SOLID Programlama Prensipleri / SOLID Principles

#solidprensipleri

Prensipler bakımı ve anlaşılması kolay kod yazmak için vardır.

Bu prensiplerin nerede ve nasıl kullanılması bilinmesi gerekir.
Ama bu her zaman ve heryerde doğru olan bir önerme değildir.
(Örneğin prototip aşamasındaki bir proje için.)

S- Single Responsibility Principle

Bir Componenet sadece bir amaç için değiştirilmeli.
Component = Sınıf, library, fonksiyon vb..

Kodun küçük ve bir amaca hizmet eden compenentler şeklinde tasarlanması yeniden kullanilabilirliği arttırır.

Componentlerin küçük olması daha rahat anlaşılmalarını sağlar.

Eğer birden fazla amaca hizmet ediyorsa o zaman bu prensibi ihlal ediyor demektir.

Örneğin bir Account Sınıfı

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

namespace SOLID
{
   public class Account
   {

      public string FirstName { get; set; }
      public string LastName { get; set; }
      public string Street { get; set; }
      public int ZipCode { get; set; }
      public string City { get; set; }
      public Guid AccountId { get; set; }
      public string Password { get; set; }
      public string UserName { get; set; }

      public void ChangePasswırd(string newPassword) => Console.WriteLine($"Changed
password to {newPassword}");
      public void ChangeAddress(string street, int zipCode, string city) =>
Console.WriteLine("Changed adress");
```

```
        public string GetAccountHolderFirstName(Guid accountNumber) => this.FirstName;
        public string GetAccountHolderLastName(Guid accountNumber) => this.LastName;

        public void DepositMoney(decimal amount) => Console.WriteLine($"{amount} has
been deposited to {this.AccountId}");
        public void WithdrawMoney(decimal amount) => Console.WriteLine($"{amount}
has been withdrawn from {this.AccountId}");
        public void Login(string username, string password) =>
Console.WriteLine($"{username} has tried to log in.");
        public void Logout(string username) => Console.WriteLine($"{username} has tried
to log out.");

    }
}
```

Görüldüğü gibi içerisinde birçok metod özellik vb şeyler var.

```
class AddressVerifier
{
    public bool IsAddressValid(Account account)
    {
        //burada account ile tonlarca gereksiz ozellik geliyor
        return true;
    }
}
```

Burada Account sınıfı ile adresin doğruluğu kontrol edilmek isteniyor ancak adresten başka tonlarca özellik geliyor. Bu tarz sıkıntıları engellemek için ;

```
public class Account
{
    public Guid AccountId { get; set; }
    public Profile Profile { get; set; }
    public Address Address { get; set; }
}

public class Address
{
    public string  Street { get; set; }
    public int ZipCode { get; set; }
    public string City { get; set; }
    public string Country { get; set; }
}
public class AddressVerifier
{
   public bool IsAdrressValid(Address address)
   {
        return true;
```

```
  }
}
```

Şeklinde yapmak en doğrusu olacaktır.

O - Open/Closed Principle

Bir Componenet değişime kapalı , gelişime açık olmalıdır. Var olan kodları değiştirerek değil genişleterek kullanmak.

Genelde (backward compatibility) sorunlarını çözmek için yararlı olur.

Örneğin zipCode yani Türkiye'de posta kodu olarak geçen bu kod ile o eyaletin veya şehrin var olup olmadığına bakacağız.

Burada basitçe gelen koda göre kodun geçerli olup olmadığı kontrol ediliyor. Bölge icabı zip Kod 10 un altında olanlar geçersiz koddur.

