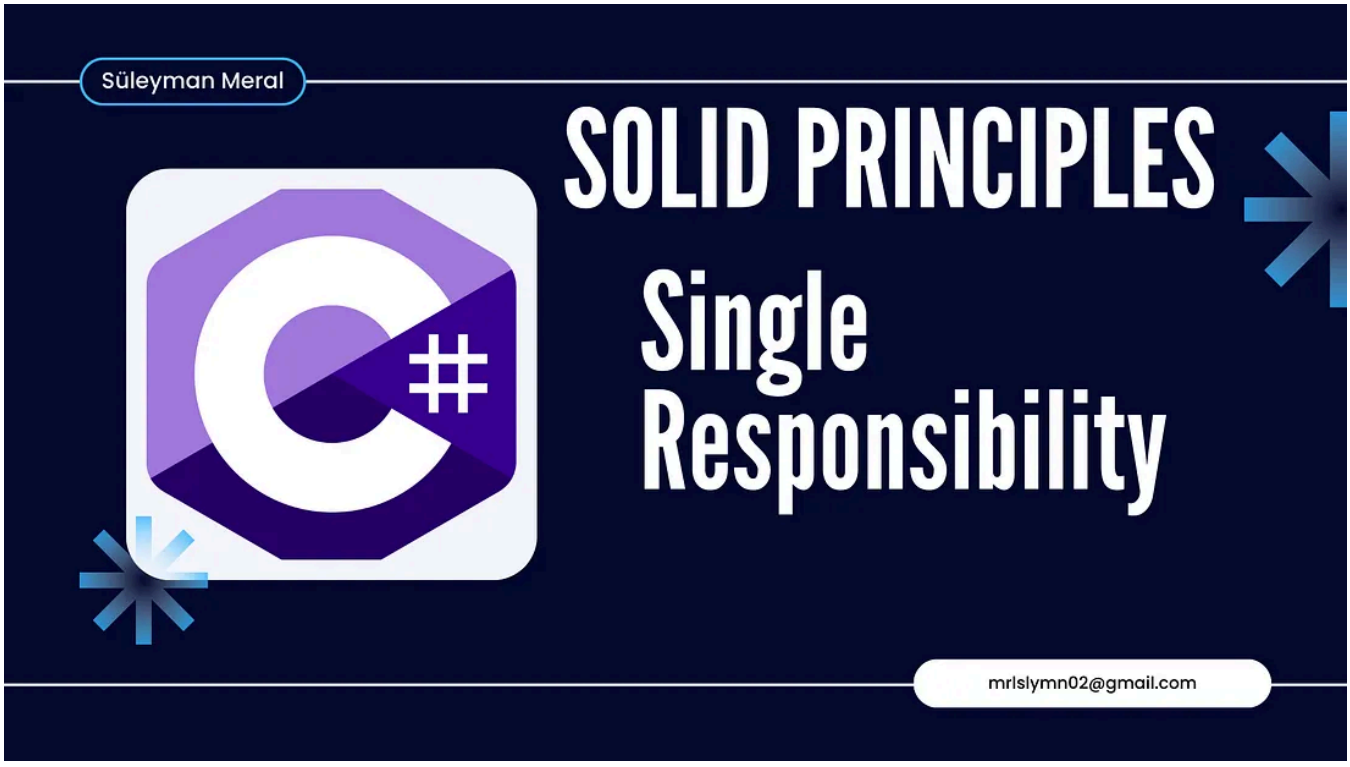
9 min read · May 5, 2025

Süleyman Meral



Herkese Merhaba! Bu yazımda sizlere yazılım gelişirmenin önemli konularından olan SOLID Prensiplerinin ilk prensibi olan Single Responsibility ilkesinden bahsedeceğim. Öncelikle teorik olarak Single Responsibility ilkesi nedir? Ne işe yarar? gibi sorulara yanıt verelim. Daha sonra bir kod örneği ile konuyu pekiştirelim.

Single Responsibility Nedir?

Single Responsibility kavramı, adından da anlaşılacağı üzere bir sınıfın ya da metodun yalnızca tek bir sorumluluğa sahip olması gerektiği prensibine dayanır. Bu

prensip, iki temel ilkeye dikkat çeker:

1. Bir sınıfın veya metodun değişmesi için yalnızca tek bir neden olmalıdır.
2. Bir sınıfın veya metodun yalnızca tek bir sorumlulukla ilgilenmesi gerekir.

