**VERİ**

metin, ekran, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

Bu şekilde bir veri olduğunu düşünelim yeni bir kurs ekleme işleminde hem değişken tanımlama, hem de yazdırma işlemi yapılması gerekmektedir. Büyük projelerde birden fazla veri ekleme işlemlerinde zaman kaybı olur. Bunun yerine Dizi(ARRAY) yapısı kullanılır.

**DİZİLER/ARRAYS**

Dizi yapısı şu şekildedir ;

metin, iç mekan, ekran, portakal içeren bir resim

Açıklama otomatik olarak oluşturuldu

Dizilerden sonra döngü kullanılır. Döngü de dizi uzunluğu kadar istenileni yazdırır.

**FOR DÖNGÜSÜ FOREACH DÖNGÜSÜ**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu  
  
Foreach dizilerde tek tek dolaşmayı sağlayan döngü yapısıdır. kurs her bir elemanı dolaşırken o elemanı tutan, takma isimdir.(Alias)

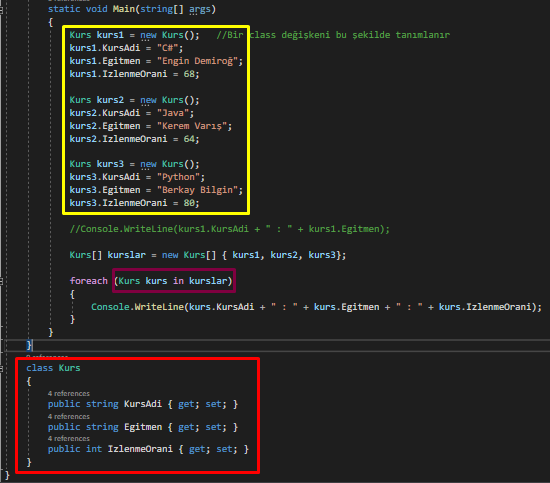
**SINIF/CLASS**

Birden fazla veri olduğunda karmaşıklığı önlemek adına ne yaparız ? Class yapısını kullanırız.   
Classlar içerisinde birden fazla veri tutan yapılardır. Nasıl string, nasıl int gibi değişken tanımlanabiliyorsa , Kurs diye değişkende tanımlanabilir. (Kendi değişken tipimiz gibi düşünelim) C# programlama dili Kurs adında veri tipini bilmediği, tanımadığı için (class Kurs{ }) adında tanımlama yapılması lazım. Sanki kendi veri tipini oluşturuyormuşsun gibi düşün.





Bir class değişkeni bu şekilde tanımlanır. Class oluştururken onun bir örneğini oluşturur gibi çağırırız. PascalCase (Kurs) , onun örneği(instance) oluşturulurken camelcase (kurs1)



**Kırmızı Kutucuk içindeki ;**  
Özellikleri (property) oluştururuz. Bir class oluşturmak için yapılacak hareket bu

**Sarı Kutucuk içindeki ;**  
Class değişkenini oluşturmak ve değerlerini atamak için yapılacak hareket bu.

**Mor Kutucuk içindeki ;**Kurs : Oluşturduğumuz veri tipi  
kurs : Her bir elemanı dolaşırken o elemana karşılık gelen takma isim (Alias, değer tutucu)

**METOTLAR/METHODS**

Bir e-ticaret sitesinde ürünlerin olduğu, siparişleri onayladığımız bir sepet kısmı var. Bu sepet anasayfa da var, ürüne gidersin orada gözükür ; yani sepet birden fazla sayfada karşımıza çıkar. Her sayfa için sepet fonksiyonu yazılmaz. Çünkü o fonksiyon için bir özellik eklemek, değiştirmek veya silmek istendiğinde tek tek her sayfaya giderek o işlemi yapmak zaman kaybıdır. Bunun için metotları kullanırız. Metotlar aynı kod bloklarının farklı yerlerde kullanılmasını sağlamaktadır. Program içerisinde bir işi, bir görevi yapmak için yazılmış kod bloklarıdır. Metotlar çağrılırken parametreleri eksiksiz ve doğru sıra ile girilmelidir.

metin, ekran, ekran görüntüsü, kapat içeren bir resim

Açıklama otomatik olarak oluşturuldu

Program.cs senin menündür. Yazdırmak istenileni yazdırırsın. Görünümdür. Ekranda o çıkar.

Product.cs Varlık(Entity) burada classını tanımlarsın. Özelliklerini verirsin. Varlık yaratırsın.

SepetManager.cs burası iş yapan kısımdır. Metotları burada tanımlarsın. Menüde de bu tanımlamış olduğun metotları çağırırsın.

ProductManager, ProductService gibi isimler olduğu takdirde ürün class’ı için anla ki o ürünle ilgili operasyonlar var. Ekleme, arama, silme, listeleme, güncelleme vs.

**Product.cs :** Ürün isimli classımız olsun. Burada özellikleri oluştur. Class oluşumu. Varlık(entity)

metin içeren bir resim

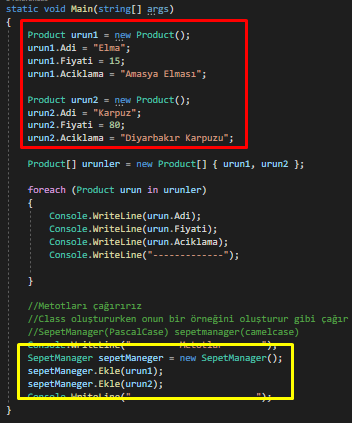
Açıklama otomatik olarak oluşturuldu

**SepetManager.cs :** Manager classı. İş yapan kısımdır. Ürünü ekleme işlemi yapalım. Ekle metodu sadece ekle diyerek neyi ekleyecek ? Metoda eklemek istenileni vermen lazım. Buna parametre denir. Parametrede tipini ve işlem yapacak ismi yazarız. Metotta parantez içinde yazılır. Ürün ekleme işlemi yapacağız Product onun tipi urun ise isimlendirmedir. Ekle ve Ekle2 metotları aslında aynıdır. Değişken ve isimlendirme nasıl olduğuna bakınız.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Program.cs :** Menü. Yazdırılmak istenilen yazdırılır. Metotlar çağırılır.



**Kırmızı Kutucuk içindeki ;**  
Oluşturulan classın değişkenini oluştur (Product) ve değerlerini ata. Dizisini ve döngüsünü oluştur.

**Sarı Kutucuk içindeki ;**Tanımlamış olduğumuz metotları çağırırız.

1. Sayfa = sepetManager.Ekle2("Armut", "Yeşil Armut", 12, 45);
2. Sayfa = sepetManager.Ekle2("Elma", "Yeşil Elma", 10, 78);

3. Sayfa = sepetManager.Ekle2("Karpuz", "Yeşil Karpuz", 20, 69);

Bunlar başka sayfalar olsun diyelim ki sen stok adedi diye bir özellik daha ekledin. Bütün sayfalara tek tek gidip o özelliği yazmak zorundasın.

Classta ise özellik olarak bir kere tanımlarsın her sayfada o gözükür.

SepetManager.cs Ekle2 metodunda stokAdedi ekledin varsay tüm sayfalara onu eklemek zorundasın.

Bunun yerine Product.cs classa parametreyle eklediğin zaman özellik olarak her yere ekler.

Buna Encapsulation(Kapsülleme) denir. Özellikleri ayrı ayrı eklemek yerine kapsülleme. Classta bir özellik ekleyip, diğer sayfalarda özelliğin çağrılması.

**DEĞER VE REFERANS TİPLER**

Değer Tip : int, decimal, float, double, bool vs.

Referans Tip : Array, Class, Abstract, İnterface vs.

Bellekte iki tane yerimiz var. Stack ve Heap.

Değer tipler sadece Stack’te çalışır.

Değer tiplerde değer atıyorsun, referans tiplerde adres atıyorsun.

Örneğin sayi1 = ? Örneğin sayilar1[0] = ?

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Stack Heap Stack Heap   
--------- --------- ---------- ---------  
sayi1 = 10 Değer tip sayilar1 [10,20,30]  
sayi2 = 30 burası boş, sayilar2 [100,200,300]  
sayi1 = sayi2 çalışmaz. sayilar2 [999,200,300]   
sayi2 = 65   
sayi1’in değeri 10, sayilar1’in adresi 101  
sayi2’nin değeri 30, sayilar2’in adresi 102  
sayi1 eşittir sayi2, sayi1 = 30 sayilar1 = sayilar2 ; sayilar1’in adresi 102   
sayi2’nin değeri 65. sayilar2[0] = 999 ; sayilar1[0] = 999**



sayilar1 dizisinde ; sayilar1 i stackte başlattın new dediği anda sayilar1’i değerler dizisi heap’ e atıyor. New demek bellekte bir adres oluştur, heap’te bir adres oluştur demektir.

**Return - Void Mantığı**

Diyelimki Manager sınıfında int tipli Topla isminde bir fonksiyon yazdık. sayi1, sayi2 toplama işlemi yap dedik. Menüde çağırdığımızda toplamaSonucu adında değişken tanımlayıp fonksiyonumuzu burada kullanarak, sayıların toplamını yazdırırız. Ama sonradan bu sayıları bir de 2 ile çarpmak istiyoruz. İşte o zaman toplamaSonucu değişkenini tekrar kullanarak 2 ile çarpma işlemi yapabiliriz. Return mantığı ;sayi1, sayi2 return(döndür) diyoruz yani kodda tekrar kullanabiliriz. Return de devam özelliği vardır. Topla fonksiyonunu 2. kez kullanılır hale getirdik.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Bir de Manager sınıfında void tipli Topla2 isminde bir fonksiyon yazdık. Menüde Topla2 fonksiyonu için değer verip çalıştırdık. Sonradan bu sayıları 2 ile çarpmak isteyelim, çarpamayız herhangi bir işlem yapamayız. Çünkü void emir kipi gibi düşün, git, yap ; gitti, yaptı işi biter tekrar kullanamayız.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**OOP(Object Oriented Programming – Nesne Yönelimli Programlama )**

**OOP1**

Classları ikiye ayırıyoruz. Kullanım yöntemi olarak.

1. İçerisinde özellik barındıran classlar.
2. İçerisinde operasyon yani metot barındıran classlar.

**Product.cs :** Ürün classım. Özellikleri oluştur. Varlık(Entity)

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Program.cs :** Classın değişkenini oluştur(Product) ve özelliklere değer ata. Okla gösterildiği gibi 2 şekilde de yazılabilir. product1 ve product2 Product’ un örneği ( instance). Ürünler dizisi oluştur ve döngüsünü yaz. Döngü yazmamızın sebebi o ürünleri tek tek dolaşsın. Burada 2 ürün var şimdilik ama birden fazla veri olduğunda ekrana ürünlerin tüm özelliklerini yazdırılmak istendiğinde her veri için tek tek yazdırılır. Ama döngüde 1 kere tanımlarsın her verinin sana tüm özelliklerini yazdırır. (product.UnitPrice = hem product1, hem product2‘nin birim fiyatını getirir.) Metotları çağır. Product classı gibi ProductManager classını da çağır. Class oluştururken onun bir örneğini oluşturuyormuş gibi çağır. Metottan sonra parantez içine hangi veri için olduğunu belirt.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**ProductManager.cs :** İş yapan class. Metotlarını oluştur. Ekleme, güncelleme işlemi için parametre ver.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**KOLEKSİYONLAR/COLLECTİONS**

Koleksiyonlarda arrayler gibi değerleri kaybetmemizi engelleyecek bir altyapı var.

İsimler adında bir dizi oluşturduk ve çağırdık.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Sonradan bir eleman daha eklemek isteyelim. Ekleme işlemi için 4 elemanlı diziyi 5 elemanlı yapmamız gerek. Newledikten sonra da 4. İndise İlker ismi verdik, isimler dizimizin 4. ve 0. indisini ekrana basalım.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Bu diziyi çalıştırdığımızda İlker ismi ekrana gelir ama dizinin 0. indisi yani 1. elemanı gelmez. Çünkü new diyerek yeni bir dizi tanımladık 5 elemanlı ; 4. elemana yani 3. indise İlker dedik ama 0. indis yani 1,2,4 elemanları boş.

**Stack Heap**

**---------- ---------**

**isimler [“Betül”, “Serap”, “Ada”, “Ozan”]**

**new [“boş”, “boş”, “boş”, “boş”, “İlker”]**

**isimler dizisi adresi : 101 new dedik**

**isimler dizisi adresi : 102 olur**

Bu örnekte diziye İlker ismini eklemememizin sebebi büyük projelerde bunlar veri olarak alındığı için sonradan ekleme gibi bir durum olamaz. Bu yüzden dizilerle çalışamıyoruz bu tarz kısımlarda. Bunun yerine koleksiyonları kullanırız. Koleksiyon ; aynı veri türünde(isimler) belirli bir sayıda, o sayıda belirsizse bir veri grubu oluşturmaya yarayan bir yapı. List yapısı (List<istediğin veri tipi>) kullanılır. List yapısının Add, Count gibi birçok özelliği vardır. List’in içine hangi veri tipini(String) yazdırırsan özelliğinde(ADD) de onu çağırırsın.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Üst dizinin sonucu List’in sonucu**

**------------------------ ---------------------**

**Betül Betül**

**Serap Serap**

**Ada Ada**

**Ozan Ozan**

**İlker İlker**

**Boş Betül**

**JENERİKLER/JENERİCS**

Jenerikler tasarladığımız interface, class, metod ya da parametrelerin(argümanların) belirli bir tip için değil bir şablon yapısına uyan her tip için çalışmasını sağlayan bir yapıdır. Tiplere genelde T denir. T ne istersen o olur, type için. < > içerisine tip yazılır. string, int, senin verdiğin isim vs.



List koleksiyonlarda içinde hangi tipi yazdırırsan ADD metodu içerisinde onu çağırıyordu ya burada da MyList adını verdiğim List’e tipini ne istersem List işlemlerinde o yazsın istiyorum o zaman MyList<T> deriz T burada isteğimiz tipi ifade eder. Yani ben sana t olarak ne verirsem istediğim elemanın türü de o olur.

**MyList.cs :** MyList<T> isimli jenerik class oluşturalım. İtem eleman demek. T[] items ifadesi bir referans tiptir tek başına bir anlam ifade etmiyor sadece değişkeni oluşturdu. Add fonksiyonu içerisine elemanları eklemek istiyorum ama eklemek için bir arrrayin newlenme zorunluluğu var. Array referans tip olduğu için newlenme zorunluluğu var. Bir class newlendiğinde çalışan bloğa **“Constructor”** denir. Void filan döndürmez. Class’ı newlersen constructor otomatik çalışır. Kırmızı kutucuk constructordır.