```
public class AddressVerifier
{
  public static bool IsValidZipCode(int zipCode)
  {
     if (zipCode < 10)
     {
        return false;
     }
     return true;
  }
}
```

Doğruluk kontrol edildikten sonra başka bir sınıfta kullanılıyor ve doğru bir kod ise karşılaştırmaya giriyor

```
public class StateFinder
{
   public String GetStateNameForZipCode(int zipCode)
   {
      if (!AddressVerifier.IsValidZipCode(zipCode))
      {
        throw new InvalidOperationException($"Invalid ZipCode : {zipCode}");
      }

      switch (zipCode)
      {
        case 11:
          return "California";
        case 12:
          return "Utah";
        default:
          throw new InvalidOperationException($"No state is found with {zipCode}");
```

```
    }
  }
}

namespace SOLID_S
{
  class Program
  {
    static void Main(string[] args)
    {
      var stateFinder = new StateFinder();
      try
      {
        Console.WriteLine(stateFinder.GetStateNameForZipCode(11));
        Console.WriteLine(stateFinder.GetStateNameForZipCode(9));
      }
      catch (InvalidOperationException exception)
      {
        Console.WriteLine(exception.Message);
      }
    }
  }
}
```

Böyle bir çalışmada ekrana çıkacak olan yazı;

Baştada söylendiği gibi değişime kapalı gelişime açık olmalı. Yani örneğin bir tane daha ülkenin desteklenmesi istendi
Ve bu ülkenin geçerli olan zip kodları bizimkini kapsıyor 10 dan küçük değilde 5 ten küçük olmayanlar geçersiz sayılıyor.
Bu durumda biz sınıfımızı değiştirmiş oluruz böylece kodda hatalar da yaratmış olabiliriz.

```
public class AddressVerifier
{
  public static bool IsValidZipCode(int zipCode)
  {
    if (zipCode < 5)
    {
      return false;
    }
    return true;
  }
}
```

Var olan bir sınıftaki metodu değiştirmiş olduk ve ihlal gerçekleştirmiş olduk. Bu bize programın işleyişinde hatalar getirecektir.
Yeni Ekran çıktısı;

Şimdi yapacağımız şey ise iki ülkenin farklı yerlerde değerlendirilmesini sağlamak. Bunun için static metodumuz olan IsValidZipCode 'u virtual olarak tanımlıyoruz. Yeni bir sınıf açıyoruz ve implemente ediyoruz.

```
public class AddressVerifier
{
  public virtual bool IsValidZipCode(int zipCode)
  {
     if (zipCode < 10)
     {
        return false;
     }
     return true;
  }
}

public class GermanAddressVerifier : AddressVerifier
{
  public override bool IsValidZipCode(int zipCode)
  {
     if (zipCode < 5)
     {
        return false;
     }
     return true;
  }
}
```

Address verifier sınıflarımız bu şekilde olmalı.

```
namespace SOLID_S
{
   public class StateFinder
   {

   //Artık burada kullanılmak istenen ülkenin verifier'i gönderilecek.
      public string GetStateNameForZipCode(int zipCode,AddressVerifier verifier)
      {
         if (!verifier.IsValidZipCode(zipCode))
         {
            throw new InvalidOperationException($"Invalid ZipCode : {zipCode}");
         }

         switch (zipCode)
         {
            case 9:
               return "Munih";
            case 8:
               return "Berlin";
```

```
            case 11:
               return "California";
            case 12:
               return "Utah";
            default:
               throw new InvalidOperationException($"No state is found with {zipCode}");
         }
      }

   }
}
```

SON OLARAK PROGRAM SINIFINA KULLANILISI GORMEK ICIN BAKARSAK
//Artık burada kullanılmak istenen ülkenin verifier'i parametre olarak gönderilecek.

