

SOLID

- R. C. Martin'e göre kötü yazılım;
 - Katıdır; Değişiklik yapmak zordur
 - Kırılgandır; Değişiklikler kırılmalara sebep olur
 - Taşınamazdır; Başka uygulamalarda kullanılamaz

- Nesne Merkezli programlamanın ve tasarımının temel prensipleridir.
 - (S)ingle Responsibility Principle (Tek Sorumluluk Prensibi)
 - (O)pen/Closed Principle (Açık-Kapalı Prensibi)
 - (L)iskov 's Substitution Principle (İkame Edilebilirlik)
 - (I)nterface Segregation Principle (Arayüz Ayırma İlkesi)
 - (D)ependency Inversion Principle (Bağımlılığı Tersine Çevirme)

- Single Responsibility Principle'a göre, her method ve class'ın tek bir görevi ve sorumluluğu vardır. Bir sınıfı değiştirmek için asla birden fazla sebep olmamalıdır.
- Open Closed Principle'a göre her class geliştirmeye açık, değişime kapalı olmalıdır.
- Liskov Substitution Principle'a göre alt sınıflardan oluşturulan nesnelerin üst sınıfların nesneleriyle yer değiştirdiklerinde aynı davranışı göstermek zorundadır. Yani; türetilen sınıflar, türeyen sınıfların tüm özelliklerini kullanmak zorundadır.

- Interface Segregation prensibine göre, her arayüz'ün belirli bir amacı olmalıdır. Tüm metodları kapsayan tek bir interface kullanmaktan kaçınılmalıdır. İstemciler kullanmadıkları arayüze bağımlı olmamalıdır.
- Robert C. Martin'in Dependency Inversion Prensipli'ne göre; Üst seviye (High-Level) sınıflar, alt seviye (Low-Level) sınıflara bağımlı olmamalıdır. İkisi de soyutlamalara bağımlı olmalıdır. Abstraction(soyutlama) detaylara bağımlı olmamalıdır, tam tersi detaylar abstraction(soyutlama)'lara bağımlı olmalıdır.

SRP Seviyeleri

- Paket (package, name space)
- Sınıf
- Metot
- Blok
- Cümle

Cümle Açısından;

- Öncelik karmaşıklığına dikkat edilmelidir.
- Tek cümlelik karmaşık ifadelerin hiç bir tek parçası başka yerde kullanılmamalıdır.
- Bir cümlede hiç bir zaman tekrar olmamalıdır.
- Cümle ya da blok birden fazla yerde bulunuyorsa metoda dönüştürülmeli.

Metot Açısından;

- Metotların ne iş yaptıkları önemlidir. Odaklı olmalıdır.
- Bazı metotlar çekirdek iş yaparlar. Bazı metotlar ise bir süreci yerine getirirler. Süreçsel metotlar sürecin bir adımını yerine getirmelidir.
- Çekirdek metot;
 - bir iş kuralı ve if else kurallarında sadece bir dal
 - tek bir kısıt
 - algoritmik bir hesaplama yapabilir
 - karmaşık bir sürecin tek bir adımı

- Tekrar kullanımı gerektirmeyen bir kod parçasının metot haline getirilmesi ancak private metotlar için söz konusudur.

- Parametre sayısı arttıkça metodun odağı düşer.
- Parametre referans tipi ise ikiyi geçmemelidir. Geçerse odak sorunu teşkil edebilir.

