Patterns gurubu içerisinde yer alır.

Visitor tasarım şablonu, bir sınıf hiyerarşisinde yeralan sınıflar üzerinde değişiklik yapmadan, bu sınıflara yeni metodların eklenmesini kolaylaştırır.İstenilen metod bir visitor sınıfında implemente edilir. Bileşik bir yapı üzerine yeni işlemler eklenmesine olanak sağlar. Ziyaretçi nesne bileşik yapı içindeki nesneleri tek tek ziyaret ederek gerekli bilgileri toplayıp işleyerek kullanıcıya sunar.




```
using System;
using System.Collections.Generic;

namespace ConsoleApplication1
{
    // Client
    class MainApp
    {
        static void Main()
        {
            ObjectStructure o = new ObjectStructure();
            o.Attach(new ConcreteElementA());
            o.Attach(new ConcreteElementB());
```

```
            ConcreteVisitor1 v1 = new ConcreteVisitor1();
            ConcreteVisitor2 v2 = new ConcreteVisitor2();

            o.Accept(v1);
            o.Accept(v2);

            Console.ReadKey();
        }
    }

    // Visitor
    abstract class Visitor
    {
        public abstract void VisitConcreteElementA(
          ConcreteElementA concreteElementA);
        public abstract void VisitConcreteElementB(
          ConcreteElementB concreteElementB);
    }

    // ConcreteVisitor
    class ConcreteVisitor1 : Visitor
    {
        public override void VisitConcreteElementA(
          ConcreteElementA concreteElementA)
        {
            Console.WriteLine("{0} visited by {1}",
              concreteElementA.GetType().Name, this.GetType().Name);
        }

        public override void VisitConcreteElementB(
          ConcreteElementB concreteElementB)
        {
            Console.WriteLine("{0} visited by {1}",
              concreteElementB.GetType().Name, this.GetType().Name);
        }
    }

    // ConcreteVisitor
    class ConcreteVisitor2 : Visitor
    {
        public override void VisitConcreteElementA(
          ConcreteElementA concreteElementA)
        {
            Console.WriteLine("{0} visited by {1}",
              concreteElementA.GetType().Name, this.GetType().Name);
        }

        public override void VisitConcreteElementB(
          ConcreteElementB concreteElementB)
        {
            Console.WriteLine("{0} visited by {1}",
              concreteElementB.GetType().Name, this.GetType().Name);
        }
    }

    // Element
    abstract class Element
    {
        public abstract void Accept(Visitor visitor);
    }
```

```
    // ConcreteElement
    class ConcreteElementA : Element
    {
        public override void Accept(Visitor visitor)
        {
            visitor.VisitConcreteElementA(this);
        }

        public void OperationA()
        {
        }
    }

    // ConcreteElement
    class ConcreteElementB : Element
    {
        public override void Accept(Visitor visitor)
        {
            visitor.VisitConcreteElementB(this);
        }

        public void OperationB()
        {
        }
    }

    // ObjectStructure
    class ObjectStructure
    {
        private List<Element> _elements = new List<Element>();

        public void Attach(Element element)
        {
            _elements.Add(element);
        }

        public void Detach(Element element)
        {
            _elements.Remove(element);
        }

        public void Accept(Visitor visitor)
        {
            foreach (Element element in _elements)
            {
                element.Accept(visitor);
            }
        }
    }
}
```

Pattern Kullanım Sıklığı Oranları (dofactory)

Creational Patterns

Abstract Factory	% 100
Builder	% 40
Factory Method	% 100
Prototype	% 60
Singleton	% 80

Structural Patterns

Adapter	% 80
Bridge	% 60
Composite	% 80
Decorator	% 60
Facade	% 100
Flyweight	% 20
Proxy	% 80