```
namespace SOLID_S
{
   class Program
   {
      static void Main(string[] args)
      {
         var stateFinder = new StateFinder();
         try
         {
            var addressVerifier = new AddressVerifier();

            Console.WriteLine(stateFinder.GetStateNameForZipCode(11,addressVerifier));
            Console.WriteLine(stateFinder.GetStateNameForZipCode(9,addressVerifier));
         }
         catch (InvalidOperationException exception)
         {
            Console.WriteLine(exception.Message);
         }

         try
         {
            var addressVerifier = new GermanyAddressVerifier();
            Console.WriteLine(stateFinder.GetStateNameForZipCode(9, addressVerifier));
         }
         catch (InvalidOperationException exception)
         {
            Console.WriteLine(exception.Message);
         }
      }
   }
}
```

BU ŞEKİLDE ARTIK ADDRESS VERİFİER SINIFIMIZI DEĞİŞTİRMEMİŞ GELİŞTİRMİŞ OLDUK

L - Liskov's Substitution Principle

Birbirinden kalıtım alan sınıfların sistemi bozmadan değiştirilebilimesini esas alır. Sistemin kararlılığının bozulmaması için değişim zamanında değişimin sorunsuz gerçekleşmesini istiyor.

**A - Invariants (Değişmeyen şeyler) **: Sınıfta asla değişmeyen özellikler : örneğin bir matematik sınıfı var ve içinde pi sayısını tutan değişken var bunun değişmemesi gerekir.

**B - Pre-Conditions (İşlem yapılmadan önce yapılan kontroller) **: Bir metod yazdınız örneğin para yatırma metodu miktar isimli bir parametre alıyor ve metodun doğru çalışması için belli değerler arasında değerler alması bekleniyor. Miktar eksi olması beklenmez miktarı kontrol etme işlemi pre-conditions'lardandır.

**C - Post-conditions (İşlem sonrası yapılan kontroller) **: Örneğin para yatırma işlemi tamamlandı daha sonra paranın 0 ' dan büyük olmasının kontrolünü sağlamak post-conditions'lardır.

Bu kontroller sayesinde yapılan hatalar hemen farkedilir ve sistemin yanlış verilerle çalışmaması sağlanır.

```
namespace SOLID_S
{
   //KONTROLCU BIR SINIF YAPIYORUZ
   public class Require
   {
      public static void That(Func<bool> check)
      {
         if (!check())
         {
            throw new ArgumentException("Invalid argument has been passed");
         }
      }
   }
}
```

REQUIRE SINIFININ POST CONDITION'I ADDRESS SERVICE SINIFININ PRE-CONDITION'IDIR

YANI BIR KODUN POST CONDITION'I BASKA BIR KOD PARCASININ PRE-CONDITION'I OLABILIR

```
namespace SOLID_S
{
   public class AddressService
   {
```

```
    public void ChangeAddress(Guid accountId, Address newAddress) =>
Console.WriteLine("Changed address");

    public virtual Address GetAddressForUser(int userId)
    {
      Require.That(() => userId > 10);

      var address = new Address()
      {
        Street = "Testreet",
        ZipCode = 12312,
        City = "Tokat",
        Country = "USA",
        State = "New England"

      };

      Require.That(() => address.ZipCode > 1000);
      return address;
    }
  }
}

public class AccountService
{
  public void ChangeAddress(Address address)
  {
    Require.That(() => address.ZipCode > 1000);
    Console.WriteLine("Account address has been changed succesfully");
  }
}
```

BU SEKILDE YAZILIMA KONTROLLER KOYARAK YAPILAN HATALARI CABUK ALGILAMA VE OLASI DURUMLARI ONLEME ICIN TEDBIRLER OLUSTURULMUSTUR

I - Interface Segregation Principle

Başka sınıfları kullanan kodların ihtiyaçlarından fazlasını kullanmasını engellemek ve bu sayede
Bakımını ve okunabilirliğini arttırmak amacıyla sunulmuş bir prensiptir.

Bir sınıfın işlevleri tutarlı ve ufak arayüzler ile bölünür.

Örneğin bir aşevi classimiz olsun ve bu aşevi class'ı insanlara yemek verme metodu olsun.