Ben bu elemanlar dizisini başlandığında 0 eleman olarak newlemek istiyorum.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Dizimiz şuan 0 elemanlı. Eleman sayısını bir arttırmak istiyoruz; koleksiyondan hatırla dizinin eleman sayısını arttırınca newlemek zorundasın.

items = new T[1] bu şekilde bir ifade yazarsak eleman sayısı hep 1 olur. O zaman ekleme işleminde newledikçe eleman uzunluğu kadar 1 arttırsın. items = new T[items.Length +1]

Diyelimki 1 eleman ekledik “Engin” sonra gitti eleman sayısını 1 arttırınca 2 eleman oldu ama newleme işlemi yaptığımız için “Engin” gitti. Giden referans numarası(adres). O yüzden referans numarasını birine tutturmamız lazım geciciDizi = tempArray

**Stack Heap**

**-------- -------**

**İtems [ ] İtems’ in adresi 101 eleman sayısı 0.**

**New [“Engin”] Eleman ekledik new dedik adresi 102 Engin gider.**



**New [ , , ] Bir eleman daha ekledik new dedik adresi 103 yine boş olur. tempArray [“Engin”]**

**Adresi 101 olduğunda tempArray kullansaydık engin elemanını eklemek için newlediğimizde tempArray adresi 102 [“Engin”] elemanını tutardı.**

metin, ekran görüntüsü, ekran, siyah içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Sarı çizgi ;**

Elemanları tempArray’ e atama. geçiciDiziye elemanları tutturma işlemi. Eleman uzunluğunu arttırdıkça newleme işlemi yapılır eski elemanları da tempArray geçici olarak tutsun.

**Lacivert çizgi ;**

İşlem bitti eleman eklemeyeceğiz daha o zaman tempArray’in bizim yerimize adresinde tuttuğu elemanları geri alalım. Yani tempArray’in bütün elemanlarını items’a geri atamalıyız. Elemanları geri alma işlemi yapılır burada 0’dan tempArray’in Length’i kadar geri al for döngüsü.

**Pembe çizgi ;**

tempArray de 5 kutu var diyelim. items = new T[items.Length +1] bloğundan sonra bir eleman daha ekledik. Yeni gelen eleman new yazan kutucuk olur. İtems artık 6 elemanlı bir kutu olur ama ilk 5 elemanı boş olur for döngüsü ile de tempArray den elemanları geri almalıyız. Pembe çizgili kısmı yazarak eklenmek istenilen elemanı ancak o zaman ekleriz. Şöyle ;

items 6 eleman 1 eksiği 5.indeks = gelen eleman,newlenen

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| B | S | E | A | O |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| B | S | E | A | O | NEW |

**Program.cs :** T yerine istediğimiz veri tipini yazabiliyorduk ne istersek o oluyordu. Burada string olarak kullandık. Metotta da doğal olarak string ifadesi yazdık.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**OOP2  
S.O.L.I.D**

**S – Single-responsibility Principle**

Bir sınıf(nesne) yalnızca bir amaç uğruna değiştirilebilir, o da o sınıfa yüklenen sorumluluktur. Yani bir sınıfın(fonksiyona da indirgenebilir) yapması gereken yalnızca bir işi olması gerekir.

**O – Open-closed Principle**  
Bir sınıf ya da fonksiyon hali hazırda var olan özellikleri korumalı ve değişikliğe izin vermemelidir. Yani davranışı değiştirmiyor ve yeni özellikler kazanabiliyor olmalıdır.

**L – Liskov-subsititution Principle**

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

**I – Interface segregation Principle**

Sorumlulukların hepsini tek bir arayüze toplamak yerine daha özelleştirilmiş birden fazla arayüz oluşturmalıyız.   
  
**D – Dependency Inversion Principle**

Sınıflar arası bağımlılıklar olabildiğince az olmalıdır özellikle üst seviye sınıflar alt seviye sınıflara bağımlı olmamalıdır.

Bir senaryo yazalım. Normal bir müşteri sınıfımız olsun. Özelliklerini tanımlayalım.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu  
Burada CompanyName(ŞirketAdı) yoktur çünkü bu müşteri şahıstır, bir şirket değildir. Eğer bir nesnede bir değeri kullanmak zorunda değilsen, o nesneye ait gibi duruyorsa o an demek ki orada soyutlama  
hatası yapıyorsun demektir. Gerçek(bireysel) müşteri(Individual Customer) ve Tüzel müşteri(Corporate Customer) 2 müşteriymiş gibi düşün birbirinin aynısı değildir birbiri yerine kullanılamaz. MusteriTipi diye bir şey tanımlayalım mesela 1 ise gerçek müşteri,0 ise tüzel müşteri olsun.(customer1.MusteriTipi = 0)Bu hatalı bir tanım olur çünkü müşteri gerçek bir müşteri gibi görünen ama gerçekte tüzel müşteri gibi olan hatalı veri girişinden kaynaklı datalarla çok karşılaşılır. Bu durumda yazılım geliştirme prensiplerinden SOLID’in L tekniği uygulanır. Ne demek istiyoruz? Bir müşteri de olan her şey tüzel müşteride de, gerçek müşteride de vardır. Müşteri ikisinin ebeveynidir. Müşteri classımız olur ama bazı özellikleri tüzel müşteriye, bazı özellikleri gerçek müşteriye, bazı özellikleri ise her ikisine de ait olur. Ortak özellikler bir classa yazılır diğerleri de ondan miras alır(inheritance).

**Customer.cs :** Müşteri class(ana classım). Tüzel ve gerçek iki müşterininde id ve müşteri numarası ortak bir özelliktir bu özellikleri customer classıma yazarsam tüzel müşteriye de gerçek müşteriye de bu özellikleri yazmam gerekmez çünkü customer classında bu özellikler ve bunlar da miras alıyor olacak. Customer classı base class yani temel sınıf demek. Eğer senin base sınıfın varsa o base sınıf referans tutucudur. Varlık(Entity), müşteri nesnesi.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**IndividualCustomer.cs :** Bireysel müşteri sınıfı. Customerdan miras alır. Varlık, gerçek müşteri nesnesi.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**CorporateCustomer.cs :** Tüzel müşteri sınıfı. Customerdan miras alır. Varlık, tüzel müşteri nesnesi.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**CustomerManager.cs :** İş sınıfı(business class). Metotları burada yazarız.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Program.cs :** Gerçek müşteri,tüzel müşteri müşteri sınıfını miras almış ve customer sınıfında olan özellikler tüzel müşteride de gerçek müşteride de yer alır. Gerçek, tüzel müşteri sınıflarının özelliklerine değerleri ata. İş classını newle ve metotları çağır.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**KALITIM,MİRAS/INHERITANCE**

OOP(Nesne Yönelimli Programlama) prensiplerinden birisidir. OOP’ da 3 temel prensip vardır.

1. Encapculation – Kapsülleme  
2. Pollyphormosicm – Çok Biçimlilik  
3. Inheritance – Kalıtım

Kalıtım, diğer sınıflarda tanımlanan davranışı yeniden kullanan, genişleten ve değiştiren yeni sınıflar oluşturmamızı sağlar. Yani kısacası, bir sınıfı miras alarak, onun metotlarını kullanan başka sınıflar türetebilmemizi sağlayan bir kavramdır.

Sınıflar arası hiyerarşik yapı kurabilmek, oluşturmak için kullanılır. Bir sınıf, başka bir sınıftan türeyerek(kalıtım alma), o sınıfın public ve protected tanımlı yapılarını devralır.

-Kalıtım alan sınıf : Devired Class(Türetilmiş/Türeyen Sınıf)  
-Kalıtım veren sınıf : Base Class(Temel/Türeten Sınıf)

Bir sınıf sadece tek bir sınıftan kalıtım yapabilir. Aynı anda birden fazla sınıftan kalıtım alamaz.   
Bunun için interface yapısı kullanılır. Türeten sınıf(base class), türeyen sınıf(devired class) için base olarak kabul edilir. Yani her sınıfın basesi kendi türediği sınıftır.

**ARAYÜZ,SOYUTLAMA/INTERFACE**

Bir arayüz, ilgili işlevselliklerin bildirimlerini içerir(Sözleşme gibi düşün). Arayüzü uygulayan varlıklar, bildirilen işlevlerin uygulanmasını sağlamalıdır. Bir projenin bir classındaki kullanılması gereken metotları bize söyleyen ve onları otomatik oluşturan yapı. Bir sınıfa interface ataması yaparak, o sınıfa özel metotlar üretip kullanabiliriz.

Bir sınıfa birden fazla Interface implement uygulayabiliriz. Interface içerisine metot tanımlaması yaparız ama işlemleri yazmayız. Yapılmasını istediğimiz işlemleri, Interface uyguladığımız sınıf içerisine yazarız. Eğer bir sınıfa Interface uygulanmışsa yani implement edilmişse, o Interface içindeki metotlar mutlaka sınıfta da uygulanmalı yani implement edilmek zorundadır.

Interface’in yaptığı iş sadece diğer classlara yön göstermektir. Bu yüzden Interface tanımları içerisinde kesinlikle kod bloğumuz bulunmaz. Interface yapılar sadece tanımlamalardan ibarettir, içerisinde hiçbir yapıcı metot ekleyemeyiz.

Bir class sadece bir classstan türetebilirken, bir class birden çok Interface’ten türetilebilir.

Bir Interface, başka bir interface’ten türeyebilir ama bir Class’tan türeyemez.

Interfaceleri miras alan sınıflar, içerisinde tanımlanan tüm metotları Implemente etmek zorundadır.

Interfaceleri, birbirinin alternatifi olan ama kod içerikleri farklı olan durumlar için kullanırız.

Okunurluğu arttırmak için interfaceleri I harfiyle başlatırız. I interface demektir.

**OOP3**

Müşteri için kredi hesaplama, başvuru yapma, veritabanına kaydetme(loglama) senaryosu yazalım.

Binlerce kredi çeşidi olduğunu düşün 300 tane mesela her kredi için faiz oranı, dosya masrafı, farklı farklı özellikleri vardır ve birbirinden farklıdır. Sen hesapla metodu için kredi1 buysa şu işlemi yap, kredi2 buysa şu işlemi yap gibi if veya switch-case bloklarını kullanabilirsin. Ama 300 kredi için bu kod kalabalığıdır. Interface yaparak bu işi kolaylaştırırız.

Bir interface oluşturduğumuzda o interface’in içerisindeki operasyonlar alternatif sistemler için bir şablon ve buna ek olarak referans tutucu görevi görüyor.

**ICreditManager.cs :** Manager, Service Business Classtır.(İş sınıfı) Kredi hesaplaması yapacak bir Interface sınıfı oluşturalım. Neden Interface ? Hesapla metodu taşıt kredisinde de, ihtiyaç kredisinde de, konut kredisinde de hesaplama işlemi yapılır hepsinin içeriği faiz oranı vs. farklıdır. Bu durumda imzanın aynı olduğu(hesapla metodu) ama içerisinin farklı olduğu durumlarda base de oluşturulan classı(CreditManager) class olarak değil, Interface(ICreditManager) olarak oluştururuz.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

3 kredimiz olsun. Taşıt, İhtiyaç, Konut kredileri. Bu kredi sınıfları base sınıflarını (ICreditManager) miras alır. Çünkü onlarda kredi aynı nesne. Eğer ki bir sınıf bir interface’i (:) böyle kullanıyorsa benim ICreditManager kurallarıma uymak zorundasın. O yüzden her kredi (taşıt, ihtiyaç, konut) ICreditManager interfacesindeki tüm metotları içermek zorunda.

**TransportCreditManager.cs :** Taşıt Kredisi iş sınıfı. Metotları taşır, kendi metodu varsa buraya yazılır.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**PersonalCreditManager.cs :** İhtiyaç Kredisi iş sınıfı. Metotları taşır, kendi metodu varsa buraya yazılır.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**MortgageLoanManager.cs :** Konut Kredisi iş sınıfı. Metotları taşır, kendi metodu varsa buraya yazılır.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**ILoggerService.cs :** İş sınıfı. Loglama işlemi yapacak bir interface oluşturalım. Interfacelere Service demeye çalış.

metin içeren bir resim

Açıklama otomatik olarak oluşturulduEğer ki bir sınıf bir interface’i (:) böyle kullanıyorsa oluşturduğumuz ILoggerService kurallarıma uymak zorundasın. Bu yüzden her log (veritabanı, dosya vs.) ILoggerService interfacesindeki her metodu içermek zorunda.

**DatabaseLoggerService.cs :** Veritabanı loglama iş sınıfı. Metotları taşır, kendi metodu varsa buraya yazılır.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**FileLoggerService.cs :** Dosya loglama iş sınıfı. Metotları taşır, kendi metodu varsa buraya yazılır.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**ApplicationManager.cs :** İş sınıfı. Krediye başvuru işlemlerini yapacağımız class. Application : Başvuru demek.  
Burada krediye başvuru yapacağımız için, tek bir kredi istiyoruz. Bir tane BasvuruYap metodu oluşturalım ve içerisine Method Injection işlemini yapalım. Method Injection demek bu metotun (BasvuruYap) kullanacağı CreditManager’ ın ne olacağı yani hangi kredi türü olacağını ve hangi logla olacağını enjekte ediyoruz.

metin, ekran görüntüsü, ekran içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Kırmızı kutucuk içindeki ;** BasvuruYap metodumuz için neye göre başvuru yapacak ? Kredi ve log servisine göre bunlar parametre metota bunu vermen lazım. Peki neden Interface olarak çağırdık ? Diyelim ki BasvuruYap metodu içerisine taşıt kredisini yazdırdık; o zaman tüm başvuru yapanlar taşıt kredisi üzerinden değerlendirilir. Ama BasvuruYap metodu içerisine interfacemizi çağırıp kredi gibi bir parametre verirsek tüm krediler için hangisi seçilirse onun için işlem yapar bu metot. Yani binevi krediyi bağımsız hale getirdik. Aynı şekilde log servisini de.

**Turuncu ok ;** BasvuruYap metodu (Add,Delete,Update gibi metot) Metot içerisine de yapılacak işlemleri yazdırırdık mesela şu eklendi, bu silindi gibi burada da BasvuruYap metodunu Kredi hesaplaması ve loglamaya göre yaparız. Hangi kredi(creditmanager) gönderilmişse onu hesapla, hangi loggerservice gönderilmişse onu logla demek isteriz.

Mesela sonradan farklı bir kredi(Esnaf kredisi) de sisteme eklenebilir ya da farklı bir loglama servisi (Sms) sisteme eklenebilir. Çünkü interfaceler, metotlar bunu bize sağlıyor. Taşıt,ihtiyaç,konut kredisi gibi ICreditManager classından miras alarak bende bir krediyim dersin ya da Veritabanı loglama, dosya loglama için bende bir loglama işlemiyim sms’im dersin ILoggerService’ ten miras alırsın.

**Yeşil kutucuk içindeki ;** Senaryoya devam ediyoruz bir kredi ön bilgilendirmesi istiyoruz burada birden fazla kredinin hesaplanmasını yapmak istiyoruz. Birden fazla kredi(sayısını bilmiyoruz) hepsinin türü kredi böyle durumlarda hatırla Aynı veri türünde belirli sayıda, o sayıda belirsizse bir veri grubu oluşturmak için Koleksiyonları (List<istediğin veri tipi>) kullanırız. Metot(veri tipi, isim)

Listedeki her bir kredinin hesaplanmasını yap, sistem krediler listesini tek tek gezip hesaplayacak bu yüzden foreach döngüsü kullandık. Hatırla, foreach döngüsü elemanları tek tek dolaşmaya yarardı.

Yapılan işlem şu ; BasvuruYap metodunda taşıt kredisini gönderirsek bellekte sadece taşıt kredisinin hesaplaması çalışır. Başvuru yapılıyor sadece bir krediye başvurulabilir. Burada tek bir kredi var onu çağırıyor. Ama KrediOnBilgilendirmesiYap metodunda bir liste var her bir krediyi dolaş ve hesapları onları bizi bilgilendir diyoruz.