Behavioral Patterns

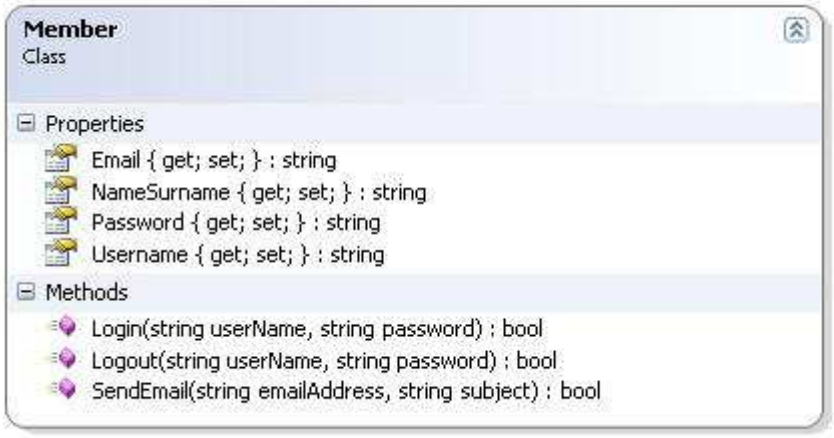
Chain of Responsibility	% 40
Command	% 80
Interpreter	% 20
Iterator	% 100
Mediator	% 40
Memento	% 20
Observer	% 100
State	% 60
Strategy	% 80
Template Method	% 60
Visitor	% 20

Design Principles

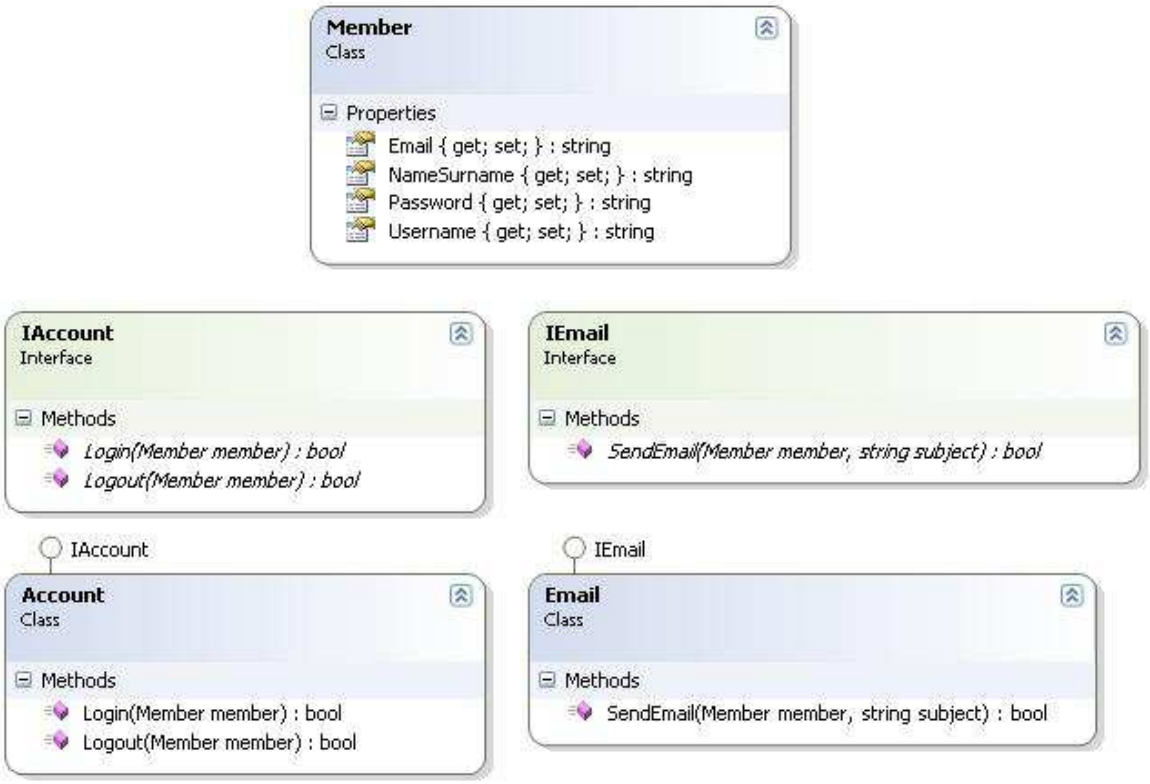
Daha iyi yazılım mimari tasarımı için uygulanılması gereken temel prensiplerdir.