```
public interface IHuman
{
  void Eat();
```

```
    void Talk();
    void Walk();
    void Sleep();
}

public class Asevi
{
    public void AddToList(IHuman human)
    {
        throw new NotImplementedException();
    }
}
```

Eğer buraya bir hayvan ekleyecek olursak onun için de ayrı bir metod yazmamız gerekir şöyle ki;

```
public interface IAnimal
{
    void Eat();
    void Talk();
    void Walk();
    void Sleep();
    void Bite();
}

public class Asevi
{
    public void AddToList(IHuman human)
    {
        throw new NotImplementedException();
    }

    public void AddToList(IAnimal animal)
    {
        throw new NotImplementedException();
    }
}
```

Görüldüğü gibi bir metod daha eklemem gerekiyor ve kodlar fazlaca büyüyor. Asevinin aradığı ortak özelliği
Yani yemek yeme özelliğini kullanarak tek metodda bu işi çözebiliriz ve aralarında ilişki oluşturmuş oluruz.

```
namespace SOLID_I
{
    public class Asevi
    {
        public void AddToList(IOmnivore human)
        {
```

```
            throw new NotImplementedException();
        }
    }

    public interface IHuman :IOmnivore
    {
        void Talk();
        void Walk();
        void Sleep();
    }

    public interface IAnimal :IOmnivore
    {
        void Walk();
        void Sleep();
        void Bite();
    }

    public interface IOmnivore
    {
        void Eat();
    }
}
```

Her ikisinin icerisindeki yemek yeme fonksiyonunu interface olarak SEGREGATE (ayırma) ettik ve onu human ile animal interface'lerine miras olarak aldık böylece ikiside eat fonksiyonunu bir interface 'dan almış oldu ve Asevi sınıfına tek bir metod eklemek yeterli gelmiş oldu. Bundan sonraki eklenecek farklı canlı türlerinde yemek yemesi ana özellik olarak alınacak ve hala tek bir metod üzerinden sürdürülebilir olacak.

D - Dependency Inversion Principle

Yüksek seviyeli sınıfların düşük seviyeli sınıfları kendi halleriyle değilde soyut halleri ile kullanmasını konu alır. Buda bağımlılığı düşürür.

Genel anlamda anlattığı şey bir sınıf eğer başka bir sınıfı kullanacaksa onu newlemeden yani soyut bir başka sınıf ile kullanmalıdır.

Dependenc Inversion Prensibinde temel olarak bir tasarım dili kullanılır bu Dependency Injection'dır

Ctor'dan bağımlılık sınıfının implementasyonu olan interfacei geçmektir.

```
class CustomerManager
{
    private IMevzuat _mevzuat;
    public CustomerManager(IMevzuat mevzuat)
    {
        _mevzuat = mevzuat
```

```
    }
    public void Add()
    {
        _mevzuat.IslemYap();
    }
}



class CustomerManager
{
    private IMevzuat _mevzuat;
    public CustomerManager(IMevzuat mevzuat)
    {
        _mevzuat = mevzuat
    }
    public void Add()
    {
        _mevzuat.IslemYap();
    }
}
interface IMevzuat
{
    void IslemYap();
}
class BırıncıMevzuat: IMevzuat
{
    public void IslemYap()
    {
        Console.Write("Birinci Mevzuat işlem yaptı");
    }
}
class IkinciMevzuat: IMevzuat
{
    public void IslemYap()
    {
        Console.Write("İkinci Mevzuat işlem yaptı");
    }
}
```

Bu yapı sayesnde yeni mevzuat eklense dahi customer manager sınıfında herhangi bir değişiklik yapmaya gerek kalmaz interface onu otomatık getirir.

Katmanlı mimarilerdeki katmanların geçişini sağlamak için dependency injection tasarım deseni yaygın olarak kullanılır. (Bkz: IoC Container Link)

Business katmanı veri erişim katmanı ile iletişim kurarken veri erişim katmanındaki somutla değil soyutla iletişim kurar.

Eğer veri erişim katmanında entity framework ile yapılmışsa ve ileride bunu değiştirereceği zaman business katmanında değişmemesi için.