Diyelim ki bir kredi için hem veritabanına loglansın, hem dosyaya loglansın. KrediOnBilgilendirmesiYap metodunda nasıl bize istediğimiz kredileri seçmemizi, hesaplamalarını karşılaştırmamızı sağlıyor ise , Listeler sayesinde bir metot daha oluşturarak bu metot içerisine liste oluşturup, menü de onu çağırırız ve bu şekilde istediklerimizi yazdırabiliriz. Mesela ;

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Program.cs :** Menü, ekranda yazılacakları gösterir. İhtiyaç, taşıt, konut İş sınıflarını çağırdık newledik. Bu sınıflarda Calculate() ve BiSeyYap() metodları vardı ,normalde iş sınıfını newlersin, yazdırmak istediği metodu çağırırsın, nesnen varsa paranteze yazdırırsın. Yorum satırı bunları anlatıyor. Biz başvuru yapacağız tek kredi hesaplaması yapmamız gerekir. Yine aynı şey ApplicationManager da senin iş sınıfın onu da çağır newle BasvuruYap ve KrediOnBilgilendirmesiYap metotlarını çağır (Hangi işlemi yapmak istiyorsan o metodu çağır, biz hem kredilerin hesaplanıp karşılaştırılmasını(KrediOnBilgilendirmesiYap) hem de kredi başvurusu yapmak istiyoruz o yüzden iki metodu da çağırdık.)

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

BaşvuruYap metodu 2 parametre alıyordu hem kredi, hem loglama. Kredi iş sınıflarını çağırıp newledik ama loglama iş sınıflarını çağırıp newlemedik. Pembe ok ; loglama iş sınıflarını da newlediğimiz ve applicationmanager.BasvuruYap metodunu yorum satırında bizlere nasıl çağıracağımızı gösteriyor. Newlemezsen de Pembe çizgiyle ; metot parantezinde nasıl newlendiğini gösteriyor. İkisi de kullanılabilir. BasvuruYap metodu konut kredisine göre hesaplama ve database loglama işlemi yaptı.

KrediOnBilgilendirmesiYap metodu ihtiyaç ve taşıt kredilerini hesaplayıp bize bir ön bilgilendirme sundu.

Bu da örnek verdiğimiz hem veritabanına, hem de dosyaya loglama işlemi yapan bilgilendirme metodumuz.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**ABSTRACT**

Tamamen kalıtım amaçlı kullanacağımız temel sınıfları oluştururken Abstract anahtar sözcüğünü kullanırız.

Abstract Class, ortak özellikli Class’lara Base(taban) Class olma görevini üstlenir.

Yani Abstract Classlar, diğer sınıflara base class olmak için yazılır. Bu nedenle Abstract Class’tan nesne türetilemez.

Abstract Method, sadece Abstract Classların içerisinde tanımlanır ve Abstract Class’ ı kalıtan sınıf tarafından override edilmek zorundadır.

Abstract Method, sadece Abstract Classların içerisinde tanımlanabilen, tanımlanırken gövde bulundurmayan, daha sonra içerisinde bulunduğu Abstract Class kalıtılırken override edilen Methodlardır.

Abstract Methodlar, Private olarak tanımlanamaz.

Abstract sınıf içerisinde bildirimi yapılmış olan metotları türetilmiş sınıflar içerisinde tanımlayabilmemiz için override anahtar sözcüğünü kullanmamız gerekmektedir.

**Abstract Class, Normal Sınıf Farkı ?**

Normal sınıflar içerisinde metot bildirimi yapılamazken , Abstract sınıflar içerisinde tıpkı interfaceler de olduğu gibi metot bildirimi yapılabilir.

Normal sınıflarda “new( )” anahtar sözcüğü ile nesneler oluşturulabilir ancak Abstract sınıflar tamamen kalıtım amaçlı geliştirildiğinden Abstract sınıflardan nesne oluşturulamaz.

**Abstract Class, Arayüz(İnterface) Farkı ?**

Arayüzlerde sadece metot bildirimi yapılabilirken Abstract sınıflarda hem metot bildirimi yapılabilir, hem de metot tanımlanabilir.

Bir sınıfa sadece bir tane Abstract sınıf inherit edilebilir ancak aynı sınıfa birden fazla Arayüz implement edilebilir.

**Virtual Tanımı**

Eğer bir metot tüm türetilmiş sınıflarda aynı şekilde tanımlanıyorsa o metodu abstract sınıf içerisinde standart bir metot tanımlar gibi tanımlarız.

Eğer bir metot birçok türetilmiş sınıfta aynı şekilde tanımlanıyor iken sadece birkaçında değişik şekilde tanımlanıyorsa o metot Abstract sınıf içerisinde virtual olarak tanımlanır. Abstract anahtar sözcüğü yerine virtual anahtar sözcüğü kullanılır.

**Override Tanımı**

**Override**, oluşturmuş olduğumuz sınıf yapılarında kalıtım sayesinde üst sınıftan, alt sınıfa geçiş sağlayacak metot yapılarının aktarıldığı alt sınıf yapısına uygun bir şekilde gövdesinin değiştirilmesine ve kod satırlarının yeniden şekil almasına imkan sağlayan yapıya override denmektedir.

Abstract, interface öğrenme :

Entities klasörü : Varlık sınıfıdır. Nesneleri tanımladığımız nesnelerin classlarının oluşturulduğu klasördür.

Abstract klasörü : Interfacelerin bulunduğu klasördür. Soyut nesnelerimiz burada bulunur. Soyut sınıf.

Concrete klasörü : Interfacelerin somutlaştırılmış nesnelerinin bulunduğu klasördür. Somut sınıf.

Adapter klasörü : Mernis sisteminin bulunduğu klasör. Başka sistemler olacak ise de buraya eklenir.

metin, elektronik eşyalar içeren bir resim

Açıklama otomatik olarak oluşturuldu

Senaryo : **Kahve dükkanları için müşteri yönetimi yapan bir sistem yazmak istiyoruz. Yani müşterileri sisteme kaydeden bir sistem. Nero ve Starbucks firmaları için çalışma yapılacak. İki firma da müşterilerini veritabanına kaydetmek istiyor. Starbucks müşterileri kaydederken mutlaka mernis doğrulaması istiyor. Mernis doğrulama sistemi kişinin kimlik doğrulamasıdır yani Türkiye Cumhuriyetinde böyle bir kişi var mı kontrol eden sistem. Nero bu sistemi istemiyor sadece müşterileri kayıt edicek. Starbucks ayrıca müşteriler için her kahve alımında yıldız kazandırmak istiyor.**

**Entities / Customer.cs :** Bir müşteri sınıfı oluşturalım. Mernis doğrulama sisteminde kayıt edilirken şahsın ismi,soyismi,doğum tarihi, tc kimlik numarası istenmektedir. Bu özellikleri müşteri sınıfında tanımlayalım. Normal sınıf oluşturuyoruz yani. Varlık sınıflarının her zaman interfacelerini oluştur. Burada kuralımız şu hiçbir sınıfımız Çıplak Class Kalmasın. Customer sınıfım bir varlık sınıfı hemen git interface sınıflarının bulunduğu klasörde Varlık interfacesi oluştur.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Abstract / IEntity.cs :** Customer classının interface sınıfıdır. İçerikte bir şey yazmasına gerek yok. Sadece Customer classını çıplak bırakmamak için varlık classlarının her zaman interfacelerini oluştururuz. Customer classı artık bir IEntity’dir.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Abstract / ICustomerService.cs :** Müşteri sınıfımızın iş sınıfıdır. Abstract Soyut sınıf. Metotları yazalım. Sisteme kayıt için kaydetme operasyonu(Save) yazalım, parantez içinde parametrelerini verelim. Interface implementasyonu yapan herkes Save operasyonunu içermek zorundadır.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Abstract / ICustomerCheckService.cs :** Normalde sadece Starbucks firması mernis doğrulaması istiyor. Starbucks sınıfına gidip kişinin gerçek olup olmamasıyla ilgili operasyon yazabilirsin ama sonradan Nero firması da mernis doğrulaması sistemi isteyebilir bu yüzden iki firmada CheckIfRealPerson operasyonunu kullanacak, ikisinin de içeriği aynı, kod aynı bu kod implementasyonudur tek Interfacete tanımlarsı müşteriyi sisteme kayıt ederken gerçek kişi mi olup olmadığını burada kontrol edersin. ICustomerCheckService adında bir interface oluşturduk. Metodunu da CheckIfRealPerson(Kişi gerçek mi çek et) olarak adlandırdık. Metot parantezine parametreleri verdik.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Concrete / CustomerCheckManager.cs :** ICustomerCheckService sınıfımızın Concrete Somut sınıfıdır. Soyut sınıfların her zaman somut sınıflarını oluştururuz. Soyut sınıfta metotlar yazılıyordu, somut sınıfta ise yazılan metotların somutlaştırılması işlevselleştirilmesi yapılır. Veri tipi boolean(bool) eğer ki kişi gerçek ise true, değil ise false döndürsün diyerek metodu kullanılır hale getirdik.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Abstract / BaseCustomerManager.cs :** Şimdi normalde iki şirketimiz de NeroCustomerManager, StarbucksCustomerManager birer ICustomerService ‘tir. Senaryo da iki firmada müşterilerini veritabanına kaydetmek istiyordu ama aynı veritabanına, aynı tabloya kayıt edildiğini varsayalım bu durumda ne yapabiliriz ? Senaryo gereği ikisinin de veritabanı operasyonu aynı, ikisi de aynı tabloya yazılacak. İkisi de Save operasyonunu içerecek neden ikisine de aynı operasyonu ayrı ayrı yazalım ? Bir tane interface oluşturup orada yazarız Nero ve Starbucks şirketi de bu interfaceten kalıtım alır. Yani şunu demek istiyoruz ikisinin de ortak kodu var bu kod implementasyonudur dolayısıyla Abstract görevi görecek, kod içeriği aynı. Abstract class oluşturduk, BaseCustomerManager adını verdik. Artık BaseCustomerManager sınıfı bir ICustomerService’tir.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Concrete / NeroCustomerManager.cs** : Nero şirketi veritabanına kaydedecek bu operasyon kimde BaseCustomerManager da o zaman NeroCustomerManager artık bir BaseCustomerManagerdır. Nero doğrulama istemiyor, bir şey yazmıyoruz.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Concrete / StarbucksCustomerManager.cs :** Starbucks şirketi veritabanına kaydedecek bu operasyon kimde BaseCustomerManager’da o zaman Starbucks artık bir BaseCustomerManagerdır. Şimdi ben BaseCustomerManagerda yer alan Save operasyonunu kullanmak istiyorum orada virtual sözcüğünü kullanırsam ezebilirim demek bu, Starbucksta da override dersek Save operasyonunu kullanabiliriz. Starbucks’un doğrulamaya ihtiyacı var(senaryo gereği) Save kullanıyor. Save içerisinde de kişinin doğrulaması yaparak bu kişi varsa veritabanına kaydedelim diyoruz.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

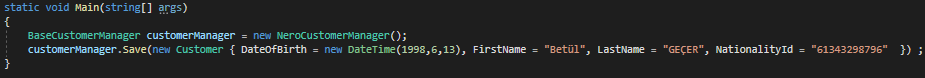
**Kırmızı Kutucuk içindeki ;** Sisteme kaydı için mernis doğrulaması istiyor ya bizim CheckIfRealPerson operasyonunu kullanmamız gerekiyor, bu operasyon ICustomerCheckService’te. Bir Manager sınıfının içinde başka Manager sınıfı yazarken onu tanımlamalısın ve constructor yapısını kullanmalısın.

**Adapter / MernisServiceAdapter.cs :** Müşterinin gerçek kişi olup olmamasına bağlı bu sistem o zaman MerniceServiceAdapter bir ICustomerCheckService’tir. O interfacete bulunan metotu taşımalı. KPSPublicSoapClient yazan ifade Müşteri gerçek kişi mi check edildikten sonra çalışacak mernis sistemine özgü ifadedir.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Program.cs :** Nero şirketi için mernis sistemine kayıt etme işlemini gerçekleştirdik.



Senaryo : Farklı bir senaryo oluşturalım. GameProject isimli bir proje oluşturalım. Oyuncu nesnesinin olduğu, oyuncuların sisteme kaydolabileceği, bilgilerini güncelleyebileceği, kayıtlarını silebileceği bir ortam simüle edelim. Müşteri bilgilerinin doğruluğu yine mernis doğrulama (e-devletten yapılacak). Oyun satışı yapılacak bir satış ortamı. Yapılan satış oyuncu ile ilişkili olacak. Sisteme yeni kampanya girişi, kampanyanın silinmesi ve güncellenmesi lazım. Satışlarda da kampanya entegrasyonu yapalım.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

İlk iş Varlıklarımızı(Entity) oluşturalım. Oyun, Kullanıcı, Kampanya ve Satış olmak üzere 4 varlık, nesne mevcut. Varlık classlarımız şu şekildedir ;

**Entities / User.cs :** Kullanıcı sınıfı. **Entities / Campaign.cs :** Kampanya sınıfı.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Entities / Game.cs :** Oyun sınıfı. **Entities / Sales.cs :** Satış sınıfı.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Varlık sınıflarımızın interface classını oluşturalım. Kuralımız neydi hiçbir class açıkta kalmayacak.

**Abstract / IEntities.cs :** Varlık classlarımızın interface sınıfı. Tüm nesneler bir IEntity’dir.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Abstract klasörü : operasyonların yazıldığı klasör. Interfaceler burada bulunur. Soyut sınıf. Önce varlıklarımızı oluşturduk sonra yapılacak operasyonlar için Service oluşturduk. Operasyon işlemlerine Interface ile başla. Ekleme, silme, güncelleme operasyonları yazılacak senaryoda istenilen gibi.

**Abstract / IUserService.cs :** Kullanıcı **Abstract / IGameService.cs :** Oyun

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Abstract / ICampaignervice.cs :** Kampanya

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Abstract / ISalesService.cs :** Senaryoya göre Satış sınıfı oyuncu ile ilişkili olmalı, kampanya satışı yapacak ve oyuna bağlı yapacak doğal olarak tüm nesnelerin parametrelerini içermeli. Burada Add operasyonu yerine CampaignSale ( Kampanya satışı ) operasyonu ekledik çünkü senaryo bizden öyle istedi.

metin, ekran, düz içeren bir resim

Açıklama otomatik olarak oluşturuldu

Concrete klasörü : Interfaceleri yazdığımız Service sınıflarının operasyonlarını somutlaştırmak için Manager sınıflarımızı Concrete klasöründe yazıyoruz.

**Concrete / GameManager.cs :** IGameService soyut sınıfının, somut sınıfıdır. GameManager sınıfı IGameService’tir. IGameService sınıfının operasyonlarını taşımak zorundadır. İmplemente ederiz.

metin, ekran görüntüsü, ekran, gümüş içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Concrete/ CampaignManager.cs :** ICampaignService soyut sınıfının, somut sınıfıdır. CampaignManager sınıfı ICampaignService’tir. ICampaignService sınıfının operasyonlarını taşımak zorundadır. İmplemente ederiz.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Concrete / SalesManager.cs :** ISalesService soyut sınıfının somut sınıfıdır. SalesManager sınıfı ISalesService’tir. ISalesService sınıfının operasyonlarını taşımak zorundadır. İmplemente ederiz.

metin, ekran, ekran görüntüsü, gümüş içeren bir resim

Açıklama otomatik olarak oluşturuldu

Senaryo bizden yeni bir kampanya satışı istemişti peki neye göre ?