Karar Karmaşıklığı

- Fazla karar birlikteliği düşürür, bağımlılığını artırır.
- Bir koddaki bağımsız akışların sayısıdır.
- Odaklı metot olabildiğince az karar verir.

```
public class SiparisIsleyici {  
    public void siparisIsle(String urun, int adet, boolean stoktaVar, boolean ozelIndirim,  
boolean vipMusteri) {  
        if (stoktaVar) {  
            if (adet > 5) {  
                if (ozelIndirim) {  
                    System.out.println("Sipariş işlendi. Özel indirim uygulandı.");  
                } else {  
                    System.out.println("Sipariş işlendi. Normal fiyat uygulandı.");  
                }  
            } else {  
                if (vipMusteri) {  
                    System.out.println("Sipariş işlendi. VIP müşteriye özel indirim yapıldı.");  
                } else {  
                    System.out.println("Sipariş işlendi. Standart fiyat uygulandı.");  
                }  
            }  
        } else {  
            System.out.println("Ürün stokta yok. Sipariş iptal edildi.");  
        }  
    }  
}
```

```
public class SiparisIsleyici {  
    public void siparisIsle(String urun, int adet, boolean stoktaVar, boolean ozelIndirim, boolean vipMusteri) {  
        if (!stoktaVar) {  
            stokYokHatasi();  
            return; }  
  
        if (adet > 5) {  
            buyukSiparisIsle(ozelIndirim);  
        } else {  
            kucukSiparisIsle(vipMusteri);  
        } }  
  
    private void stokYokHatasi() {  
        System.out.println("Ürün stokta yok. Sipariş iptal edildi."); }  
  
    private void buyukSiparisIsle(boolean ozelIndirim) {  
        if (ozelIndirim) {  
            System.out.println("Sipariş işlendi. Özel indirim uygulandı.");  
        } else {  
            System.out.println("Sipariş işlendi. Normal fiyat uygulandı.");  
        } }  
  
    private void kucukSiparisIsle(boolean vipMusteri) {  
        if (vipMusteri) {  
            System.out.println("Sipariş işlendi. VIP müşteriye özel indirim yapıldı.");  
        } else {  
            System.out.println("Sipariş işlendi. Standart fiyat uygulandı."); } } }
```

Metodu Single Responsibility Yapmak için;

- Bölünebilir,
- Private olarak ayrılabilir.

```
public class RaporOlusturucu {  
    public void raporOlustur(String veri) {  
  
        String islenmisVeri = veriyilsle(veri);  
  
        pdfOlustur(islenmisVeri);  
  
        dosyayaKaydet(islenmisVeri, "rapor.pdf");  
  
        epostaGonder("kullanici@example.com", "rapor.pdf");  
    }  
  
    private String veriyilsle(String veri) {  
        return "İşlenmiş: " + veri;  
    }  
  
    private void pdfOlustur(String veri) {  
        System.out.println("PDF oluşturuldu: " + veri);  
    }  
  
    private void dosyayaKaydet(String veri, String dosyaAdi) {  
        System.out.println(dosyaAdi + " dosyasına kaydedildi: " + veri);  
    }  
  
    private void epostaGonder(String alici, String ekDosya) {  
        System.out.println(alici + " adresine e-posta gönderildi. Ek: " + ekDosya);  
    }  
}
```

```
public class RaporOlusturucu {  
    public void raporOlustur(String veri) {  
        String islenmisVeri = veriyilsle(veri);  
        raporuOlusturVeKaydet(islenmisVeri);  
        kullaniciyiBilgilendir("kullanici@example.com", "rapor.pdf"); }  
  
    private String veriyilsle(String veri) {  
        return "İşlenmiş: " + veri; }  
  
    private void raporuOlusturVeKaydet(String veri) {  
        pdfOlustur(veri);  
        dosyayaKaydet(veri, "rapor.pdf"); }  
  
    private void kullaniciyiBilgilendir(String alici, String ekDosya) {  
        epostaGonder(alici, ekDosya); }  
  
    private void pdfOlustur(String veri) {  
        System.out.println("PDF oluşturuldu: " + veri); }  
  
    private void dosyayaKaydet(String veri, String dosyaAdi) {  
        System.out.println(dosyaAdi + " dosyasına kaydedildi: " + veri); }  
  
    private void epostaGonder(String alici, String ekDosya) {  
        System.out.println(alici + " adresine e-posta gönderildi. Ek: " + ekDosya); }  
}
```

Sınıf için;

- Sadece tek bir sorumluluğu bir araya getirmelidir.
- Sınıfın birden fazla üst tipinin olması bileşik/küme sınıflar olarak isimlendirilebilir.