Single Responsibility Principle - SRP

Bir sınıf tek bir sorumluluğa sahip olmalıdır ve sadece sorumlu olduğu görevi yerine getirmelidir.
Single Responsibility ilke olarak bir sınıfın birden fazla sorumluluğu olmasına karşı çıkar.
Sınıftaki sorumluluk sayısı arttıkça bağımlılık artar, yeniden kullanabilirlik azalır, test maliyetleri artar aşırı büyüme gösterir bu da karmaşıklığa neden olur, bakımı zorlaşır.
Pek çok sorumluluğu üstlenmiş iç içe geçmiş bir sınıf yapısı içinde sınıf üyelerini ve farklı işleri yapan metotları barındırıyor. Görünüşte pek çok görevi yerine getirmesine rağmen bu şekilde tasarlanan bir sınıf Single Responsibility ilkesine ters düşmektedir.



Her sınıfa sadece bir sorumluluk yüklenen, sorumlulukların dağıtıldığı, olması gereken mimari yapı aşağıdaki şekilde olmalıdır.


Member
Class
Properties
Email { get; set; } : string
NameSurname { get; set; } : string
Password { get; set; } : string
Username { get; set; } : string
IAccount
Interface
Methods
Login(Member member) : bool
Logout(Member member) : bool
IEmail
Interface
Methods
SendEmail(Member member, string subject) : bool
IAccount
IEmail
Account
Class
Methods
Login(Member member) : bool
Logout(Member member) : bool
Email
Class
Methods
SendEmail(Member member, string subject) : bool

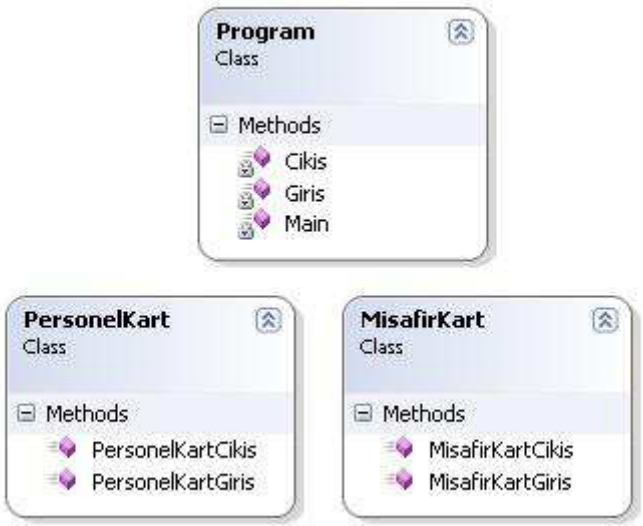
Open Closed Principle - OCP

Yapıların geliştirilmeye, genişletilmeye açık ama değiştirilmeye kapalı olması olarak tanımlanır. Yapılar bir kez kurgulanır ve inşa edilir. Mevcut yapılar ileriki bir zamanda ihtiyaç sonucu oluşan değişime her zaman kapalıdır. Uygulamaların sahip olduğu özellikler çıkarılan ilk sürümden sonra değişime uğramaktadır. Bu değişimler mevcut yapıların değiştirilmesi ile sağlanabilir.
Open Closed Principle buna izin vermemektedir.
Bu durumda değişim veya yeni ek özellikler kazandırılmak istendiğinde mevcut yapıya dokunmadan yeni davranışlar kazandırılarak yapılar genişletilerek gelişimine devam edebilir.
Yeni kodlar yazılır fakat varolanlar değiştirilmez.

Bir değişim ve yazılım güncellemesinde mevcut yapıya dokunulmadığından

- Risk en alt seviyede tutulmuş olur.
- Programcılar değişime kapalı alanlardaki kod bloklarını anlamaya çalışmak ve değiştirmek yerine yeni yapılar kurabilirler.
- Mevcut yapılar değiştirilmediğinden test maliyetleri düşer.

Bir console application projesi içerisinde kartlı giriş çıkış işlemlerinin yapıldığı bir senaryo mevcut.