**Yeşil çizgi ;** Biz indirime göre bu satışı belirledik. Price adında kampanya indirimini ifade eden bir değişken tanımladık. İf-Else ile Kampanya indirimi sıfıra eşit değil ise fiyat(price) : oyun fiyatı – (oyun fiyatı \* kampanya indirimi / 100) olsun. Değil ise oyunun fiyatı yazılsın.

**Renkli çizgiler ;** 0. Eleman GameName(oyun adını), 1. Eleman FirstName (Kullanıcı adını), 2. Eleman CampaignName(Kampanya adını), 4. Eleman price(indirimi) ifade eder.

UserManager sınıfımızı yazmadan önce senaryo bizden mernis doğrulaması istemişti önce onu yazalım.

**Abstract / IUserValidatonService :** Mernis doğrulaması yapacağımız için önce operasyonun bulunduğu iş sınıfını yazıyoruz. Validate : Doğrulama demektir. Kimi doğrulayacağız ? Kullanıcıyı o yüzden Validate (doğrulama) operasyonuna parametre olarak kullanıcıyı veririz.

metin içeren bir resim

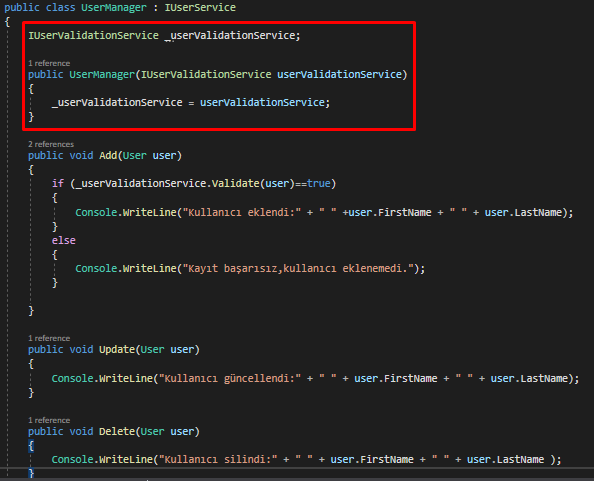
Açıklama otomatik olarak oluşturuldu

**Concrete / UserValidationManager :** IUserValidationService soyut sınıfımızın somutlaştırılmış sınıfı UserValidationManagerdır. Validate operasyonunu burada işlevselleştiriyoruz, somutlaştırıyoruz.   
E-devlet mernis kimlik doğrulaması ; isim,soyisim,doğum yılı,tc numarasına göre yapılır. Kişi bu şartlara uyarsa true, uymazsa false döndür diyoruz.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Concrete / UserManager.cs :** Kullanıcı nesnesinin somut sınıfıdır. UserManager sınıfı IUserService’tir.  
IUserService sınıfının operasyonlarını taşımak zorundadır. İmplemente ederiz.



**Kırmızı kutucuk içindeki :** Biz ekleme işlemini neye göre yapacağız ? Kullanıcının doğrulanmasına göre Validate metodunu kullanmamız gerekiyor. Validate operasyonu nerede ? IUserValidationService’te. IUserValidationService sınıfını çağırmalıyız. Bir manager sınıfının(UserManager) içinde başka bir Manager sınıfı(IUserValidationService) yazarken onu newlemeyiz, constructor yapısını kullanırız.

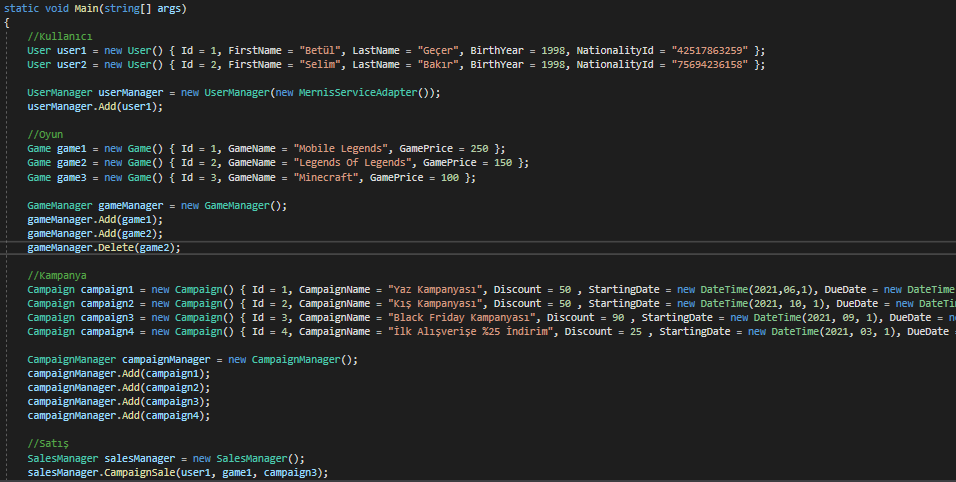
Adapter klasörü : Mernis sisteminin olduğu klasör. Mesela sisteme sonradan farklı bir sistemde eklenecek ise buraya eklenir bu bize yapı bozulmadan farklı sistemleri simüle edebileceğimizi gösterir.

**Adapter / MernisServiceAdapter.cs :** Kullanıcının gerçek kişi olup olmamasına bağlı bu sistem o zaman MerniceServiceAdapter bir IUserValidationService’tir. O interfacete bulunan metotu taşımalı. KPSPublicSoapClient yazan ifade Kullanıcı gerçek kişi mi doğrulandıktan sonra çalışacak mernise sistemine özgü ifadedir.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Program.cs :** Menü. Varlık sınıflarını newle özelliklerine değişkenleri ata ve iş sınıflarını newle ve yapmak istediğin işlemin operasyonunu (metot) çağır. UserManager iş sınıfını çağırdığımızda bizden servisi tanımlamamızı ister.



**VERİTABANI MODELLEME**

RDMS(Relational Database Management System/İlişkisel Veritabanı Yönetim Sistemi) ; Oracle, Sql Server, Mysql, Postresql, Access sektörde en yaygın veritabanlarıdır. Bu veritabanlarının tamamını kabul ettiği bir standart var. ANSIISQL standartıdır. Normalisation dediğimiz modelleme standartlarını kullanan yapıları benimserler.

NoSql ; MongoDb, Firebase

Normalisation tasarım ve modelleme tekniği, datayı modelleme tekniği. Doğru modelleme ve amaç veri kaçakçılığını engellemektir.

Overdesign : Aşırı tasarım , C# ta DTO = Data Transformation Object

**SQL**

**SELECT**

Select Sütun\_Adı from Tablo\_Adı ya da Select (\*) from Tablo\_Adı \* : sütunların tamamı

* Müşteriler tablosunun Adi, şirket adi, şehir sütunlarını getir.

Select ContactName Adi, CompanyName Sirketadi, City Sehir from Customers

* Müşteriler tablosunda Şehri Berlin olanları getir.

Select \* from Customers where City = 'Berlin'

* Ürünler tablosunda Kategorisi 1 olan veya kategorisi 3 olan ürünleri getir.

Select \* from Products where CategoryID = 1 or CategoryID = 3

* Ürünler tablosunda Kategorisi 1 olan ve birim fiyatı 10 dan büyük olan ürünleri getir.

Select \* from Products where CategoryID = 1 and UnitPrice>=10

* Ürünler tablosunda ürünleri birim fiyatına göre düşen olarak sırala

select \* from Products order by UnitPrice desc

--ascending(artan) descending(düşen)

* Ürünler tablosunda kategorisi 1 olan ürünlerin birim fiyatına göre düşen olarak sırala

select \* from Products where CategoryID=1 order by UnitPrice desc

* Ürünler tablosunda kaç tane ürün olduğunu sayıyor

Select count(\*) from Products

* Ürünler tablosunda kategorisi 2 olan kaç ürün var say göster adı da Adet olsun.

Select count(\*) Adet from Products where CategoryID=2

* Ürünler tablosunda her kategorideki ürünlerin sayısını gösterme

Select categoryId,count(\*) from Products group by CategoryID

\*Group By kullandığımız zaman Select ettiğimiz kolon sadece Group by da yazdığımız alan olabilir ve kümülatif datalar olabilir. (Count gibi)

* Ürünler tablosunda ürün sayısı 10 dan az olan kategorileri listele

Select categoryId,count(\*) from Products group by CategoryID having COUNT(\*)<10

* Önce her zaman where çalışır. Fiyatı 20 den fazla olan ürünleri categoryıd ye göre grupla onlardan da sayısı o grupta 10 dan az olanları sırala

Select categoryId,count(\*) from Products where UnitPrice>20

group by CategoryID having COUNT(\*)<10

\*Buradaki where koşulu kümülatif dataya ( count(\*)) yapılır. Group by larda kümülatif dataya yapılan koşul havingdir. havingi her zaman kümülatife uygularız.Count,sum, vs.

\*Joın = tabloları birleştirmek

\*İnner Join iki tablolada da sadece eşleşenleri bir araya getirir eşleşmeyen data varsa onu getirmez

\*left join solda olup sağda olmayan, right join sağda olup solda olmayan

* Ürünler ve kategoriler tablolarını birleştir .Yanyana birleşik tüm kolonlar gelir.

Select \* from Products inner join Categories

on Products.CategoryID = Categories.CategoryID

* Ürünler tablosunda fiyatı 10 dan büyük olanları kategoriler tablosuyla join et. Tablo birleştirmede hangi tablodan hangi kolon gelecek ?

Select Products.ProductID, Products.ProductName, Products.UnitPrice, Categories.CategoryName

from Products inner join Categories

on Products.CategoryID = Categories.CategoryID

where Products.UnitPrice>10

* Burada left join solda olup(products) sağda olmayanları(order detais) getir istiyoruz

Select \* from Products p left join [Order Details] od

on p.ProductID = od.ProductID

Order details tablosunun köşeli parantez içinde yazmasının sebebi aradaki boşluğu yanlış anlamasın diye sql köşede yazıyor yazım hatasından kaynaklı tedbir alıyor. p,od alias takma addır

* Müşteri ve sipariş tablosunu birleştir. 830(kolon sayısı) kişi sipariş veren müşteri

Select \* from Customers c inner join Orders o

on c.CustomerID = o.CustomerID

* 832(kolon sayısı) kişi solda olup(müşteri) sağda olmayan(orders). Bu demekki 2 kişi sipariş vermemiş

Select \* from Customers c left join Orders o

on c.CustomerID = o.CustomerID

* Sipariş vermeyen iki müşteriyi göster bana

Select \* from Customers c left join Orders o

on c.CustomerID = o.CustomerID

where o.CustomerID is null

sağda olmayanlar null olarak gelir çünkü solda(customers) var sağda(order) yok, nullar is ile gösterilir. Primarykeyimiz customerıd null’u ona yollarız. Primarykey hiç boş olamaz ondan.

* Birden fazla tabloyu join et.

Select \* from Products p inner join [Order Details] od

on p.ProductID = od.ProductID

inner join Orders o

on o.OrderID = od.OrderID

* Her bir üründen toplamda ne kadar para kazanıldığını bulunuz. Group by kullanılacak. Products,orders,order details tabloları join edilecek. SUM kullanılacak.

Select p.ProductName[Urun Adi], Sum(od.UnitPrice \* od.Quantity) as ToplamKazanc

From Products p

inner join [Order Details] od on p.ProductID = od.ProductID

inner join Orders o on od.OrderID = o.OrderID

Group by p.ProductName

Order by ToplamKazanc asc

**LINQ İLE VERİ SORGULAMA**

Gerçek hayat projelerinde veritabanı programlama yapıyoruz. Veritabanı programlama otomasyon sistemlerinin bel kemiğidir. Veritabanı programlama yaparken kodlarımızı farklı parçalara bölüyoruz. Katmanlara (layers) bölüyoruz.

1. **Data Access ( Veri Erişim ) :** Veriye ulaşmak için yazdığımız kodları içeren katmandır. Veriye erişmek için gerekli olan kodlar buraya yazılır. Veriye erişim için farklı teknikler vardır.   
   Değişiklik yapılacağı, veri erişim değiştirileceği zaman sadece burası katman değiştirilir.
2. **Business ( İş Katmanı ) :** İş kodlarını, kurallar bu katmana yazılır. En çok if’i kullanan katmandır.
3. **UI ( User Interface / Kullanıcı Arayüzü ) :** .NET ile birlikte geliştirilebilecek, herhangi bir arasayfasına ihtiyaç duymadan direk Business ‘ a bağlanıp geliştirilebilecek versiyonları var. Bunlardan sektörde en yaygın kullanılan ASP.NET. Günümüzde de farklı ortamlarda da yazılım geliştirmek durumundayız. Bu Android, IOS ürünü olabilir. .NET’le (Android, IOS) Business ‘i direkt görüştüremeyiz. Çünkü bunlar farklı ortamlar, bunların birbiriyle görüşme şansı yok. Farklı sistemlerin birbirini anlayabilmesi için bir teknoloji kullanıyoruz. Farklı sistemlerin birbirini anlayabilmesi için Service odaklı mimariler dediğimiz bir katman daha yazarız. Günümüzde bunun karşılığı olarak API dediğimiz altyapıları en yaygın biçimde kullanıyoruz.

Yani Business’i ona bağlı olarak DataAccess’i API katmanı vasıtasıyla dış dünyaya açıyoruz. Bu API Restfull denilen standart mimariyle karşımıza çıkıyor. Bu da bir formatla (json) denilen veri formatıyla karşımıza çıkıyor.

Kısacası DataAccess’te SQL sorgularını çalıştırırız. (E-ticaret sisteminde kayıt olma işlemi, bilgileri veritabanına kaydetme(SQL-Insert) işlemi gibi.) Sadece SQL değil, excel’den veya metin dosyalarından da data çekebiliriz. Business’ta iş kurallarını yazıyoruz. UI kullanıcıyla etkileşimli, ekranların olduğu yer.

|  |
| --- |
| DATA ACCESS |

Kaç ürün var ?

EKLENDİ

Kaç ürün var ?

9 adet

|  |
| --- |
| BUSİNESS |

Ürünü ekleyeyim mi ?

EKLENDİ

|  |
| --- |
| UI |

UI ‘ da tedarikçi ürün eklemeye çalışıyor. Bilgileri aldı, Business’a gidiyor. Diyor ki bu tedarikçi böyle bir ürünü eklemeye çalıştı. Business kuralları da şöyle olsun ; eklenecek ürün minimum 2 karakter olsun ve 10 dan fazla ürün ekleyemesin. Business if ile kontrol etti okeyledi bu ürün bu şartı sağladı, tekrar if diyip bu tedarikçinin daha önce eklenmiş ürünleri var mı? DataAccess’ gidiyor. Diyor ki bu tedarikçinin sistemde kaç ürünü var (select veritabanına işleyecek DataAccess) Data Access’ ten dönüş 9 adet olsun. İş kurallarından geçti. Business Data Access’e al bunu ekle diyor. Data Access Business’ a ekledim diyor. Business UI’ a ekledim diyor. UI da tedarikçiye ürün eklendi der.

Data Access : ürün ekleme işlemi, Business : kontrol, UI : ürünü gösterme

Katmanlı mimariyi öğrenelim.