Biraz daha açıklamak gerekirse, uygulamamızdaki mantıksal işlemleri sınıf veya metod olarak tasarlarken, farklı türdeki sorumlulukları tek bir yapıya yüklemememiz gerekir. Özetle yalnız bir işi yerine getiren bileşenler tasarlamanın önemini vurgular.

Her yazılım birimi(Sınıflar,fonksiyonlar hatta kütüphaneler) daha iyi bir tasarımın ortaya çıkması için sadece bir sorumluluğa sahip olmaya özen göstermelidir. Bu sayede birimlerin değişmesi için tek bir sebebi olacağından gerekli değişikliği uygulamak daha ucuz olacaktır.

> *(Temiz Kod — M.Furkan Ardoğan)*

Single Responsibility, bir sınıfın veya metodun yalnızca tek bir sorumluluğu olması gerektiğini savunan bir yazılım tasarım ilkesidir. Her bileşenin sadece kendi görevinden sorumlu olması, kodun daha anlaşılır ve bakımı kolay olmasını sağlar.

Son olarak daha iyi kavramak adına yine Sayın M.Furkan Ardoğan hocamın "Temiz Kod" adlı kitabından bir örnek verelim.

İsviçre çakısı,üzerinde makas,bıçak,testere gibi birçok edevatın birlikte bulunduğu bir malzemedir. Bunca farklı edevatın tek bir yerde toplanması ilk bakışta iyi gibi gözüküyor olsa da bu aletin aslında kullanışsız olduğu zamanla kabul edilecek bir gerçektir. Özellikle yazılım bileşenlerinin tasarımı yapılırken asla ama asla buna benzer tutumlar takınılmamalıdır. Hatta bu durum,yazılım geliştiriciliğinde o denli sorundur ki *Swish Army Knife* adlı bir karşıt kalıp literatüre girmiştir.

> *(Temiz Kod — M.Furkan Ardoğan)*

Single Responsibility İlkesi Ne İşe Yarar?

Single Responsibility prensibi, yazılım bileşenlerinin daha okunabilir, test edilebilir ve sürdürülebilir olmasını sağlar. Her sınıfın veya metodun tek bir amaca odaklanması, kodun daha kolay yönetilmesine ve değişikliklerin diğer alanları etkilemeden yapılabilmesine olanak tanır.

- Kod karmaşıklığını azaltır.
- Kodun okunabilirliğini arttırır.
- Test etmeyi kolaylaştırır.

Tek sorumluluk prensibi,sistemdeki her bir sınıfın ve hatta sınıflar altında *fonksiyonların ne yaptıklarıyla* ilgilidir. Her sınıf hatta her fonksiyon yazılımın kötü tasarlanmış bölümlerini yok etmek için yalnızca tek bir sorumluluğa sahip olduğundan emin olmalıdır. Böylecek sorumlulukların değişmesi durumunda değişikliği nerede yapacağınızı bilirsiniz.

> *(Temiz Kod — M.Furkan Ardoğan)*

Teorik olarak Single Responsibility ilkesini bu şekilde kısaca anlatabiliriz. Şimdi konuyu daha iyi kavramak amacı ile ufak bir proje örneği yapalım. Öncelikle karmaşık bir kod oluşturalım daha sonra Single Responsibility ilkesine göre kodumuzu değiştirelim. Sonuçları gözlemleyelim.

Farklı ödeme yöntemleri ile satın alma işlemi yaptığımız basit bir senaryomuz olsun. Başlangıçta Payment adında bir class oluşturalım.

```
public class Payment
{
    public string CardNumber { get; set; }
    public string CardHolderName { get; set; }
    public string Cvc { get; set; }
    public string IBANNumber { get; set; }
    public string ReceiverName { get; set; }
    public string SenderName { get; set; }
    public string Mail { get; set; }

    public void PayByCreditCard(string CardNum,string CardHolder,string Cvc,str
    {
        if(CardNum.Length==16 && !string.IsNullOrEmpty(CardHolder) && Cvc.Lengt
        {
            // Process
            Console.WriteLine($"Payment process has been completed successfully
            SendMail(email);
        }
        else
        {
            Console.WriteLine($"Invalid Payment Data");
        }
```

```
    }
    public void PayByTransfer(string IbanNumber,string ReceiverName,string Senc
    {
        if (IbanNumber.Length == 34 && !string.IsNullOrEmpty(ReceiverName) && !
        {
            // Process
            Console.WriteLine($"Payment process has been completed successfully
            SendMail(email);
        }
        else
        {
            Console.WriteLine($"Invalid Payment Data");
        }

    }
    public void SendMail(string mail)
    {
        Console.WriteLine($"Mail has been sent successfully. Mail Adress:"+mail
    }
}
```