Sadece Personel Kart türünde işlem yapan uygulamaya Misafir kart türünden de giriş çıkış işlemleri yapılması ihtiyacı oluştu. Belki ileride başka kart tiplerinede ihtiyaç olabilecek.

```
public class PersonelKart
    {
        public void PersonelKartGiris()
        {
            Console.WriteLine("Personel Kart Giriş");
        }

        public void PersonelKartCikis()
        {
            Console.WriteLine("Personel Kart Çıkış");
        }
    }
```

Yeni kart tipi için oluşturulan sınıf yapısı MisafirKart

```
public class MisafirKart
    {
        public void MisafirKartGiris()
        {
            Console.WriteLine("Misafir Kart Giriş");
        }

        public void MisafirKartCikis()
        {
            Console.WriteLine("Misafir Kart Çıkış");
        }
    }
```

Mevcut uygulamamıza bu kart tipini tanıyacak şekilde değiştiriyoruz.

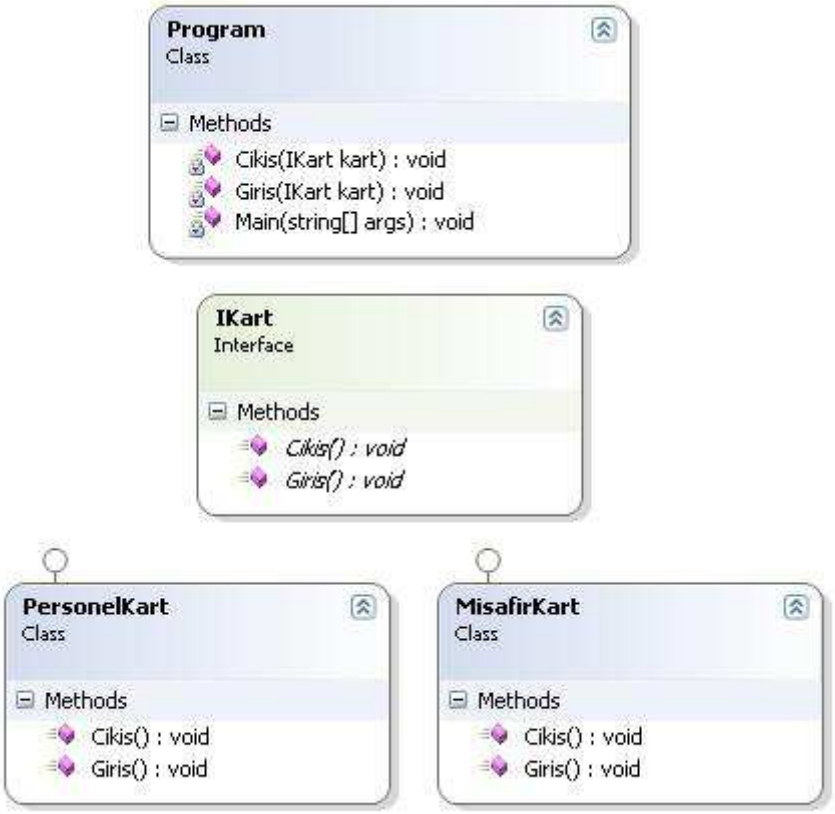
```
class Program
    {
        static void Main(string[] args)
        {
            PersonelKart pk = new PersonelKart();
            MisafirKart mk = new MisafirKart();

            Giris(pk);
            Giris(mk);

            Console.Read();
        }

        private static void Giris(object kart)
        {
            if (kart is PersonelKart)
                ((PersonelKart)kart).PersonelKartGiris();

            if (kart is MisafirKart)
                ((MisafirKart)kart).MisafirKartGiris();
        }


      private static void Cikis(object kart)
        {
            if (kart is PersonelKart)
                ((PersonelKart)kart).PersonelKartCikis();

            if (kart is MisafirKart)
```

```
                ((MisafirKart)kart).MisafirKartCikis();
        }
    }
```

Programı tasarlarken oluşturduğumuz Giris ve Cikis fonksiyonları yeni eklenen kart tipi için değişime uğradı. Bu tür bir operasyon Open Closed Principle ilkesine aykırıdır.

Her ne sebeple olursa olsun değişim olamaz.