* DataAccess : Veri erişim,
* Business : İş katmanı,
* ConsoleUI: Konsol arayüzü,
* Entities : Varlık, yardımcı katman. Tüm katmanların kullanabileceği katman.
* Concrete : Somut yani gerçek işi yapan nesneler.
* Abstract : Soyut nesneleri, abstract classları, interfaceleri, base classları, referans tutucuları koyarız.
* Public bu class’a diğer katmanlarda ulaşabilsin demek. Çünkü DataAccess ürünü ekleyecek, Business ürünü kontrol edecek, ConsoleUI ürünü gösterecek. Entities’i bu üç katmanda kullanacak o yüzden erişimi sağlayabilmek için public kullanırız.
* Product : IEntity Çıplak Class Kalmasın. Bu şu demek varlıkları(entities) işaretleme(gruplama) eğilimine gidilir. Nedir bu gruplama ? Concrete klasöründeki sınıflar bir veritabanı tablosuna karşılık geliyor.
* Sektörde DataAccess için 2 nesne, isim kullanılır. Genellikle Dal(.NET çiler) ve Dao(Javacılar) kullanır. Data Access object, veri erişim objesi. İkisini de kullanabilirsin, bir yerde Dal veya Dao nesnesini gördüğün zaman hangi katmana ait olduğunu anlarsın. Yani veri erişim katmanına.

Senaryo : E-ticaret sistemi için ürün listeleme, ekleme, güncelleme ve silme işlemleri yapılacak. Katmanlı yapıyla bunu gerçekleştireceğiz. MyFinalProject adında bir proje oluşturalım. 4 katmanımız mevcut. ConsoleUI hariç diğer katmanların Abstract ve Concrete klasörü olmalı.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Önce varlıklarımızı(entities) oluşturuyoruz. Products ve Category adında iki nesnemiz mevcut. Varlıklar somut Concrete klasörüne, interfaceler soyut Abstract klasörüne yazılır.

**Entities/Concrete/Product.cs :** Product nesnesi. Bir IEntity’dir.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Entities/Concrete/Category.cs :** Category nesnesi. Bir IEntity’dir.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Entities/Abstract/IEntity.cs** : Varlık classları oluşturulduktan sonra her zaman interface classını oluştur.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

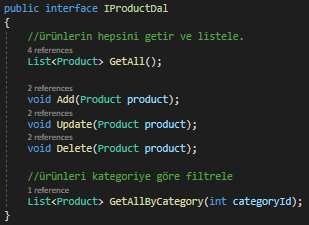
Varlıklarımızın classlarını oluşturduk ve interface classını oluşturduk. Entities katmanını tamamladık. Gelelim DataAccess katmanına. Şimdi bu katmanda da Concrete ve Abstract klasörüm var. Abstractta soyut nesnem(interface), Concrette somut nesnem(interface’in somutlaştırılması) bulunmalı. İlk iş her zaman interfacelerini oluşturmak olsun. İş yapacağın classların her zaman interfacesini oluştur.

**DataAccess/Abstract/IProductDal.cs :** Veri erişim katmanında Abstract varlıklarının interfacesini oluşturuyorsun. Neden IProductService değil de IProductDal ? Yukarıda bahsettiğimiz gibi sektörde Data Access için 2 isim kullanıyoruz : Dal,Dao. Burada da katmanlı mimari kullanıyoruz ya Dal diyerek o İnterface sınıfının Data Access katmanına ait olduğunu anlıyoruz.

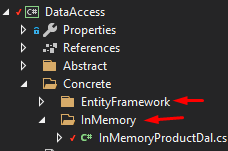
IProductDal : Productla ilgili veritabanına yapacak operasyonları(şunu ekle, şunu sil) içeren interface.

* I : İnterface
* Product : Hangi tabloya karşılık geldiğini (entities’e yani)
* Dal : Hangi katmana karşılık geldiğini ( Data Access Layer )

IProductDal classımızda ürün listeleme, ürün ekleme, ürün güncelleme, ürün listeleme ve ürünleri kategoriye göre filtreleme operasyonları oluşturalım.

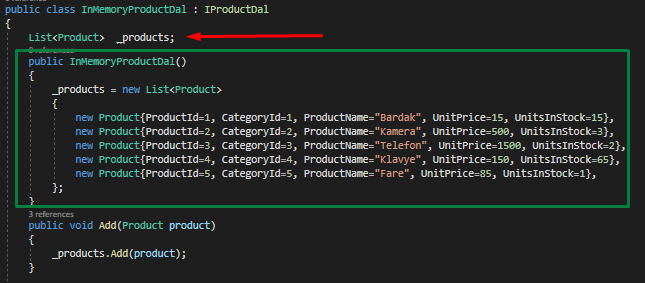


Data Access Abstract klasörümüzü oluşturduktan sonra şimdi Concrete klasörünü oluşturacağız. Bir projede alternatif sitemler olabilir. Concete klasöründe alternatif sistemlerimizi(x,y,z) bir klasörle belirtelim. Burada Entity Framework ve InMemory adında iki klasör oluşturalım. Şimdilik bellekte çalışalım. Veritabanı olmadığını varsayalım bellekte(InMemory) işlem yap, sanki veritabanı varmış gibi.



**DataAccess/Concrete/InMemory/InMemoryProductDal.cs :**IProductDal’da operasyonlarımızı yazdık. İnterface’i(IProductDal) iş yapan sınıf haline getirmeliyiz. Unutma interface yapıyorduk bir de somutunu yapıyorduk. İnterface’i burada implemente ediyoruz, somutlaştırıyoruz. Concrete somut nesne = InMemoryProductDal (Bellekte ürünler ile ilgili erişim kodları yazılacak yer). InmemoryProductDal bir IProductDal’dır yani onun bir implemantasyonu demek.

Şuan bellekte veri varmış ,biz onu yönetiyormuşuz gibi davranıyoruz. Dolayısıyla proje başladığında veri varmış gibi ortamı simüle edelim. Veri varmış gibi davranacağımız için ürün listesi oluşturalım. Şöyle düşün IProductDal’ da ürünlerin hepsini getir diye operasyon var, bir ürün listesi olması lazım ki getirsin.



Proje çalıştırıldığında bizim yerimize bir tane bellekte ürün oluşturdu. Sanki bu bize Oracle, Sql server, Postgres, MongoDb veritabanlarından geliyormuş gibi simüle ediyoruz.

**Kırmızı çizgi ;** Products isimli bir değişken tanımladık. Değişken nasıl tanımlanır : string products. Bizim bir ürün listemiz varmış gibi davranıyoruz string yerine List< > veri tipini kullandık.

Bu nesneyi classın içinde ama metotların dışında tanımlandığı zaman bu tip değişkenlere global değişken deniyor yani o class için global. Genellikle alt çizgiyle gösterilir. (\_produts) Bu da nameing commention adını verdiğimiz bir isimle standartıdır.

Kırmızı çizgiyle gösterdiğimiz ifade bir referans tip tek başına bir anlam ifade etmiyor sadece değişkeni oluşturdu. Projeyi başlatınca bellekte bir tane ürün listesi oluştur ifadesini yansıtır.

**Yeşil çizgi içindeki ;** Constructor yapısıdır. Bellekte referans aldığı zaman çalışacak olan bloktur. Void vs. döndürmüyor, direk classın ismiyle olunca constructor oluyor. Yani değişken oluşturduk ya o referans tip tek başına bir anlam ifade etmiyordu, constructor yapısıyla değişkene veri atadık. İçerisinde ürünleri barındıran bir liste oluşturduk.

IProductDal classındaki operasyonları yazıyoruz. Ekleme operasyonunu yazdık. Bu ürünü bu List<Product> \_products veri kaynağımıza ekleriz. Yani elimizdeki \_products’a Add diyerek parametre olarak Business’tan gönderilen ürünü product veritabanına ekliyorum. O da List olduğu için bu şekilde listeye eklenir. Şimdi diğer operasyonlarımızı yazalım.

Delete operasyonunu bu şekilde yazdığımızda listeden ürün silinmez neden ?

metin, portakal, izleme, küme içeren bir resim

Açıklama otomatik olarak oluşturuldu

Arayüzden bir tane ürün newleyip gönderildiğinde bunun referansı 200 diyelim, sen ürünler listesinden 200 ‘ü sil diyorsun. Ama 200 yok çünkü oluşturduğum adreslerin referans numaraları farklı bu yüzden silmez bu ifade. Burada yapılması gereken (Product product) bunun id’sini kullanarak ürünler listesindeki adreslerden eşleşen id’ yi bulup o referansı yakalamak. Bu sisteme de LINQ denir.

LINQ-Language Integrated Query (Dile Gömülü Sorgu)

metin içeren bir resim

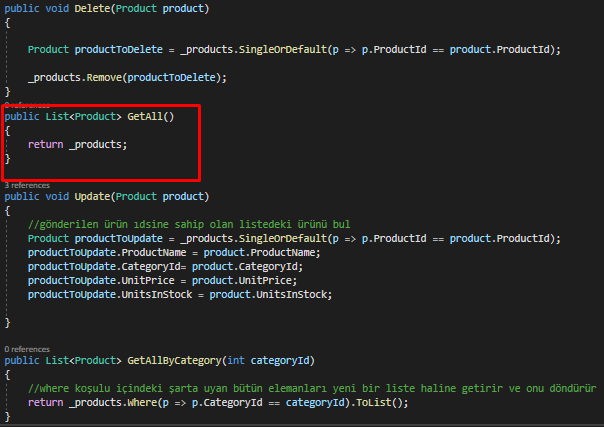
Açıklama otomatik olarak oluşturuldu

productToDelete = silinecek ürün. Foreach ile ürünleri tek tek dolaş. Her dolaştığındaki p o an dolaştığın elemana denk gelir. İf ile gönderilen product’un ProductId’si eşit eşitse o anki product’un ProductId’si kontrolü, doğruysa silinecek eleman o anki bu eleman diye şartlarız. Normalde burada az eleman olduğu için foreach ile elemanları tek tek dolaşırız ama bu işlem linq ile daha kolay bir şekilde yapılabilir.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

SingleOrDefault tek bir eleman bulmaya yarar. Bu ürünleri tek tek dolaşmaya yarar. p tek tek dolaşırken verilen takma isim foreachteki gibi.



Geri kalan operasyonlar şekildeki gibidir. Kırmızı çizgi içindeki ; GetAll operasyonu veritabanındaki datayı(List<Product>) Business’a vermek lazım. Business ürün listesini istediğinde, ona liste verilmeli. Tüm listeyi return ifadesi ile veritabanını olduğu gibi tümünü döndürüyoruz.

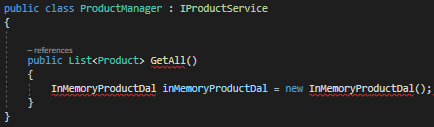
Gelelim iş katmanımıza.

**Business/Abstract/IProductService.cs :** İş katmanının soyut interface’i Service.

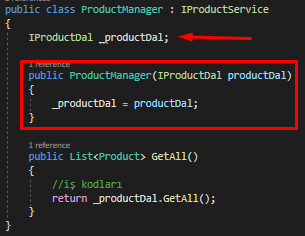
metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Business/Concrete/ProductManager.cs :** İş katmanının somut sınıfı Manager.



Bunu bu şekilde yazarsak iş kodlarının tamamı bellekle çalışır. Veritabanına geçeceğin zaman mesela EntityFramework kullanacağın zaman GetAll( ) gibi binlerce operasyon var, hepsini değiştirmen gerekir. Kural bir iş sınıfı başka sınıfları newlemez. Bunun yerine soyut nesne ile bağlantı kuracağız.



**Kırmızı ok ;** Değişken, referans tutucu, constructor injekte et.

**Kırmızı çizgi içindeki ;** Bu ne işe yarıyor ? ProductManager newlendiğinde constructor diyor ki bana bir tane IProductDal referansı ver. (InMemory, EntityFramework, Depar vs.)

Sonrasında IProductService interface sınıfımızın GetAll operasyonunu somutlaştırırız. Return ifadesinden önce varsa iş kodları if’ler bulunur. İf böyleyse,şöyleyse (Yetkisi var mı ?) gibi kontroller bulunur. Kuralları da yazdıktan sonra diyelim ki geçti DataAccess’e diyor ki bana ürünleri verebilirsin ben kurallardan geçtim.

**ConsoleUI/Program.cs :** Listedeki tüm ürünlerin ismini yazdırır.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Projemizi geliştirelim 8.Hafta :

Senaryo Devam : 2 hafta sonra bu projeye yeni nesne ekleyelim her şey sıfırdan tekrarlayalım.

**Entities/Concrete/Customer.cs :** Customer nesnesi. Bir IEntity’dir. **Entities** yeni hali.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

DataAccess katmanımıza gelelim IProductDal gibi diğer nesnelerimizin de operasyonlarını yazmalıyız.

**DataAccess/Abstract/IProductDal.cs :** **DataAccess/Abstract/ICategoryDal.cs :**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

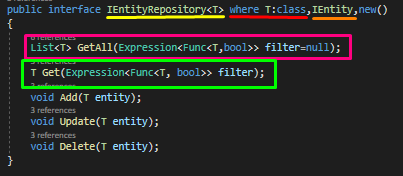
Product gibi Category nesnesini de yazdık. Veriden geliyormuş gibi kategori listesi oluşturduk. Operasyonlarını yazdık. Kategoriye göre filtreleme işlemi yaptık. Aynı productaki gibi. Değişen tek şeyin List’in veya operasyonların parantez içlerinin değiştiğini gördük yani product veya category. Bu şekilde bir sürü tablomuz, varlığımız olduğunu düşün her nesne için aynı işlemler yapmak yerine jeneric yapısını kullanabiliriz. Çünkü sadece tip değişiyor. Customer nesnesinin operasyonları da product ve category nesnelerinin operasyonuyla aynı , 3 farklı sınıfta aynı operasyonu, içeriği tekrarlıyoruz.

Hatırla Jenerikler tasarladığımız interface, class, metod ya da parametrelerin(argümanların) belirli bir tip için değil bir şablon yapısına uyan her tip için çalışmasını sağlayan bir yapıdır. Tiplere genelde T denir. T ne istersen o olur, type için. < > içerisine tip yazılır.

Bunun yerine bir tane interface yapsak jenerik tipi olsa IProductDal daki product nesnesi veya ICategoryDal daki category yerine jenerik T yapsak daha kolay olur. Bu yapının adı da Generics Repository Design Pattern.(Jenerik Repository Tasarım Deseni)

Şöyle düşün operasyonlar ortak, içerik ortak değişen şey tip. jeneriklerde T yapısı kullanıyorduk burada bunu kullanabileceğimiz ortak bir class oluşturuyoruz.

İsmi IEntityRepository. Jenerik yapısı yapalım IEntityRepository<T> Buradaki T bana çalışacağım tipi söyle. Product dersem Product olacak, Category dersem Category olacak, Customer dersem Customer olacak.



Backend kodluyoruz belirli kurallar var karşılaşılan biz de bu IEntityRepository<T> T’yi sınırlandırmak istiyoruz. Herkes T’yi yazamasın T olarak gelmesi gereken veritabanı nesneleridir. Sonuçta biz neyi listeler, ekler, günceller ve sileriz veritabanı nesnelerimizi(entities).

Bunu sınırlandıralım buna GENERİC CONSTRAİNT (Jenerik Kısıt) deniyor.

Yani mesela ICategoryDal da ICategoryDal:IEntityRepository<Category> category yerine int yazsak sistem bunu da kabul eder tip çünkü T yerine hepsi yazılabilir. Ama biz T için sadece nesnelerimizi yazalım diye kural koyuyoruz.

