Katmanlar

Harika bir başlangıç! Hemen kısa ve öz şekilde anlatıyorum. ⚙️

**🔹 Katmanlı Mimari Nedir?**

Yazılımı **parçalara ayırarak** düzenli ve yönetilebilir hale getiririz. Her katmanın kendi görevi vardır:

**1️⃣ Entity Layer (Varlık Katmanı)**

🔹 Veritabanındaki tabloya karşılık gelen sınıflar.  
🔸 Yani sadece "veri nedir?" sorusuna cevap verir.

📌 **Hayattan örnek:**  
Bir kargo firmasında **müşteri kartı** sadece ad, soyad, adres içerir. İş mantığı veya görsel kısmı yoktur.

public class Customer

{

public int Id { get; set; }

public string Name { get; set; }

}

**2️⃣ Data Access Layer (Veri Erişim Katmanı)**

🔹 Entity'leri kullanarak veritabanına **ekleme, silme, güncelleme** işlerini yapar.  
🔸 Ama iş mantığı içermez.

📌 **Hayattan örnek:**  
Müşteri bilgilerini **veritabanına yazan personel** sadece kayıt yapar, karar vermez.

public class CustomerDal

{

public void Add(Customer customer)

{

// Veritabanına ekleme kodları

}

}

**3️⃣ Business Layer (İş Mantığı Katmanı)**

🔹 Kurallar burada. Hangi müşteri eklenebilir, hangi sipariş onaylanmalı gibi kararlar burada alınır.

📌 **Hayattan örnek:**  
Yöneticiler “şu müşteri uygundur, bu değil” der. Veriyi alır, kontrol eder, sonra işlemi başlatır.

public class CustomerManager

{

CustomerDal dal = new CustomerDal();

public void AddCustomer(Customer customer)

{

if (customer.Name.Length > 2)

{

dal.Add(customer);

}

}

}

**4️⃣ User Interface / Presentation Layer (Kullanıcı Arayüzü)**

🔹 Kullanıcının etkileşimde bulunduğu kısım: butonlar, formlar, web sayfası vs.  
🔸 Arka plandaki mantığı bilmez, sadece tetikler.

📌 **Hayattan örnek:**  
Kargo şubesindeki personelin kullandığı ekran. Ne olur ne olmazı bilmez, sadece formu doldurur ve gönderir.

**5️⃣ Repository Design Pattern**

🔹 Veri erişimini soyutlar. Kod tekrarını azaltır, test etmeyi kolaylaştırır.

📌 **Hayattan örnek:**  
Farklı şehirlerdeki şube kayıt işlemlerini aynı standartta yapan bir şablon gibi düşün.

public interface IRepository<T>

{

void Add(T entity);

void Delete(T entity);

List<T> GetAll();

}

**🔹 Abstract ve Concrete Nedir?**

Bunlar genelde **katmanlı mimaride** kullanılır ve **katı bir yapı sağlar**.  
Ana amaç: **bağımlılığı azaltmak** ve kodu daha esnek hale getirmek.

**1️⃣ Abstract (Soyut Klasör)**

🔹 İçinde sadece **interface** veya **abstract class** olur.  
🔹 Ne yapılacağını tanımlar ama nasıl yapılacağını söylemez.  
🔸 “Kurallar burada yazılı.”

📌 **Hayattan örnek:**  
Ehliyet sınavı: Sürücü ne yapacağını bilir ama nasıl kullanacağı farklıdır.  
(Arabayı sür, ama hangi marka araba fark etmez.)

public interface ICustomerService

{

void Add(Customer customer);

}

**2️⃣ Concrete (Somut Klasör)**

🔹 Abstract’taki interface’leri **gerçekleştirir (implement eder)**.  
🔹 Yani “nasıl yapılacağı” burada yazılır.

📌 **Hayattan örnek:**  
Biri BMW sürer, diğeri Tofaş. Kurallar aynı (Abstract), uygulama farklı (Concrete).

public class CustomerManager : ICustomerService

{

public void Add(Customer customer)

{

// Kurala uygun gerçek kod

}

}

**💡 Neden Böyle Ayırıyoruz?**

* Kodun test edilebilirliğini artırır.
* Daha kolay bakım yapılır.
* Katı bir yapı kurulur = profesyonellik artar.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |

**🎯 AMAÇ: İki tablo arasında ilişki kurmak.**

Senin örneğinde:

* **Blog → Category’ye bağlı** (her blog’un bir kategorisi var)
* **Category → birden fazla Blog’a sahip** (bir kategoriye birçok blog yazısı ait olabilir)

Bu ilişki: **Bire–Çok (One-to-Many)** ilişkidir.