Open Closed Principle ilkesine uygunluk için mevcut yapımızı aşağıdaki şekilde yeniden düzenliyoruz.




```
interface IKart
    {
        void Giris();
        void Cikis();
    }
```

```
public class PersonelKart:IKart
    {
        #region IKart Members

        public void Giris()
        {
            Console.WriteLine("Personel Kart Giriş");
        }
```

```
        public void Cikis()
        {
            Console.WriteLine("Personel Kart Çıkış");
        }

        #endregion
    }
```

```
public class MisafirKart:IKart
    {
        #region IKart Members

        public void Giris()
        {
            Console.WriteLine("Misafir Kart Giriş");
        }

        public void Cikis()
        {
            Console.WriteLine("Misafir Kart Çıkış");
        }

        #endregion
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            PersonelKart pk = new PersonelKart();
            MisafirKart mk = new MisafirKart();

            Giris(pk);
            Giris(mk);


            Cikis(pk);
            Cikis(mk);

            Console.Read();
        }

        private static void Giris(IKart kart)
        {
            kart.Giris();
        }

        private static void Cikis(IKart kart)
        {
            kart.Cikis();
        }
    }
```

Görüldüğü gibi Giris ve Cikis metotları bu ve bundan sonraki değişimlerden etkilenmeyecek şekilde düzenlendi. Yeni özellik eklenmesi genişletilmesi en az kod değişimi ile sağlanmış oldu.

Liskov Substitution Principle - LSP

Barbara Liskov tarafından geliştirilmiştir.

İlkenin açıklaması Alt sınıflar üst sınıfların yerini sorunsuz bir şekilde alabilmelidir.

- Alt sınıflardan oluşturulan nesneler üst sınıfların nesneleriyle tamamen yer değiştirebilmeli ve yer değiştirdiklerinde aynı davranışı göstermelidir.
- Üst sınıftan oluşturulmuş bir nesneyi kullanan bir fonksiyon, üst sınıf yerine bu üst sınıftan türemiş bir alt sınıftan oluşturulmuş bir nesneyi de aynı yerde kullanabilmelidir.
- Üst sınıf ve Alt sınıf arasında farklılık yoktur.

Üst sınıf ile alt sınıf nesne örnekleri yer değiştirdiklerinde, üst sınıf kullanıcısı alt sınıfın operasyonlarına erişmeye devam edebilmelidir.