Sınıfı Single Responsibility Yapmak için;

- Bölünmelidir.
- Yeni soyutlamalar yapılmalı.
- Interface Segregation gibi prensiplerle bölme yapılabilir.
- Sınıfın birden fazla üst tipi olabilir.

```
public class Customer
{
    public int Id { get; set; }
    public string Name { get; set; }

    public void CreateCustomer(Customer customer)
    {
        // Customer created !
    }

    public void SaveToFile(Customer customer)
    {
        // Customer saved to file !
    }
}
```

```
public class File
{
    public void SaveToFile(Customer
customer)
    {
        // Customer saved to file !
    }
}
```

Open Closed Principle

- Yazılım genişlemeye açık, değiştirmeye (kaynak kodunda) kapalı olmalıdır.
- İdeal prensip olmasına rağmen çok gerçekleştirilebilir değil.
- Amaç tekrar kullanımı artırmak ve bakımı kolaylaştırmak.
- Genişleme;
 - Baştan değişebilecek kısımlar değişmeyecek kısımlardan ayrılarak yapılandırılması
 - Bağımlılıkların soyutlamalar üzerinden yapılması.

- Var olan metotların override edilerek yeni davranışlar elde edilebilmeli.
- OCP için;
 - Düşük birliktelikten
 - İç yapılara ve somut tiplere bağımlılıktan
 - Global değişkenlerden
 - Çalışma zamanı tip tanımlamalarından kaçınılmalıdır.

- Sistemin ve parçalarının nasıl evrilebileceğini öngörmek ürün, piyasa ve kullanıcı davranışlarına göre stratejik olarak planlamalıdır. Bunun için diğer paydaşlar düşünmeye zorlanmalı.
- Temelde iş mimarisini belirler.
- En sık değişenler en kolay değişmeli.
- Takım değişime, değişimlerin maliyetine göre karar verilmelidir.

```
public class File
{
    public void SaveToFile(Customer customer)
    {
        if (TypeOfFile == "txt")
        {
            // Save to txt file !
        }

        if (TypeOfFile == "xls")
        {
            // Save to xls file !
        }

        // ... and more file types
    }
}
```

```
    abstract class File
    {
        public abstract void SaveToFile(Customer customer);
    }
```

```
class TxtFile extends File
{
    public override void SaveToFile(Customer customer)
    {
        // Saved to txt file !
    }
}
```

```
class XlsFile extends File
{
    public override void SaveToFile(Customer customer)
    {
        // Saved to xls file !
    }
}
```

Liskov Substitution Principle

- Miras ilişkisi kurulunca temelde gerçek tipleri ve detayları bilmeden kullanılabilmelidir.
- Polymorphism gibi.

LSP'de Kaçınılması Gerekenler

- Yerine geçmede, anlamsal uyumu göz önünde bulundurarak, problem çıkaran durumlardan kaçınılmalıdır (Kare sınıfın dikdörtgenden miras alması LSP'ye uymaz).
- Çalışma zamanı tip tanımlamaları.

- Yazılım bileşenleri için formal, kesin ve doğrulanabilir arayüzler tasarlanmalı.
- Bu amaçla ön ve son şartlar söz konusudur. Alt tipler ön şartları zayıflatabilir, son şartları sıkılaştırabilir. Bu hiyerarşi tersine çevrilmemelidir.

```
public abstract class Player
{
    public virtual void KickTheBall()
    {
        // Ball was kicked !
    }

    public virtual void KeepTheBall()
    {
        // Ball was kept !
    }
}

public class Striker : Player
{
    public override void KeepTheBall()
    {
        // Striker should not keep the ball !
        throw new NotImplementedException();
    }

    public override void KickTheBall()
    {
        // Ball was kicked by Striker !
    }
}

public class Goalkeeper : Player
{
    public override void KeepTheBall()
    {
        // Ball was kept by Goalkeeper!
    }

    public override void KickTheBall()
    {
        // Ball was kicked by Goalkeeper !
    }
}
```