**🔗 Bire–Çok (One-to-Many) İlişki**

**🔹 Blog tarafı (Çok olan taraf)**

public int CategoryID { get; set; } // Foreign Key

public virtual Category Category { get; set; } // Navigation Property

Bu satırlar demek oluyor ki:  
➡️ Bu blog'un ait olduğu bir kategori var.  
➡️ virtual yazınca EF, lazy loading (tembel yükleme) yapabiliyor. Yani **kategori bilgisi gerektiğinde çekilir**, performans için güzel.

**🔹 Category tarafı (Bir olan taraf)**

public ICollection<Blog> Blogs { get; set; }

Bu, **bir kategoride birçok blog olabilir** demek.  
ICollection<Blog> diyerek ilişkiyi koleksiyon (liste gibi) tutuyorsun.

**🤔 Neden Blogs diye çoğul?**

Çünkü bu **birden çok Blog'u tutan** bir koleksiyon.  
Eğer Blog yazsaydın hata vermezdi ama ilişkiyi **yanlış anlar veya karıştırırdı**.  
**İsimlendirme kuralı olarak** çoğul kullanılır: Blogs, Posts, Comments gibi.

“Category'nin birçok Blog'u var” → Blogs

**🧠 Neden aynı isimde olmak zorunda?**

Aslında **zorunda değilsin** ama:

* EF, **adlandırma kurallarına göre** eşleştirme yapar.
* Eğer adlar uyuşmazsa ya hata verir ya da Fluent API ile manuel ayarlaman gerekir.

Sen bu örnekte hem CategoryID hem Category property'lerini doğru yazmışsın, bu yüzden EF otomatik ilişkilendirir.

**🧪 Hayattan Örnek**

Bir markette ürünler var. Her ürün bir reyonda durur.  
“Blog” ürün gibidir, “Category” reyon.  
Bir reyonda birçok ürün olur (→ ICollection<Blog>),  
Her ürün sadece bir reyondadır (→ Category)

**🔧 Kodun Tam Haliyle Eşleşme:**

**Blog.cs**

public int CategoryID { get; set; } // Foreign Key

public virtual Category Category { get; set; } // Navigation Property

**Category.cs**

public ICollection<Blog> Blogs { get; set; } // One-to-many

Context Dosyası

public class Context:DbContext

{

public DbSet<About> Abouts { get; set; } // sonuna s almayan kısım projemde duracak S alan kısım ise veritabanımda olacak

public DbSet<Admin> Admins { get; set; }

public DbSet<Author> Authors { get; set; }

public DbSet<Blog> Blogs { get; set; }

public DbSet<Category> Categories { get; set; }

public DbSet<Comment> Comments { get; set; }

public DbSet<Contact> Contacts { get; set; }

}

<connectionStrings>

<add name="Context" connectionString="data-source=BARANPC\SQLEXPRESS;initial catalog=BlogDb;integrated security=true;" providerName="System.Data.SqlClient"/>

<!-- bu kısım veritabanı bağlantısı için ama sadece mvcde var corede yok ve system webin altına yazılmalı web.configde -->

<!-- burada name veritabanına gidecek dosyalar data source veritabanı ismimiz catalog veritabanına kaydolacak ismi security ise güvenlik -->

</connectionStrings>

**🔹 Interface Nedir?**

interface, sınıfların **hangi metotları içermek zorunda olduğunu belirten bir şablondur**.  
Ama **içini doldurmaz** — sadece "şunlar olmalı" der.

**🧪 Hayattan Örnek:**

Diyelim ki bir **uzaktan kumanda** yapıyorsun.  
Her kumanda Aç, Kapat, SesAç, SesKıs tuşlarını barındırmalı.  
Ama o tuşların içini TV kumandası başka, klima kumandası başka şekilde doldurur.

İşte o “şunlar olmalı” kısmı = **interface** ✅

**🔧 Basit Kod Örneği:**

**1️⃣ interface tanımlama:**

public interface IDevice

{

void TurnOn();

void TurnOff();

}

**2️⃣ Sınıfın bunu uygulaması (implement etmesi):**

public class Television : IDevice

{

public void TurnOn()

{

Console.WriteLine("TV açıldı");

}

public void TurnOff()

{

Console.WriteLine("TV kapatıldı");

}

}

Sınıf interface'i uygulayınca **tüm metotları yazmak zorundadır.**

**🤔 Neden Kullanırız?**

✅ Kod tekrarını önler  
✅ Ortak kurallar belirler  
✅ Bağımlılığı azaltır → Test, geliştirme kolaylaşır  
✅ SOLID prensiplerinin “Interface Segregation” kısmıdır

**📌 Notlar:**

* interface içinde **değişken tanımlanmaz**, sadece **metot imzası olur**
* Genellikle “I” harfiyle başlar: IRepository, ILogger, IService gibi
* Bir sınıf **birden fazla interface** uygulayabilir

Sonraki işlem abstract klasöründe ırepository adlı bir interface oluşturmak

public interface IRepository<T> //buradaki t paramatresi bizm tablolarımızdır

{

//yapılacak temel crud işlemleri oluşturduk

List<T> List();

int Insert(T p);

int Update(T p);

int Delete(T p);

T GetByID(int id);

////🔍 Neden int var?