Üst Sınıfı : Base Class

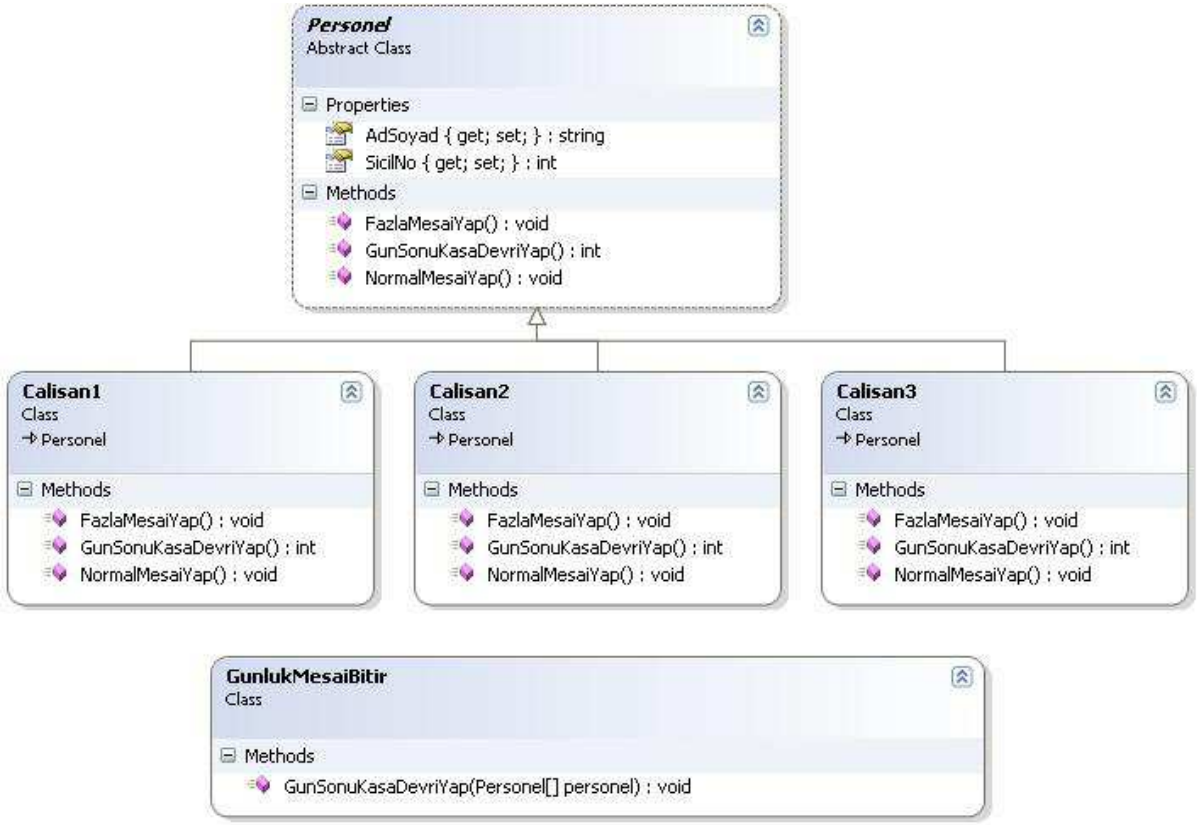
Alt Sınıf : Sub Class

Örnek

Bir bankada çalışan gurubu vardır. Her bir çalışan Personel abstract sınıfını şablon olarak kullanmaktadır. Çalışanlar gün içinde normal mesaisini varsa fazla mesaisini yapmaktadır. Ve gün sonunda kasalarını kapatarak devir yapmaktadır.

2 tip çalışan vardır. Calisan1 ve Calisan2 açık kasa tutarak para alma ve verme işlemlerini yapmaktadır.

Daha sonra yeni bir çalışan tipi ihtiyacı doğmuştur ve Calisan3 olarak adlandırılmıştır. Calisan3'de personel sınıfını referans almıştır. Normal ve Fazla mesai yapmaktadır. Fakat bir kasa sahibi değildir ve ödeme işlemleri yapmamaktadır.



Gün sonunda GunlukMesaiBitir sınıfı tüm calisanları dolaşarak GunSonuKasaDevriYap() metotlarını çağırmaktadır. Çağrılan bu metotlar kasadaki paraları int tipinde döndürmektedir.

```
foreach (Personel p in personel)
        {
           p.GunSonuKasaDevriYap();
        }
```

Bu durumda Calisan1 ve Calisan2 için sorun yoktur fakat Calisan3 bu metot çağrısına ödeme işlemlerinde yer almadığı için ne cevap verecektir.

Metot çağrıldığında bir exception dönderebilir.

```
public int GunSonuKasaDevriYap()
        {
            throw new System.NotImplementedException();
        }
```

Yada geriye return 0 dönderir. Eğer void tanımlanmışsa metod gövdesi boş bırakılabilir.

```
public int GunSonuKasaDevriYap()
        {
            return 0;
        }
```

Bu örnekte Calisan1,2,3 tümü aynı davranışı gösterememiştir. Örnekteki Alt sınıf, Üst sınıf ilişkileri Liskov ilkesine uymamaktadır. İlkeye uyması için Calisan3 ödeme işlemleri yapabilmeli ve GunSonuKasaDevri() alabilmeli 3 tipte birbirinin yerine geçebilmeli.
Bu şekilde kullanıcı istisnai durumlardan, geriye boş dönen metot sonuçlarından, alt sınıfların varlığından bile haberdar olmamalı. Tek bilmesi gereken tümünün yer değiştirme durumunda bile aynı davranışları göstermeli.

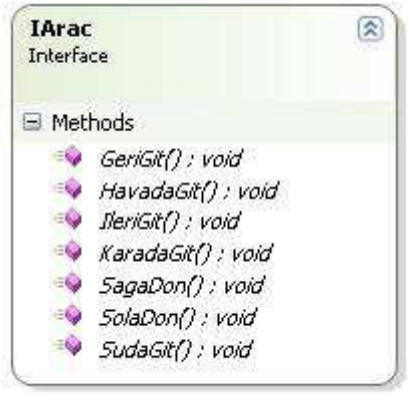
Interface Segregation Principle - ISP

Bir interface tasarlarken interface'in içereceği tüm elemanları bir interface içerisinde toplamak yerine bunları işlevlerine göre guruplandırmak ve farklı interface'ler içerisine dağıtmak olarak tanımlanabilir. Pek çok elemanı olan büyük interface'ler yerine, işlevsel ayrım yapılarak guruplanan birden fazla interface kullanmak ilkenin savunduğu düşüncedir.
Sınıflar şablon olarak kullandıkları Interface'in tüm özelliklerini uygulamak zorundadır.
Fakat kimi zaman interface üzerinde tanımlı yapıların tümüne ihtiyaç duymamakta kullanmamaktadır.