Yukarıdaki kodu hep birlikte inceleyelim. Payment classı içerisinde ödeme bilgilerine ait propertyler barındırıyor. Aynı zamanda ödeme yöntemleri için gerekli metodlar da yine Payment classında tanımlanmış. Metodların içeriğine bakacak olursak hem belli validation işlemleri gerçekleşiyor hem de ödeme işlemleri gerçekleşiyor. Classımız single property ilkesini tamamen ezmiş durumda. Payment classına birden fazla sorumluluk yüklenmiş. Metodlarda veya propertylerde bir değişikliğe gidecek olursak bütün yapı bundan etkilenecek. Bizim için istenmeyen bir durum olacak. Aynı zamanda test işlemimiz çok zorlaşacak.

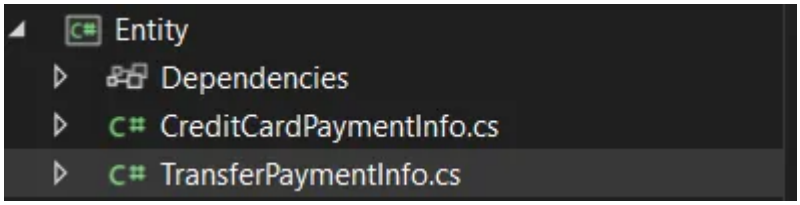
```
using singleresponsibilityMedium;

Payment pay = new Payment();

pay.CardNumber = "1111111111111111";
pay.CardHolderName = "Süleyman Meral";
pay.Cvc = "123";
pay.Mail = "mrlslymn02@gmail.com";

pay.PayByCreditCard(pay.CardNumber, pay.CardHolderName, pay.Cvc, pay.Mail);

pay.IBANNumber = "AA123456789123456789123456789123456789";
pay.SenderName = "Süleyman Meral";
```

```
pay.ReceiverName = "Emine Meral";

pay.PayByTransfer(pay.IBANNumber, pay.ReceiverName, pay.SenderName, pay.Mail);
```

Kodumuzu bu şekilde kullanmak dışarıdan bir probleme sebep olmaz. Ama kodumuzu değiştirmemiz gerektiği zaman bir sürü sorunla karşılaşacağız. Solid bize bunun önüne geçmemiz için yardım edecek. Unutmayalım ki, SOLID'e uygun kod yazmak aynı zamanda geleceğe yönelik kod yazmaktır.

Şimdi çeşitli işlemlerle kodumuzu SOLID'in Single Responsibility ilkesine uygun hale getirelim. Katmanlı mimariye uygun bir düzenleme yapacağız. Bu da SOLID'i destekler nitelikte olacak.

İlk olarak bir Entity katmanı oluşturalım. Ve bu katman içerisine Kredi kartı ve Havale işlemleri için gerekli varlıkları ekleyelim.




```
public class CreditCardPaymentInfo
{
    public string CardNumber { get; set; }
    public string CardHolderName { get; set; }
    public string Cvc { get; set; }
    public string Mail { get; set; }
}
```

```
public class TransferPaymentInfo
{
    public string IBANNumber { get; set; }
    public string ReceiverName { get; set; }
    public string SenderName { get; set; }
    public string Mail { get; set; }
}
```