//Çünkü genellikle veritabanındaki bir kaydı tanımak için kullanılan birincil anahtar(Primary Key) tipi int olur.

//Yani: int id → Hangi veriyi istiyoruz, onu belirtiriz.

}

Sonra her sınıf entity tablomuz için ınterface oluştur

Örnek

using EntityLayer.Concrete;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DataAccessLayer.Abstract

{

public interface IAboutDal:IRepository<About> //repositoryden miras alıyoruz ekle sil güncelleden hangi sınıfı kullancal ıaboutdalda olduğumuz için about tablosu

{

}

}

**Sonra concrete repository sınıfı oluştur ve alttaki kodları yazalım**

using DataAccessLayer.Abstract;

using System;

using System.Collections.Generic;

using System.Data.Entity;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DataAccessLayer.Concrete

{

public class Repository<T> : IRepository<T> where T : class //dışarıdan bir t nesnesi aldık nereden verileri alacaz ı repositoryden alacağız ı repositoryde de zaten bir t değeri where bu t değerleri nereden gelecek classımızdan entity layer klasslarımızdan

{

Context c = new Context();

DbSet<T> \_object; // dışarıdan alacağımız veriler

public Repository()

{

\_object = c.Set<T>(); // context üzerinden gönderdiğimiz verileri objecte ata

}

public int Delete(T p)

{

\_object.Remove(p); // sil veriyi gönderdiğim p parametresiyle

return c.SaveChanges(); // değişiklikleri kaydet

}

public T GetByID(int id)

{

return \_object.Find(id);

}

public int Insert(T p)

{

\_object.Add(p);

return c.SaveChanges();

}

public List<T> List()

{

return \_object.ToList();

}

public int Update(T p)

{

return c.SaveChanges();

}

}

}

Sonra business Layerda concrete oluşturuyoruz

using DataAccessLayer.Concrete;

using EntityLayer.Concrete;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace BusinessLayer.Concrete

{

public class CategoryManager

{

Repository<Category> repoCategory = new Repository<Category>();

public List<Category> GettAll()

{

return repoCategory.List();

}

}

}

Ama bu kısımlar daha da farklı oluyor başka projede kullanılabilir

Fluent Validation Kurulumu

Business layer katmanında abstract Klasörü oluştur

Sonra hepsine interface oluştur mesela IaboutService gibi

using EntityLayer.Concrete;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace BusinessLayer.Abstract

{

public interface IAboutService

{

List<About> GetList();

void AboutAdd(About about);

About GetById(int id);

void AboutDelete(About about);

void AboutUpdate(About about);

}

}

İçleri böyle olucak

Sonra BusinessLayer Katmanında klasör oluştur ve sınıf aç

using EntityLayer.Concrete;

using FluentValidation;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace BusinessLayer.ValidationRules

{

public class CategoryValidator : AbstractValidator<Category>

{

public CategoryValidator()

{

RuleFor(x => x.CategoryName).NotEmpty().WithMessage("Kategori Adını Boş geçemezsiniz");

RuleFor(x => x.CategoryDescription).NotEmpty().WithMessage("Kategori Açıklamaşını Boş geçemezsiniz");

}

}

}

View kısmı

@Html.ValidationMessageFor(x => x.CategoryDescription)

Controller kısmı

CategoryValidator categoryValidator = new CategoryValidator();

ValidationResult results = categoryValidator.Validate(category);

if (results.IsValid)

{

categoryManager.CategoryUpdateBL(category);

return RedirectToAction("AdminCategoryList");

}

else

{

foreach (var item in results.Errors)

{

ModelState.AddModelError(item.PropertyName, item.ErrorMessage);

}

}

return View();

Ek bilgi Serisi

<p>@item.BlogContent.Substring(0,item.BlogContent.Substring(0,200).LastIndexOf(" "))...</p>

@\*buranın amacı 0 ile 200. karakterde boşluğa kadar al ve sonuna ... ekle ki garip şeyler olmasın elma yazacakken el yazmasın\*@

<span>&nbsp;-&nbsp; @item.BlogDate.ToString("dd-MMM-yyyy")</span>

@\*BÖYLE YAPARAKDA AY DEĞERİNİ DÜZGÜNCE ALIYORUZ\*@