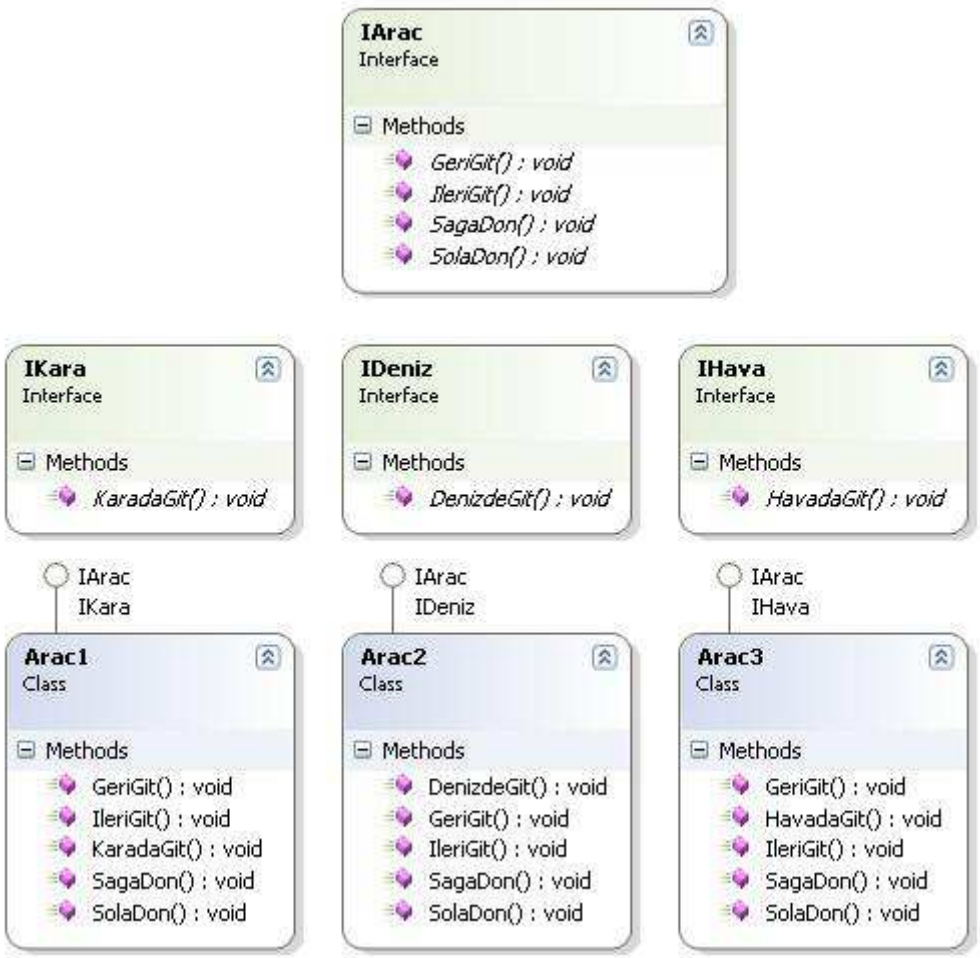
Sınıflar uyguladıkları interface'lerin tüm öğelerine ihtiyaç duymuyor, kullanmıyorsa bu Interface Segregation ikesine aykırıdır ve sınıflar bu inteface'leri uygulamamalıdır.
Bunun yerine interface'in elemanlarını işlevlerine göre ayırarark guruplandırmak ve bunları farklı interface'ler içerisine dağıtmak gerekiyor. Böylece sınıfların kullanmadıkları elemanları uygulama zorunluluğu kalkar ve bu elemanlar program içerisine eklenmemiş olur.

```
public interface IArac
    {
        void IleriGit();
        void GeriGit();
        void SagaDon();
        void SolaDon();

        void KaradaGit();
        void SudaGit();
        void HavadaGit();
    }
```