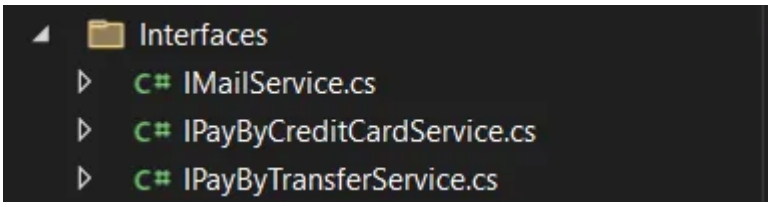
Kodumuzdaki nesne taşıma işlemini ayrı classlara taşıdık. Bu sayede Payment classı bir sorumluluktan kurtulmuş oldu.

```
public class Payment
{
    public void PayByCreditCard(string CardNum,string CardHolder,string Cvc,str
    {
        if(CardNum.Length==16 && !string.IsNullOrEmpty(CardHolder) && Cvc.Lengt
        {
            // Process
            Console.WriteLine($"Payment process has been completed successfully
            SendMail(email);
        }
        else
        {
            Console.WriteLine($"Invalid Payment Data");
        }

    }
    public void PayByTransfer(string IbanNumber,string ReceiverName,string Senc
    {
        if (IbanNumber.Length == 34 && !string.IsNullOrEmpty(ReceiverName) && !
        {
            // Process
            Console.WriteLine($"Payment process has been completed successfully
            SendMail(email);
        }
        else
        {
            Console.WriteLine($"Invalid Payment Data");
        }

    }
    public void SendMail(string mail)
    {
        Console.WriteLine($"Mail has been sent successfully. Mail Adress:"+mail
    }
}
```

Kodumuz bu işlemden sonra yukarıdaki halini almış oldu. Şimdi ise metodları çeşitli classlara taşıyalım ve birden fazla metodu içerme zorunluluğundan kurtulmuş olalım.

İşlemler için bir Business katmanı oluşturalım. Ödeme yöntemleri için ayrı interfaceler oluşturalım. Ve bu interfacelerden türeyen classlar yazalım. Bu sayede

ödeme yöntemleri de kendi içerisinde ayrılmış olsun. Tek bir interface'den türetmememizin sebebi SOLID'in Interface Segregation ilkesini ezmemek bunu daha sonra inceleyeceğiz.



Interfaces klasörü oluşturup içerisine yukarıda görülen interfaceleri ekledik.

```
public interface IMailService
{
    void SendMail(string mail);
}
```

```
 public interface IMailService
 {
     void SendMail(string mail);
 }
```

```
 public interface IPayByCreditCardService
 {
      void PayByCreditCard(CreditCardPaymentInfo creditCardPaymentInfo);

 }
```

```
public interface IPayByTransferService
{
    void PayByTransfer(TransferPaymentInfo transferPaymentInfo);

}
```

Şimdi bu interfacelerden türeyen classları yazalım ve metodların içeriğini oluşturalım.

```
public class PayByCreditCardService : IPayByCreditCardService
{
    public void PayByCreditCard(CreditCardPaymentInfo creditCardPaymentInfo)
    {
        if (creditCardPaymentInfo.CardNumber.Length == 16 && !string.IsNullOrE
        {
            // Process
            Console.WriteLine($"Payment process has been completed successfull

        }
        else
        {
            Console.WriteLine($"Invalid Payment Data");
        }
    }
}
```

```
public class PayByTransferService : IPayByTransferService
{
    public void PayByTransfer(TransferPaymentInfo transferPaymentInfo)
    {
        if (transferPaymentInfo.IBANNumber.Length == 34 && !string.IsNullOrEm
        {
            // Process
            Console.WriteLine($"Payment process has been completed successful

        }
        else
        {
            Console.WriteLine($"Invalid Payment Data");
        }
    }
}
```

Metodlarımızıda ayrı classlara taşımış olduk. Fakat metodların içine odaklandığımızda metodların hem doğrulama hem de ödeme işlemlerini gerçekleştirdiğini görüyoruz. Bu yapı da Single Responsibility ilkesine ters düşen bir yapıdır. Peki bunun için ne yapabiliriz?

Görüldüğü gibi ilgili propertyler kontrol ediliyor. Property'lerimize bir attribute tanımı yaparsak metodun içine eklemeden bir kontrol yapabiliriz. Konunun sapmaması açısından .NET Framework üzerinde hazır bulunan attribute yapılarını

kullanacağız. Ama dilersek kendi attribute yapılarımızı da yazmamız mümkün. Aşağıdaki yazımda bunu nasıl yapabileceğimi anlatmıştım. Dilerseniz inceleyebilirsiniz.


```
public class CreditCardPaymentInfo
{
    [Required]
    [StringLength(11, MinimumLength = 11, ErrorMessage = "Card number must b
    public string CardNumber { get; set; }

    [Required]
    public string CardHolderName { get; set; }

    [Required]
    [StringLength(3, MinimumLength = 3, ErrorMessage = "Cvc number must be e

    public string Cvc { get; set; }

    [Required]
    public string Mail { get; set; }
}
```

CreditCardPaymentInfo classımızı bu şekilde güncelliyoruz. Çalışma zamanında ilgili proplar kontrol edilip hata varsa bize söylenecek.

```
public class TransferPaymentInfo
 {

    [Required]
    [StringLength(34, MinimumLength = 34, ErrorMessage = "IBAN Number must be
    public string IBANNumber { get; set; }

    [Required]
```

```
        public string ReceiverName { get; set; }

        [Required]
        public string SenderName { get; set; }

        [Required]
        public string Mail { get; set; }
    }
```

TransferPaymentInfo classımızı da bu şekilde düzenliyoruz.