**Sarı çizgi ;** Jenerik yapısı.

**Kırmızı çizgi ;** Where koşulu ile T class olsun diyoruz. Tip olarak int,string filan olamasın.

* **where T : class** ; class : buradaki class referans tip olabilir demek.

Yani ICategoryDal : IEntityRepository<Category> Category yerine int yazsak kabul etmez bu sefer çünkü referans tip engeller. İnt,string değer tip engeller bunları.

**Turuncu çizgi ;** Where T:class olduğunda ise yine Category classı yerine bir class ismi yazılsa hata vermez o yüzden sadece IEntity classını şartlayacağım ki bana sadece nesnelerimi versin kısıtlasın. Nesne classlarının ortak özelliği hepsinin IEntity olması.

* **where T : class, IEntity**

Bu şu demek T bir referans tip olmalı (class) ve ya IEntity ya da IEntityden implemente olan bir   
şey olabilir.

* **IEntity**; IEntity olabilir veya IEntity implemente eden bir nesne olabilir.

Bu süreçlerden sonra ICategoryDal : IEntityRepository<Category> Category yerine customer, product, category ve IEntity yazılabilir sadece. Ama ben IEntity yazmak istemiyorum soyut nesne işimi görmüyor, devre dışı bırakmak istiyorum.

* **where T:class,IEntity,new( ) ;** bu şekilde yazıldığında devre dışı bırakabilirim nasıl ?

new : new'lenebilir olmalı. IEntity interface newlenemez. Customer, Product, Category somut nesne newlenebilir.

Jenerik repository nesnelerinde genellikle liste, add, update, delete ve id’ye göre getirme olur.

**Pembe Kutucuk ;** Mesela kategori idsi 2 olan ürünleri getir. Bu filtreleme işlemini yapmak için manager sınıfında GetAll operasyonu içerisine return \_productDal.GetAll(p=>p.CategoryId == 2); yazarız.

GetAll operasyonunda datanın tamamını değil de belli bir kısmını getirmek istediğimizde böyle bir filtre vermemizi sağlayan yapının ismi EXPRESSİON’ dır. Bir şey verebilmemizi sağlayan yapı Expression’dır. Expression’ı verebilmenin syntax’ı budur. Bu da Linqle beraber geliyor. Filter == null filtre vermeyedebilirsin demek. Eğer filtre vermemişse tüm datayı istiyor, filtre vermişse filtreleyip verecek.

Yani kategoriye göre getir, ürünün fiyatına göre getir gibi ayrı ayrı metotlar yazmak gerekmiyor bu yapıyı kullandığımızda.

**Yeşil kutucuk ;** GetAll( ) gibi bir tane T döndüren Get( ) operasyonu yazalım.

Tek bir data getirmek için kullanırız bu genellikle bir şeyin detayına gitmek için mesela bir bankacılık uygulamasında hesaplar var GetAll( )’ la liste olarak geliyor, bir tane hesaba tıklayıp o hesabın detayına gidiyoruz o zamanda Get( ) kullanırız. Burada filtre vermek zorunda olduğumuz için filter == null kullanmayız.

Operasyonları barındıran tek bir class yaptığımıza göre DataAccess katmanımızın classlarını düzeltelim.

**DataAccess/Abstract/IProductDal.cs :** IProductDal sen bir IEntityRepositorysin ve çalışma tipin Productdır. Yani IEntityRepository’i Product için yapılandırdın demektir.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**DataAccess/Abstract/ICategoryDal.cs :** ICategoryDal sen bir IEntityRepositorysin ve çalışma tipin Categorydir. Yani IEntityRepository’i Category için yapılandırdın demektir.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**DataAccess/Abstract/ICustomerDal.cs :** ICustomerDal sen bir IEntityRepositorysin ve çalışma tipin Customerdır. Yani IEntityRepository’i Customer için yapılandırdın demektir.



**DataAccess/Abstract/IEntityRepository.cs :**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin içeren bir resim

Açıklama otomatik olarak oluşturulduBunlar alternatif sistemler. Projeyi geliştirmeden önce InMemory bellekte çalışıyorduk. Şimdi EntityFramework ile çalışacağız. Projeye framework ekledik. İlerleyen zamanlarda farklı bir sisteme Deeper’a geçebiliriz gibi.

**Entity Framework**

ADO.NET : Veritabanına bağlanmamızı sağlayan, veritabanını kontrol etmemizi sağlayan, veritabanında sorgu yazmamızı sağlayan bir kütüphane. Bu kütüphane vasıtasıyla verilerimiz ekleyebilir, listeleyebilir, güncelleyebiliriz vs vs. Ama bununla yazdığımız kodlar biraz yorucu.

Günümüzde bu süreçleri kolaylaştırmak ,nesne yönelimli programlamayla ilişkiyi daha kolaylaştırmak, sağlamak adına çeşitli ORM dediğimiz yapıları kullanırız.

Object Relational Mapping : Nesne İlişki Bağdaştırması. Nesnelerin veritabanıyla bağdaştırılması. Bunu bize sağlayan bir frameworktür. Entity Framework

Entity Framework Microsoft’un bir ürünü. ORM dediğimiz bir ürünü. Linq destekli çalışıyor. Amaç şu Orm demek veritabanındaki tabloyu sanki classmış gibi onunla ilişkilendirip bundan böyle bütün operasyonları yani sqlleri bizim linq ile yaptığımız bir ortam. Orm kodlar ile veritabanı nesneleri arasında bir ilişki, bağ kurup veritabanı işlerini yapma süreci.

Veritabanı isimlendirmelerinde çoğul kullanılır. Products mesela veritabanı tablosu.

Entity Framework ile bu tablo(products) ve class arasındaki ilişkilendirmeyi nasıl kurarım ? İşte bunun için Context dediğimiz bir yapı oluşuyor. Context classı veritabanıyla(veritabanındaki Products tablosuyla) classımız(Product nesnesi) arasındaki ilişkiyi kurduğumuz classtır. Aynı zamanda veritabanına bağlantı açmamızı sağlayan da yerdir. İsminin Context olması Context olduğu anlamına gelmez base classını söylemeliyiz DbContext diye Entity Framework classı vardır.

Entity framework sisteme nasıl dahil edilir ?

Bugüne kadar C# ın kendisini kullandık. C# in .NET içerisinde implementasyonlarını kullandık ama biz ilerledikçe başkalarının yazdığı (bunlara paket diyoruz) kodları da kullanacağız. bu kodların ortak koyulduğu ve yönetildiği bir ortam var, bu ortamın adı NuGet.

NuGet : .NET ortamındaki kullanabileceğimiz paketleri barındıran yer.

.NetCore içerisinde default olarak Entity Framework bir paketle geliyor. Bu paketin adı Entity Framework.

**Dataccess'e sağ tıkla >> Manage Nuget seçeneğinde >> Browse >> .netframeworkcore.sqlserver kur(3.1.0) // dataccess properties target framework - 4.7.2**

Şu ana kadar yapılan iş Entity Framework kurulumu. Veritabanı var artık.

**İlk Aşama**

Veritabanıyla kendi nesnelerimizi kısacası; Veritabanındaki products tablosu ile nesnelerimizdeki product tablosunu, veritabanındaki categories tablosu ile category tablosunu, veritabanındaki customers tablosu ile customer tablosunu ilişkilendirmemiz gerekiyor. Bu işlemi yapabilmek için Context denilen yapıyı kurmalıyız.

* **Context :** Veritabanı ile kendi classlarımızı ilişkilendirdiğimiz class. Db tabloları ile proje classlarını bağlamak.

**DataAccess/Concrete/EntityFramework/NorthwindContext.cs :** Biz northwind diye bir sistemle, çalışıyorduk başta hatırlarsan. NorthwindContext diye class oluşturduk Entity Frameworkta. NorthwindContext dememiz ismine bunun Context olduğu anlamına gelmez. Entityframeworkla birlikte DbContext adında base bir class gelir. DbContext aslında bizim contextimizin ta kendisi.

Entity Framework ben nereye bağlanacağım diye sana soruyor veritabanını belirtmelisin.

DbContextin içerisinde OnConfiguring isimli bir metod var. Bu metod senin projen hangi veritabanıyla ilişkiliyi belirtilecek yer.

OnConfiguring adlı metodu override etmemiz gerek. Override üzerine yazmak demek.

Virtual bir classın içerisinde bir metodu yazıyoruz ama onu inherit eden kişi yani NorthwindContext.DbContext onu inherit eden yer(NorthwindContext isterse o kodun içerisini değiştirebilsin.

A classı içerisinde Do diye metodumuz olsun. B classı A'dan inherit eder. Yani Do B'nin içinde de geçerli. (DbContext içerisindeki OnConfiguring NorthwindContext'te de geçerli.) Ama diyoruz ki istediğimiz zaman B'nin içerisinde bu Do metodunu değiştirebileyim. İşte bu işlemi yapabilmemizin tek yöntemi onu override etmektir. B'nin içerisinde Do'yu değiştirmek istiyoruz. Bunun için virtual keywordinden yararlanırız. Virtualdan sonra override yazdığımızda do metodunu bize verir. Bu şu demek ben Do metodunun üzerine yazıyorum.

public class A

{

public virtual void Do

{

//

//

}

}

public class B:A

{

public override void Do

{

base.Do(); //basedeki do yu çağır. A daki do yu çağır.

}

}

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Kırmızı çizgi ;** Sql Server kullanacağız. Sql Server’a nasıl bağlanacağını belirtmelisin.

**Mor çizgi ;** optionsBuilder.UseSqlServer(@"Server = 175.45.2.12"); normal projede böyle IP görürüz.

Ama biz development ortamındayız onun yerine viewda sql server object explorerdaki yeri yazman gerekir. (localdb)\ mssqllocaldb

**Sarı çizgi ;** Bu serverda hangi veritabanı ? Onu da belirt. Database=Northwind

**Beyaz çizgi ;** Trusted\_Connection=true : kullanıcı girişimiz yok ya, bu veritabanına direk doğrudan giriş gibi. Kullanıcı adlı şifre gerektirmeden.

**Yeşil kutu ;** Context hangi veritabanına bağlanacağını buldu yukarıda. Burada ise benim hangi classım hangi tabloya karşılık geliyor ?

**DataAccess/Concrete/EntityFramework/EfProductDal.cs :** EfProductDal sen bir IProductDal’sın. Soyutlama tekniği somutlaştırıyoruz. İnterfaceteki operasyonlarımızı implemente ediyoruz.

Bir classı newlediğimizde Garbage Collector(çöp kutusu) belli bir zamanda düzenli olarak gelir ve bellekten onu atar.

Using içerisine yazılan nesneler, using bitince garbage collectore gelip, beni bellekten at diyor.

Burada using içerisine yazılan northwindcontextle işi bitince bellekten atılacak.

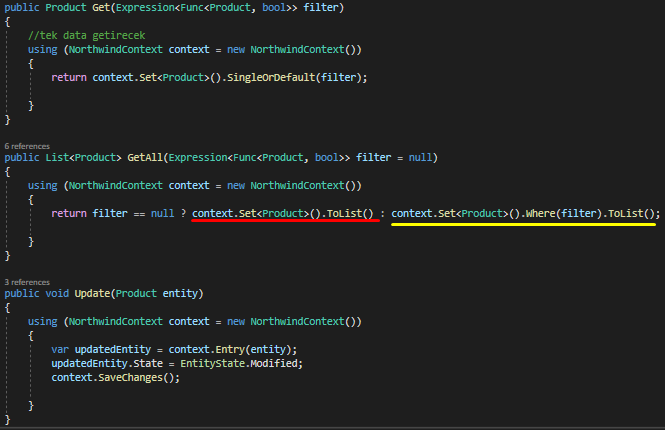
Using CSharp’a özgü güçlü bir yapı. Buradaki Using IDisposable pattern implementation of c# ( CSharp’ın Idisposable pattern denilen bir implementasyonu demek )

Entity Framework’ta Using içerisine yazılan işlemlerle ekleme, silme, güncelleme operasyonları yazarız.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

EfProductDal operasyonlarımızı yazmaya devam ediyoruz.



GetAll( ) operasyonu diyelim ki filtre göndermemişse veri kaynağındaki yani veritabanında ilgili tablodaki tüm datayı listelesin. Ama filtre vermişse filtreyi uygula ona göre datayı listele.

**Kırmızı çizgi** Products’a yerleş. Veritabanındaki bütün tabloyu listeye çevir ve onu bana ver. **;** Eğer filtre null ise ilk kısım çalışır, **Sarı çizgi ;** değilse diğer kısım çalışır.

**DataAccess katmanımın yeni hali**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Business katmanına geldiğinde hiçbir şey değişmez. Çünkü standartlara uyduk. Daha önce InMemory ile çalışıyorduk. Şimdi Entity Frameworkla çalışıyoruz. İş katmanımda herhangi bir şeyi değiştimeme gerek kalmadı. Businessımız değişmedi, iş kurallarımız sadece veriye erişim yöntemimiz değişti.

**Business/Abstract/IProductService.cs :** Businessta IProductService te bir tane daha liste yazalım GetAllByCategoryId. GetAll( ) tümünü getir. CategoryId ile tümünü getir. Bir tane daha liste yazalım. GetByUnitPrice fiyata göre getir.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Business/Abstract/ProductManager.cs :** Businessta interfacede listeleri yazdık Concrete Managerda işlet onu.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

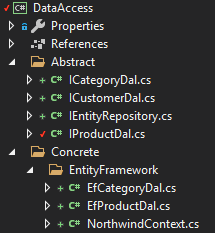
**Console/Program.cs :** Console da GetaAll, GetAllByCategoryId, GetAllUnitPrice çağırıp çalıştırdığında belirtilen filtreye göre gelir.

SOLID prensibinin O harfidir. Open Closed Principle--Yaptığın yazılıma yeni bir özellik ekliyorsan mevcuttaki hiçbir koduna dokunamazsın.

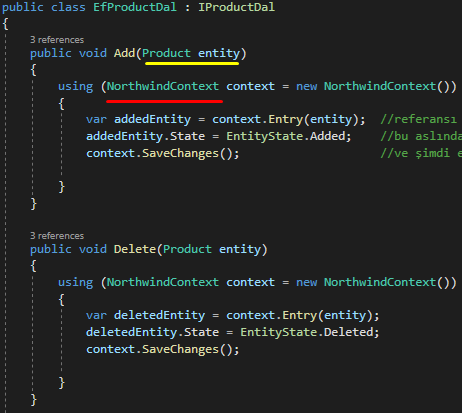
metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Projemizi geliştirelim 9.Hafta :



Data Access katmanımızın Abstract klasöründe IEntityRepository altyapısını kurmuştuk. ICategoryDal, ICustomerDal, IProductDal sınıflarımızda aynı operasyonlar mevcuttu değişen tek şey tipti ve biz bunu bir jenerik sınıf(IEntityRepository) oluşturarak diğer sınıfların bu jenerik sınıftan kalıtım almasını sağlamıştık. Data Access katmanımızın Concrete klasörüne baktığımızda interfacelerin somutlaştırılmış hali, implemente ettiğimiz sınıflar EfProductDal ve EfCategoryDal içerik aynı, operasyonlar aynı o zaman burada da jenerik yapısını kullanabiliriz.



Burada değişeceğini bildiğimiz kısımlar

1-)Çalıştığını bildiğimiz entity. Product,Category,Customer,Employee vs.

2-)Contextimiz değişken. Mesela CarRental veritabanına bağlandığımızda CarRentalContext olur orası.