Örneğimizde IArac isimli bir Interface Arac1, Arac2 ve Arac3 sınıflarına uygulanmaktadır.
Arac1 kara, Arac2 deniz ve Arac3 hava taşıtıdır. 3 araç tipi içinde yön komutları ortaktır.
Fakat diğer elemanlar ortak değildir. Her araç kendi ortamında çalışır. Örneğin bir araba uçamaz.
Arac1 sınıfına bu interface uygulandığında SudaGit() ve HavadaGit() metotlarını içermek zorundadır.
Bu elemanlar kara taşıtı olan Arac1 için fazladan gereksiz şeylerdir.



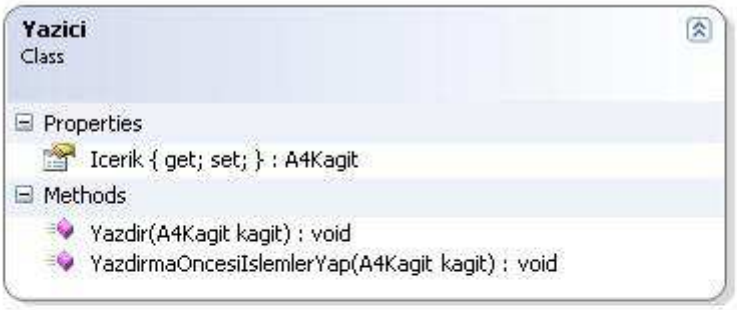
Bu yapıyı Interface Segregation ikesine uygun hale getirelim.

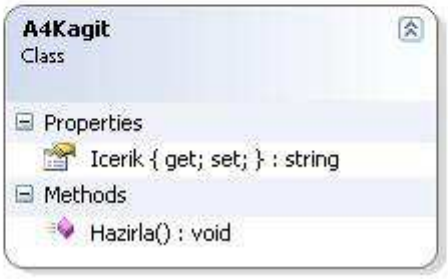


Dependency Inversion Principle - DIP

Alt sınıfılar ile üst sınıflar arasındaki bağımlılığın azaltılılarak
sınıflar arasında kuvvetli bağ oluşturulmasının engellenmesi gerektiğini belirten bir prensiptir. Sınıflar arasında doğrudan bir bağ kurulması alt sınıfılarda meydana gelebilecek değişim durumlarında bu değişimden üst sınıflarında etkilenmesine neden olacaktır. Bu tür bir senaryoda üst sınıflarında değişime uğraması gerekecektir. Üst seviye sınıflar bir zincir halkası olarak düşünülebilir. Alt seviyedeki değişimden etkilenen zincirin ilk halkasından son halkasına kadar tüm üst seviye sınıfların değiştirilmesi gerekebilir. Bu tür zincirleme tasarım mantıksal olarak bir bölmleme/modülleştirme olarak görünsede aralarında kurulan kuvvetli bağ nedeniyle sınıflar diğer sınıflara bağımlı olduğundan diğer sınıfılar olmadan tek başlarına çalıştırılamayacaktır. Bunun için bağımlılık tersine çevrilmeli ve üst sınıflar alt sınıflara bağımlı olmamalı alt sınıflar üst sınıfların arayüzlerine bağımlı olmalıdır. Yapılacak olan şey somut yapıların kaldırarak üst sınıf ile alt sınıf arasına soyut arayüz (Interface) eklemektir.

Örneğimizde Yazici sınıfı A4 sınıfına bağımlıdır. Herhangi bir değişim yada genişleme durumu bu yapı için uygun değildir.





Bu yapıyı Dependency Inversion Principle ilkesine nasıl uygun hale getirebiliriz onu inceleyelim.

Görüldüğü gibi Yazıcı sınıfı herhangi bir tipe doğrudan bağımlı değildir. Alt sınıflar A3,A4,A5 Üst sınıf Yazicin arayüzüne IKagit'a bağımlıdır. Her koşulda arayüzün uygulanması gerektiği garanti altına alındığından alt sınıflarda meydana gelecek bir değişim üst sınıfı etkilemeyecektir.

- SON -