Metodlar ve propertylerimizi ayırdık başlangıçta oluşturduğumuz Payment classı şuan bir işlem yapmıyor. Bu classımızı yönlendirici olarak düzenleyeceğiz. Ödeme işlemleri buradan yönetilecek.

```
public class PaymentProcessor
{

    private readonly IMailService _mailService;
    private readonly IPayByCreditCardService _creditCardService;
    private readonly IPayByTransferService _payByTransferService;

    public PaymentProcessor( IMailService mailService, IPayByCreditCardService
    {

        _mailService = mailService;
        _creditCardService = creditCardService;
        _payByTransferService = payByTransferService;
    }

    public void ProcessCreditCardPayment(CreditCardPaymentInfo info)
    {
    _creditCardService.PayByCreditCard(info);
    _mailService.SendMail(info.Mail);

    }

    public void ProcessTransferPayment(TransferPaymentInfo info)
    {

    _payByTransferService.PayByTransfer(info);
    _mailService.SendMail(info.Mail);

    }
```

```
}
```

PaymentProcessor classı sadece koordinasyon yapıyor. Bu da prensibimizi destekler nitelikte. Burada solidin Dependency Inversion Principle (DIP) prensibini de kullanmış olduk bu ilkeyi başka bir yazıda inceleyeceğiz. Aslında bir ilkeyi takip ettiğimiz de diğer ilkeler de bizi destketliyor. Bu sayede ortaya mükemmel bir kod yapısı çıkıyor.

Şimdi validasyon işlemlerimizin kontrolü için de ayrı bir katman oluşturalım.

“ValidationService” adında yeni bir katman oluşturalım. Bu sayede katmanlar da sadece tek bir sorumluluğa hizmet edecek.

Katmanımıza bir interface ve ServiceManager classı ekleyelim.

```
public interface IValidationService
{
    bool Validate<T>(T model);
}
```

```
 public class ValidationServiceManager
 {
     public bool Validate<T>(T model)
     {
         var context = new ValidationContext(model, serviceProvider: null, ite
         var results = new List<ValidationResult>();
         bool isValid = Validator.TryValidateObject(model, context, results, t

         if (!isValid)
         {
             foreach (var error in results)
             {
                 Console.WriteLine("Validation Error: " + error.ErrorMessage);
             }
         }

         return isValid;
```

```
        }
    }
```

ValidationService classımız içerisinde Validate isimli bir metod tanımlıyoruz. Bu metod generic bir metod olma özelliğine sahiptir. Generic yapılar hakkında bilgi sahibi olmak istiyorsanız aşağıdaki bağlantıdan bu konu üzerinde yazmış olduğum yazıyı da inceleyebilirsiniz.

C# Generic Yapılar Örnekli Anlatım

Herkese merhaba! Bu yazımızda C# üzerinde kullanılan generic yapılardan bahsedeceğiz. Bu yapılara gerçek bir proje...

medium.com

Validasyon işlemimizi de ayırmış olduk. Şimdi bu işlemide PaymentProcessor classımıza entegre edelim .

```
using Business.Interfaces;
using Entity;
using ValidationService;

public class PaymentProcessor
{
    private readonly IMailService _mailService;
    private readonly IPayByCreditCardService _creditCardService;
    private readonly IPayByTransferService _payByTransferService;
    private readonly IValidationService _validationService;

    public PaymentProcessor(
        IMailService mailService,
        IPayByCreditCardService creditCardService,
        IPayByTransferService payByTransferService,
        IValidationService validationService)
    {
        _mailService = mailService;
        _creditCardService = creditCardService;
        _payByTransferService = payByTransferService;
        _validationService = validationService;
    }

    public void ProcessCreditCardPayment(CreditCardPaymentInfo info)
    {
        if (_validationService.Validate(info))
        {
```

```
            _creditCardService.PayByCreditCard(info);
            _mailService.SendMail(info.Mail);
        }
        else
        {
            Console.WriteLine("Kredi kartı bilgileri geçersiz.");
        }
    }

    public void ProcessTransferPayment(TransferPaymentInfo info)
    {
        if (_validationService.Validate(info))
        {
            _payByTransferService.PayByTransfer(info);
            _mailService.SendMail(info.Mail);
        }
        else
        {
            Console.WriteLine("Havale bilgileri geçersiz.");
        }
    }
}
```