Bunları base bir sınıfa temel bir sınıfa taşısak ve onu jenerik yapsak, bana çalışacağım tipi ver(entity) ve çalışacağım context'i ver desek jenerik olarak. Biz onları bütün varlıklar için kullansak. Bu ortaya farklı şeyler çıkartıyor. Bundan önce yapmamız gereken ;

Yazılım geliştirme projelerinde framework katmanı oluştururuz. CORE adında katman oluşturarak ortak kodları bu katmana yazarız bundan sonra bütün projelerde bu katmanı kullanabilir.

Core katmanına evrensel kodlarımızı yani tüm projelerde kullanılacak kodlarımızı koyabiliriz. O yüzden bu Core katmanında hangi katmanla ilgileneceksek o katmanla ilgili klasör oluşturuyoruz. Ben altyapımda veri erişimlerde kullanmak üzere bir klasör oluşturuyorum. Yani DataAccess katmanına hizmet edecek kodları bunun içine koyuyorum. Bizim için IEntityRepository tam olarak bu işe yarıyor. Core katmanını CarRental projesinde de aynen kullanabiliriz çünkü evrensel kodlar orada.

**Core/DataAccess :** Veri erişimi ilgilendiren evrensel kodlar buraya konulur. IEntityRepository classını buraya taşıdık.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Core/DataAccess/IEntityRepository.cs :** Burada IEntity’in altını çizdi. Burada Entities katmanını Core katmanına referans verilebilir olarak düşünebilirsiniz. Ama Core katmanına Entities katmanını referans verirsek bu Northwind projesindeki Entities’e bağımlı olur. Bu doğru değil. Peki ne yapacağız ? Entities katmanı IEntity sınıfını kullanıyor. IEntity classını Entities katmanı içerisinde bıraktık ama IEntity Northwind projesine özel bir şey değil ki. IEntity şu anlama geliyordu bir classa IEntity dediğimizde o class bir veritabanı tablosu görevi görüyordu. Yani bu IEntity IEntityRepository gibi Core katmanına özgü bir şey. Çünkü ben o IEntity’i her projede kullanabilirim.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Core katmanının mantığı; istediğim katmanı burada ayrı ayrı klasörleyip, implemente edeyim.

**Core/Entities/IEntity.cs :** Entities klasörü oluşturarak IEntity classını buraya taşıdık. Namespace’ni değiştir. Şuan yapılan işlem iki nesneyi Core’a taşıdık. Core benim evrensel katmanım. Yani sadece Northwind de değil, bütün .NET projelerinde Core katmanını kullanabilirim anlamına gelir. Core katmanı diğer katmanları referans almaz!!!!

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Pembe çizgi ;** Artık anlamsız onu silelim.

**Sarı çizgi ;** IEntity artık Core katmanından geliyor onu ekleyelim.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

IEntityRepository sınıfı düzeldi, önceden DataAccess katmanındaydı ve IProductDal, ICategoryDal sınıfları ondan kalıtım ediyordu. Ve o sınıflarda şuan altı çizili. Şimdi bu sınıflar IEntityRepository kullanacaksa nerede bu sınıf ? Core katmanında o zaman bu sınıfları taşıyan DataAccess katmanı Core katmanına bağımlı. O zaman referans veririz.

**DataAccess katmanına sağ tıkla >> Project Reference >> Core katmanını ekle**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Bu işleme Code Refactoring (Kodun İyileştirilmesi) denir.

Aynı şekilde IEntity classını da Core katmanına taşıdık. Önceden Entities katmanındaki nesneler ondan kalıtım alıyordu. Bunları da iyileştirelim. Using core kütüphanemizi de ekleyelim.

**Entities katmanına sağ tıkla >> Project Reference >> Core katmanını ekle**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Core katmanını oluşturduk, her şeyi düzelttik. Gelelim EfProductDal, EfCategoryDal classlarını jenerik yapıya çekmeye.

**Core katmanına gel >> DataAccess sağ tıkla >> Add >> New Folder >> EntityFramework adını ver**

Bahsettiğim sınıflar DataAccess katmanının EntityFramework klasöründeydi , Core katmanında EntityFramework adında klasör oluşturdum. EntityFramework’ü kullanarak evrensel bir kod yazacağız. O yüzden klasörlüyoruz.

**Core/DataAccess/EntityFramework/EfEntityRepositoryBase.cs :** EntityFramework’ü kullanarak bir Repository Base’i oluştur demek. Bu sınıf EfProductDal,EfCategoryDal,EfCustomerDal sınıflarında operasyonların ortak olması sadece tip ve contextin değişmesi sebebiyle oluşacak olan Jenerik sınıfım. Jenerik sınıflarda hatırla <T> yazıyorduk. Farklı isimlerde istediğin kadar T yazabilirsin. Sadece tip ve context değiştiği için TEntity, TContext diyoruz. Bu şu demek bana bir tane Entity tablo ver ve Context tipi ver ben ona göre çalışacağım demek bu yapı.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

EfProductDal da operasyonlarda hatırla Entity Framework kodu yazdık. Burada da o operasyonları yazacağız, entity framework kodu kullanmamız gerek, o zaman Core katmanına da DataAccess katmanına daha önceden eklediğimiz gibi Entity Framework sistemini eklemeliyiz.

**Solution sağ tıkla >> Manage Nuget >> Browse >> .NetFrameworkCore.SqlServer >> Core işaretle**

metin, ekran görüntüsü, siyah, ekran içeren bir resim

Açıklama otomatik olarak oluşturuldu

Kurulumdan sonra jenerik sınıfımızı yazmaya devam edelim. Hatırla jenerik kısıtlaması yapmıştık, programcı doğru bir şekilde o yapıyı kullanabilsin diye. Generic Constraint yapalım, çeşitli kurallar koyalım. T’yi sınırlandıralım.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Sarı çizgi ;** Jenerik yapısı.

**Kırmızı çizgi ;** Where koşulu ile TEntity class olsun diyoruz. Tip olarak int, string filan olamasın.

* **where TEntity : class** ; class : buradaki class referans tip olabilir demek.

**Turuncu çizgi ;** Where TEntity : class olduğunda ise yine Category classı yerine bir class ismi yazılsa hata vermez o yüzden sadece IEntity classını şartlayacağım ki bana sadece nesnelerimi versin kısıtlasın. Nesne classlarının ortak özelliği hepsinin IEntity olması.

* **where TEntity : class, IEntity**

Bu şu demek TEntity bir referans tip olmalı (class) ve ya IEntity ya da IEntityden implemente olan bir   
şey olabilir.

* **IEntity**; IEntity olabilir veya IEntity implemente eden bir nesne olabilir.

Bu süreçlerden sonra EfCategoryDal : EfEntityRepositoryBase<Category, Context> Category yerine customer, product, category ve IEntity yazılabilir sadece. Ama ben IEntity yazmak istemiyorum soyut nesne işimi görmüyor, devre dışı bırakmak istiyorum.

* **where T:class,IEntity,new( ) ;** bu şekilde yazıldığında devre dışı bırakabilirim nasıl ?

new : new'lenebilir olmalı. IEntity interface newlenemez. Customer, Product, Category somut nesne newlenebilir.

**Pembe çizgi ;** Bunun Context olabilmesi için EntityFramework DbContext’i inherit etmesi gerekiyor.

* **where TContext : DbContext** ; buraya her classı yazamazsın. DbContext’ten inherit etmesi lazım. Burada da newlenebilir yapıyı koyuyoruz ki sadece NorthwindContext gibi classlar gelsin.

Şimdi DataAccess katmanında Abstractta ICategoryDal bir IEntityRepository’di. Concrete klasöründeki EfCategoryDal ‘ da bir ICategoryDal’dı. O zaman EfCategoryDal gibi ortak operasyonlara sahip sınıfları taşıyan jenerik sınıfım EfEntityRepositoryBase bir IEntityRepository’dir.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

EfEntityRepositoryBase sınıfını EfProductDal, EfCategoryDal sınıfları için oluşturmuştuk. EfProductDal da yazılan operasyonları al bu classa yapıştır. Product yazan yerlere TEntity, NorthwindContext yazan yerlere de TContext yazarak jenerik hale getirelim.

metin içeren bir resim

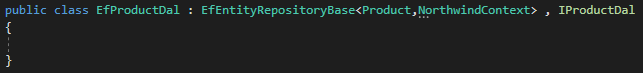
Açıklama otomatik olarak oluşturuldu

**DataAccess/Concrete/EntityFramework/EfProductDal.cs :**

EfProductDal daki operasyonlarımızı aldıktan sonra bize kızıyor ama artık ben EfEntityRepositoryBase jenerik sınıfımı bu classlar(EfProductDal,EfCategoryDal,EfCustomerDal) için oluşturduğumdan buraya yazmalıyım.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu



Şuan EfProductDal’ da bütün veritabanları operasyonlarını yazmaya hazırız.

**Hadi bir tane ORDER nesnesi ekleyelim. Sistemin nasıl çalıştığını görelim.**

1. **Entities katmanına gidip Entity eklenir.**

**Entities/Concrete/Order :** Biliyorum ki bu benim için bir veritabanı nesnesi o yüzden diyorum ki sen bir IEntitysin.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

1. **DataAccess katmanında Abstract klasörüne eklenen Entity’in interface’i oluşturulur.**

**DataAccess/Abstract/IOrderDal.cs :** IOrderDal sen bir IEntityRepositorysin.



1. **Bu interfacesi somutlaştır. EntityFramework’ünü hazırla.**

**DataAccess/Concrete/EntityFramework/ EfOrderDal.cs :** EfOrderDal sen bir IEntityRepositorBase ve IOrderDalsın.



1. **Kalan son şey Order nesnesini EntitFramework’e belirtip veritabanıyla ilişkilendirmek.**

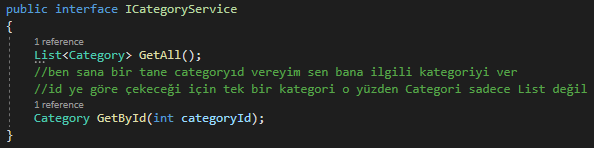
**DataAccess/Concrete/EntityFramework/NorthwindContext.cs :** Order ile Orders tablosunu bağlama.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Her şey tamam. Business katmanımıza Category nesnesini dahil edelim.

**Business/Abstract/ICategoryService.cs :** Category nesnesinin iş sınıfı, interface. Kategoriyle ilgili dış dünyaya neyi service etmek istiyorsak o operasyonları yazıyoruz. GetAll ve GetById kullandık.

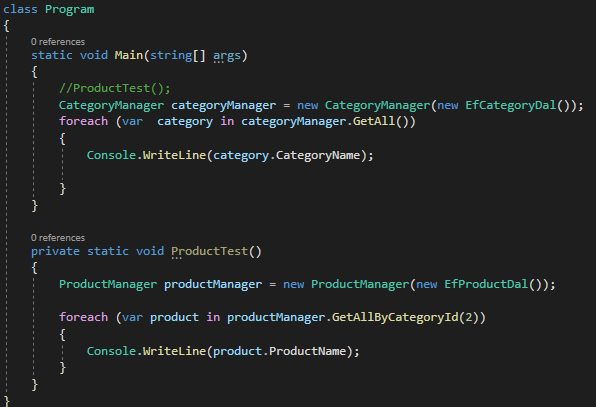


**Business/Concrete/CategoryManager.cs :** ICategoryService interface sınıfını somutlaştırıyoruz. CategoryManager sen bir ICategoryService implementasyonusun. İmplemente ederek metotları çağırırız. 2 metodum var dış dünyaya servis etmek istiyorum, iş kodlarını buraya yazacağım o yüzden beni implemente et diyor.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**ConsoleUI/Program.cs :** Category nesnesini GetAll metoduyla kategori isimlerini yazdırdık.



**DTO (DATA TRANSFORMATİON OBJECT)**

Bir e-ticaret sistemini düşünelim ürünün ismi ve kategorisinin ismi yanyana ekranda gözükür. Ürün ismi product tablosunda, kategori ismi kategori tablosundadır ama ekranda nasıl yanyana gelir ? Dto sayesinde

Örneğin ; veri tabanından çekilen veride isim, soyisim ayrı propertylerdedir. Sizin tek property olarak kullanmanız gerekir ya da iki farklı veri tablosundan maplediğiniz iki modelden gelen propertyleri harmanlayarak bir model oluşturmanız gerekebilir. İşte veri tabanından mapplenen modelleri uygulama içerisinde kullanacağınız modelle aktardığınızda bu modeller dto olarak adlandırılır.

ProductDetailDto sen bir IDto’sun. Neden IEntity değil de IDto çünkü bir veritabanı tablosu değil, birden fazla tablonun joini olabilir. Context’e gidip eklemesin diye IDto dedik çünkü tablo değil, tabloya karşılık gelmiyor. Belki birkaç tablonun birkaç kolonuna karşılık geliyor olabilir ama tek başına bir tablo değil.

**Entities/DTOs/ProductDetailDto.cs :** Entities katmanına DTOs diye klasör oluşturalım ve içine ProductDetailDto isimli class oluşturalım. Bahsettiğimiz tablolarda bulunan özellikleri buraya yazalım.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Core/Entities/IDto.cs :** IDTO interface classı evrensel kodları içereceğinden Core’un içindeki Entities katmanına ekliyoruz.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**DataAccess/Abstract/IProductDal.cs :** Ürünlerin detaylarını getir listele diye metot yazalım.

metin içeren bir resim

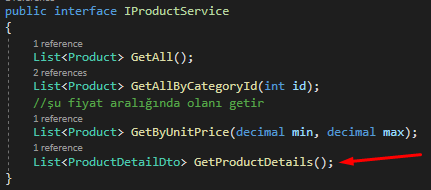
Açıklama otomatik olarak oluşturuldu

**DataAccess/Concrete/EntityFramework/EfProductDal.cs :** IProductDal da bir metot ekledik burası da onun somut sınıfı metodu implemente etmeliyiz o metodu somutlaştıyoruz. Tabloları join etmeliyiz ki Dto görevi yerine getirsin. Products tablosu ile Categories tablosunu birleştiriyoruz. İki tabloda da CategoryId mevcut olduğundan bu özelliği eşitliyoruz(equals).ProductDetailDto classındaki özellikleri; join ettiğimiz tablolardaki kolonlarına denkleştiriyoruz. Yani ProductDetailDto classındaki ProductId özelliğini, NortwindContextteki Products tablosuna eşitliyoruz. ProductDetailDto classındaki CategoryName özelliğini, NortwindContextteki Categories tablosuna eşitliyoruz.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Business/Abstract/IProductService.cs :** Metodu iş sınıfımızın Service interfacesinede ekleyelim.