Bütün işlemlerimiz solide uygun bir şekilde PaymentProcessor classında toplanmış durumda. Tek yapmamız gereken program.cs de bunu kullanmak. Console uygulaması oluşturduğumuz için bu şekilde kullanacağız. ASP.NET Core projelerinde daha farklı bir yapı oluşmaktadır. Dilerseniz o konu üzerinde de bir araştırma yapabilirsiniz.

```
using Business.Interfaces;
using Business.Services;
using Entity;
using ValidationService;


IMailService mailService = new MailService();
IPayByCreditCardService creditCardService = new PayByCreditCardService();
IPayByTransferService transferService = new PayByTransferService();
IValidationService validationService = new ValidationServiceManager();


PaymentProcessor processor = new PaymentProcessor(mailService, creditCardServic

var creditCardInfo = new CreditCardPaymentInfo
{
    CardNumber = "1234567812345678",
    CardHolderName = "Süleyman Meral",
```

```
    Cvc = "123",
    Mail = "test@example.com"
};

var transferInfo = new TransferPaymentInfo
{
    IBANNumber = "TR0012345678901234567890123499999",
    ReceiverName = "Ali Veli",
    SenderName = "Veli Ali",
    Mail = "transfer@example.com"
};

processor.ProcessCreditCardPayment(creditCardInfo);

processor.ProcessTransferPayment(transferInfo);
```

```
Validation Error: Card number must be exactly 11 characters.
Validation Error: Card number must be exactly 3 characters.
Kredi karti bilgileri geçersiz.
Payment process has been completed successfully.
Mail has been sent successfully. Mail Adress:transfer@example.com
```

Hatalarımız attribute yapılarında belirlediğimiz gibi bizlere sunuluyor.

```
public class Payment
{
    public string CardNumber { get; set; }
    public string CardHolderName { get; set; }
    public string Cvc { get; set; }
    public string IBANNumber { get; set; }
    public string ReceiverName { get; set; }
    public string SenderName { get; set; }
    public string Mail { get; set; }

    public void PayByCreditCard(string CardNum,string CardHolder,string Cvc,str
    {
        if(CardNum.Length==16 && !string.IsNullOrEmpty(CardHolder) && Cvc.Lengt
        {
            // Process
            Console.WriteLine($"Payment process has been completed successfully
            SendMail(email);
        }
        else
        {
            Console.WriteLine($"Invalid Payment Data");
        }
```

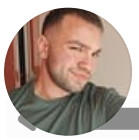
```
        }
        public void PayByTransfer(string IbanNumber,string ReceiverName,string Senc
        {
            if (IbanNumber.Length == 34 && !string.IsNullOrEmpty(ReceiverName) && !
            {
                // Process
                Console.WriteLine($"Payment process has been completed successfully
                SendMail(email);
            }
            else
            {
                Console.WriteLine($"Invalid Payment Data");
            }

        }
        public void SendMail(string mail)
        {
            Console.WriteLine($"Mail has been sent successfully. Mail Adress:"+mail
        }
    }
```

Başlangıçtaki kodumuzu çeşitli yapılara ayırarak SOLID'in Single Responsibility ilkesine uygun hale getirdik. Bu sayede kodumuz daha yönetilebilir oldu. Yeni bir yapı eklemek daha kolay hale geldi. Kodumuzu geleceğe uygun hale getirdik.

SOLID'in ilk prensibi olan Single Responsibility ilkesini kendimce anlatmaya çalıştım. Değerli yorumlarınızı,eleştirilerinizi bekliyorum. Herkese kolay gelsin!

Kaynakça

> *Temiz Kod — M.Furkan Ardoğan*
>
> *https://www.geeksforgeeks.org/single-responsibility-in-solid-design-principle/*



Edit profile

Written by Süleyman Meral

6 followers · 6 following

Backend Developer(.NET Core)