**Business/Concrete/ProductManager.cs :** İnterface sınıfımızı somutlaştıralım. Yeni metot ekledik interface sınıfımıza burada da implemente etmeliyiz. Return diyerek listenin tamamını döndürüyoruz.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**ConsoleUI/Program.cs :** Products tablosu ile Categories tablosunu join ederek ProductDetailsDto classındaki özelliklere ilgili kolonları eşitledik. Ve burada artık ProductName ve CategoryName’i yanyana yazabiliyoruz. Dto mantığı bu şekildedir.

metin içeren bir resim

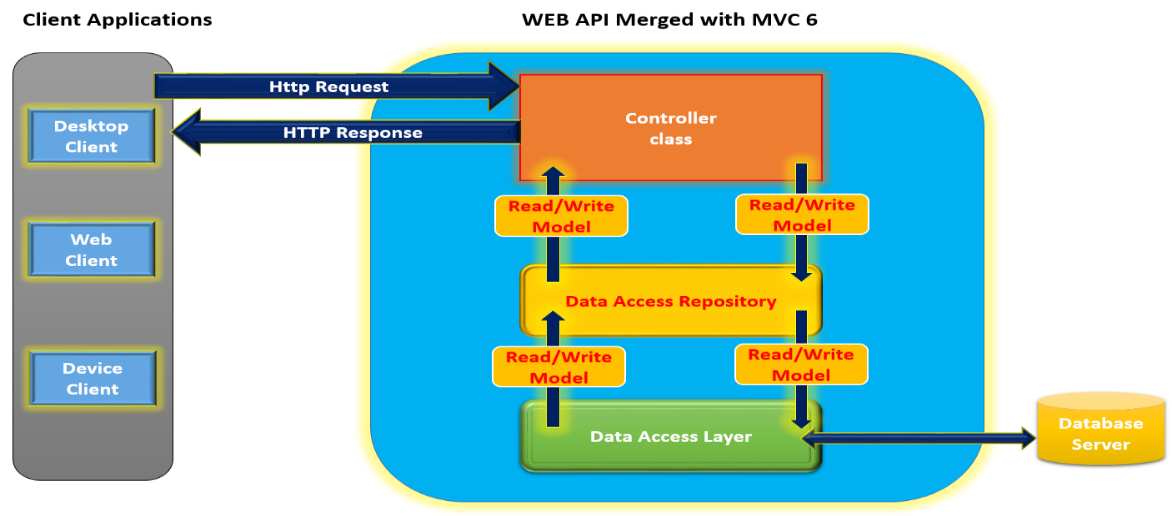
Açıklama otomatik olarak oluşturuldu

Projemizi geliştirelim 10.Hafta :

Üç katmanımız vardı. Data Access, Business, Console. Günümüzde de farklı ortamlarda da yazılım geliştirmek durumundayız. Farklı projelerde, farklı ara yüzlerde olabilir mobil mesela (Ios,Android) Ios altyapısında olan bir mobil uygulama gibi. .NET’le (Android, IOS) Business ‘i direkt görüştüremeyiz. Çünkü bunlar farklı ortamlar, bunların birbiriyle görüşme şansı yok. Farklı sistemlerin birbirini anlayabilmesi için bir teknoloji kullanıyoruz. Farklı sistemlerin birbirini anlayabilmesi için Service odaklı mimariler dediğimiz bir katman daha yazarız. Günümüzde bunun karşılığı olarak API dediğimiz altyapıları en yaygın biçimde kullanıyoruz.

Web Api ne yapıyor ? .Net kodu yani C# kodu Ios uygulamalar için, Flutter uygulamalar için, Angular için bir anlam ifade etmiyor. Console için yazdığımızda bir anlam ifade eder. Çünkü Console bir .Net projesi. Ama bir Angular, bir React, bir Android, bir Ios uygulaması; .Net kodunu hiçbir şekilde anlamaz. O yüzden bunu anlasın diye yazılım dünyasında bir standart var. Web API Restfull denilen standart mimariyle karşımıza çıkıyor. Bu da bir formatla (json) denilen veri formatıyla karşımıza çıkıyor.

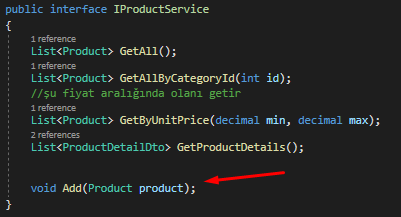
Yani Business’i ona bağlı olarak DataAccess’i API katmanı vasıtasıyla dış dünyaya açıyoruz.



Web Api oluşturmadan önce; bu farklı clientlarla (console,react,angular,ios,android ) bunların her biri farklı uygulamaların clientı yani müşterisi, çalışacağımız zaman bu clientlerin Restfull service istek yapmasını bekliyoruz bu isteğe Request(İstek) denir. Bizim verdiğimiz Yanıt ise Response olan kelimedir. Request ve Response dediğimiz bu süreci daha iyi yönetebilmek için profesyonel bir altyapı yükleyeceğiz sisteme.

**Business/Abstract/IProductService.cs :**

Şimdi IProductService’te bir CRUD operasyonu daha ekleyelim ADD( ). Ekleme işlemi liste döndürmeyecek void. Bu void ya bir şey döndürmüyor. Ama ben ekleme işleminin başarılı olup olmadığını(true ya da false) ve aynı zamanda işlemle ilgili bilgilendirme yapmak istiyorum   
kullanıcıya. ( Ürün eklendi veya ürün şundan dolayı eklenemedi gibi. ) Bunları döndürmem lazım sistemde ama zaten metot List< > ya liste döndürüyor başka bir şeyi daha nasıl döndürücek ? Bir metotta sadece bir değer döndürebilirsin liste döndürürsün mesela ama aynı anda birden fazla bir   
şey döndürmek istiyorsan encaplücation yapısını kullanırız.



Void yerine IResult diyelim. Mesela bu Result bir class olsun içerisinde de 2 tane property olsun. Bir tane işlem sonucu(true ya da false), bir de işlem mesajı 2 tane alan ekleyebilmek istiyorum.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Core katmanının içerisine Utilities adlı bir klasör oluşturalım. Onun altına Results klasörünü oluşturalım. Ve Results klasörüne de IResult interfacesini, interfacenin somutlaştırılmış sınıfı Result sınıfını oluştur.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Core/Utilities/Results/IResult.cs :** İçerisinde bir tane işlem sonucu, bir tane de kullanıcıyı bilgilendirmek adına mesaj olsun. Result sonuç demek, Success başarılı mı başarısız mı, Message yapmaya çalıştığın işlem başarılı true o zaman ürünü ekle gibi bilgilendirme.

Hatırla iki property vardı Getter, Setter. Getter okumak için, Setter yazmak için. Get diyoruz sadece.

metin, iç mekan içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Core/Utilities/Results/Result.cs :** IResult'un somut sınıfı. Result sen bir IResultsun. Getter return demekti o yüzden buraya ne yazarsak return eder.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Business/Concrete/ProductManager.cs :** IProductService’te Add operasyonunu ekledik ve IResult yaptık. ProductManager sınıfı da IProductService interface sınıfının somut hali olduğu için imzayı taşımalı. İmplemente ediyoruz, Add operasyonu buraya da gelir. Add bize kızıyor; bir şey döndürmek gerektiğini söyledin ama bir şey döndürmüyor çünkü burada return var artık. Burada void yerine IResult yazdık ve return de IResult döndürmeliyiz ama Result da bir IResult olduğu için onu yazabiliriz.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Add operasyonu iş kodlarını geçti ekleme işlemini gerçekleştirdi o zaman diyoruzki sen onu döndürebilirsin. return Result(resultu döndür). Orada ekleme işlemini gerçekleştirdikten sonra return result döndürme işleminde Result içerisine (true,”Ürün Eklendi”) yazdırmak istiyorum. Bunu yapabilmenin yöntemi Constructor’dır. Çünkü iki tane parametre yolluyoruz ama parametreler mevcut değil. Result hata verdi çözüm olarak Result sınıfına constructor yapısını kur dediğimizde sorun çözülür.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Result sınıfında constructor yapısını kurdu. Burada constructorun içini boş bırakalım geri kalan alanları silelim çünkü eşitlemeyi başka türlü yapacağız. V1 yerine success, v2 yerine message yazalım.

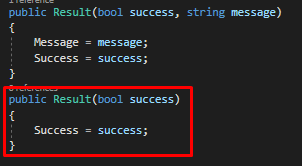
metin, ekran, telefon, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Constructor’ın içerisini dolduralım. Aşağıdaki Message, parametredeki message’a eşitleyelim. Peki Message hani getti, set edilemezdi ? Getter read only’dir. Read onlyler Constructor da set edilebilirler! Zaten Constructor dışında Set etmeyeceğiz, o yüzden bu yapıyı kurduk getter dedik setter koymadık ki tamamen constructor yapısıyla kullanalım. Setter konulsaydı programcı kafasına göre kullanabilirdi (mesela result.success) ama biz onu sınırlandırarak,yazmamasını sağlıyoruz. Tip dönüşümlerini (return) constructorla yapacağız. Kodların okunurluğu standart olsun diye bu yapıyı standartize ediyoruz.

Resultta bu yapıyı kurduk Add metodunda sonuç başarılıysa message verecek. Ama ben mesaj yazdırmak istemiyorum sadece sonuç başarılı mı onu döndürmek istiyorum kullanıcıya o zaman bir metot daha yazarız.



Bu şekilde yazdığımızda da kırmızı kutucuk çalıştırıldığında true,false döndürür ama üstteki metot çalıştırıldığında hem true,false hem de message döndürür. Üstteki metot alttakinin işlevini de görüyor o zaman üstteki metot içerisindeki success bilgisini kırmızı kutucuk içindeki metot set etsin diyoruz.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Ama üstteki metot çalıştırıldığında alttaki metot da çalışsın o zaman this yapısını kullanıyoruz. C# da This demek classın kendisi demek. Result’u kastediyor. Her ikisi de duruma göre çalışır bu haldeyken.

**Kırmızı Çizgi ;** Result’un tek parametreli constructoruna success’i yolla.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Şimdi bu sistemi biraz daha güzelleştirelim. Result yerine Success Result ya da Error Result döndürebilsin yapıyı geliştiriyoruz.

**Core/Utilities/Results/SuccessResult.cs :** Sen bir Result’sun. SuccessResult sadece mesaj göndersin, Resultta ya success, ya da hem success hem mesaj göndersin. Base result demek. Base yazdığımda bana iki seçenek sunuyor ya 1. yi gönderebilirsin ya da 2. Gibi yukarıda yaptığımız tam olarak bu.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

ProductManager da Add operasyonu bu şekildeydi ben artık kullanıcıya burada true döndürmeyeyim, burası SuccessResult olsun ve sadece message yazsın, true ifadesini SuccessResult sınıfında döndüreyim demektir.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Kullanıcıya mesaj vermek istemiyoruz sadece true yollayacağız o zaman ne olacak? O zaman base ‘i tek parametreli olanı çalıştır.

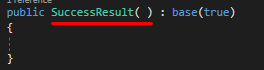
metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Kısacası ProductManager da şu şekilde yazdığımızda SuccessResult sınıfına gider SuccessResult’u default olan base de true olan yani Result sınıfında. Result sınıfında onu bulup çalıştırıyor.

**ProductManager** **SuccessResult**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu 

**Result**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Madem SuccessResult sınıfımız var bir de ErrorResult sınıfımız olmasın mı ? Add operasyonunu şu şekilde ErrorResult dahil ederek yazmayalım mı ? ErrorResult sınıfı oluşturalım.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Core/Utilities/Results/ErrorResult :** HataSonucu Sen bir Resultsun. True değil false döndürüyorsun. Dolasıyla Add operasyonunda if içerisinde false değer , değilse true değer işlevlenecek.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

ProductManagerda add içerisindeki şu kötü yazılımdan kurtulalım. Altı çizili stringler literatürde Magic strings olarak adlandırılır. Bunun için çözüm geliştirelim.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

O zaman Business katmanına bir klasör ekleyelim. Constant(Sabit) bu klasör proje sabitlerini içerir. Mesela metinler, mesajlar, IEnumerable içerir.

**Business katmanımın yeni hali ;**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Business/Constants/Messages.cs :** Bu static bir constant. Mesajları yazdık.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Artık ProductManager da magic stringsleri temizleyerek daha iyi bir yazılım elde edebiliriz.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

ProductService sınıfına gelelim. Add operasyonunda void için IResult demiştik. Listler için burada da bir liste döndürdüğümüz için IDataResult diyelim. İçerisinde List<Product>(list of product) olan data sonucu. IDataResult hem işlem sonucunu, hem mesaj, hem de döndüreceği List<Product>(list of product) içeren bir yapı görevi görecek.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

O zaman Core/Utilities/Results altında bir IDataResult sınıfı oluşturalım.

**Core/Utilities/Results/IDataResult.cs :** Hangi tipi döndüreceğini bana söyle. IDataResult sen bir IResultsun. IDataResutta IResult gibi mesaj içersin, işlem sonucu içersin ama aynı zamanda data içersin. Mesajla, işlem sonucunu zaten IResult içeriyordu zaten o zaman IResult kalıtım veririz. Onun haricinde bir de T türünde data olacak.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Yani kısacası IDataResult ama jenerik, hangi tiple çalışacağımı söylemem gerekiyor çünkü her metot başka bir şey döndürüyor(ürün,kategori listesi,dto gibi). Aynı zamanda IResulttaki yapılar var bir de ek olarak sen de Data var. ProductService gel şimdi T sini ver. Önceden List<Product> döndürüyorduk artık diyorumki ben List<Product> ile birlikte mesajı da, işlem sonucu da döndürücem.

GetById Product döndürüyordu T o zaman Product olur. GetAll List<Product> döndürüyordu T o zaman List<Product> döndürür. Bunlara dikkat et.

**Business/Abstract/IProductService.cs :**

metin, ekran, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Core/Utilities/Results/DataResult.cs:** Bir DataResult sınıfı oluştur Core da. Result sınıfından farkı Data içermesi. DataResult sen hem Resultsun hem de IDataResultsun. T tipinde Data döndür.

metin, iç mekan, ekran, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Business/Concrete/ProductManager.cs :** IProductService te IDataResult yaptık yapıyı burada düzeltmeliyiz aynı şekilde. Artık DataResult döndürmek istiyorum.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Ben DataResult döndürüyorum çalıştığım tip List<Product> **(Yeşil çizgi)**, döndürdüğüm data**(Kırmızı çizgi)**, işlem sonucum**(Sarı çizgi)**, mesajım**(Pembe çizgi)**

Nasılki Result’un SuccessResult,ErrorResult’unu gerçekleştirdik aynı şekilde süreci daha kolaylaştırmak için DataResultunda Success ve Error lerini gerçekleştirebiliriz.

**Core/Utilities/Results/SuccessDataResult.cs:** Sen bir DataResultsun.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Core/Utilities/Results/ErrorDataResult.cs:** Sen bir DataResultsun.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Business/Concrete/ProductManager.cs :** Şimdi ProductManagerdaki GetAll operasyonunu SuccessResult ve ErrorResult kullanarak yazalım daha düzgün kod olsun.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Kırmızı Çizgi;** Saat 22.00 olduğunda MaintenanceTime(Bakım) yapılsın.

Zaten SuccessResult true ya yazmayalım onu ürünler eklendi yerine de Message yazalım. Mesajları Generate Field ile oluştur.

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Artık ProductManagerdaki tüm operasyonları düzenleyebiliriz. Tüm metotları GetAll gibi yapınız.

**ProductManager sınıfının son hali;**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldumetin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**Messages sınıfının son hali;**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

**ConsoleUI/Program.cs:** ProductDetails hata verdi çünkü o IDataResult döndürüyor artık.

metin, iç mekan, ekran, ekran görüntüsü içeren bir resim

Açıklama otomatik olarak oluşturuldu

İf-else yapısıyla iyileştirelim kodu. Bu kategoride sana göstereceğim ürün varsa if yoksa else çalışır demek.

**Program.cs sınıfının son hali ;**

metin içeren bir resim

Açıklama otomatik olarak oluşturuldu

Projemizi geliştirelim 11.Hafta :