- Kodda Striker aslında ihtiyaç duymadığı KeepTheBall metodunu barındırmakta yani bir forvet oyuncusu topu eliyle tutamaz. Normal şartlarda bu fonksiyonu kullanamayacak, bu fonksiyonda exception fırlatması gerekecektir.
- Yani aslında gereksiz bir kod kalabalığı ve kod yönetimi açısından ek bir efor oluşacaktır, bunun nedeni base classda aslında gereksiz bir metod bulunmasıdır.

```

        public abstract class Player
    {
        public virtual void KickTheBall()
        {
            // Ball was kicked !
        }
    }

    public interface IKeepTheBall
    {
        void KeepTheBall();
    }

    public class Striker : Player
    {
        public override void KickTheBall()
        {
            // Ball was kicked by Striker !
        }
    }

    public class Goalkeeper : Player, IKeepTheBall
    {
        public override void KickTheBall()
        {
            // Ball was kicked by Goalkeeper !
        }

        public void KeepTheBall()
        {
            // Ball was kept by Goalkeeper !
        }
    }

```

Interface Segregation Principle

- Arayüzleri ayırma prensibidir.
- İstemciler kullanmadıkları arayüze bağımlı olmamalıdır.
- Bir arayüzün farklı istemcilere hizmet veren metotlardan oluşuyorsa arayüzler bölünmelidir.

- Birden fazla arayüz devralan sınıflar şişman ya da kirli diye nitelendirilse de bazen sınıfları fazla arayüzle tasarlamak gerekebilmektedir.

```
interface IPost
{
    void CreatePost();
    void ReadPost();
}
```

```
interface IPostCreate
{
    void CreatePost();
}
```

```
interface IPostRead
{
    void ReadPost();
}
```

Dependency Inversion Principle

- Yüksek seviyeli modüller aşağı seviyeli modüllere bağımlı olmamalıdır. İkisi de soyutlamalara bağımlı olmalıdır.
- Katmanlı yapıdaki bağımlılıkta en temel problem, alt seviyedeki modüllerdeki değişikliklerin üst seviyedeki modülleri etkilemesidir.
- İstenen yukarı seviye modüllerin değişimi belirlemesidir.
- Çözüm somut bağımlılıkların soyut bağımlılığa dönüştürülmesidir. Her katman soyut bir arayüzle temsil edilmelidir. Böylece hiçbir katman diğer katmana doğrudan bağlanamayacaktır.

```
class Blog
{
    // High Level Class
    public void Create()
    {
        Post post = new Post();
        post.CreatePost(true);
    }
}
```

```
class Post
{
    // Low Level Class
    public void CreatePost(bool picture)
    {
        // Process
    }
}
```

- Blog sınıfı içerisindeki Create metodu, Post sınıfı içerisindeki CreatePost isimli metoda bağımlıdır. Bunun sebebi CreatePost method'unun Create içerisinde kullanılmasıdır. Bu metotda yapılacak tüm değişiklikler Create method'unda da değişiklik gerektirecektir.

```
interface IContent
{
    void CreatePost(bool picture);
}

class Blog
{
    //High Level Class
    IContent content;
    public Blog()
    {
        content = new Post();
    }
    public void Create()
    {
        content.CreatePost(true);
    }
}

class Post : IContent
{
    //Low Level Method
    public void CreatePost(bool picture)
    {
        //Process
    }
}
```

Demeter Law (En Az Bilgi Prensibi)

- **Demeter Yasası**, bir nesnenin yalnızca doğrudan bildiği nesnelerle etkileşimde bulunmasını önerir. **“Talk to friends, not to strangers”** (Arkadaşlarıyla konuş, yabancılarla değil) prensibine dayanır.
- Bir nesne başka bir nesnenin iç detaylarına doğrudan erişmemeli, sadece doğrudan ilişkili nesneleri kullanmalıdır.
- Nesneler birbirlerini arayüzleri üzerinden bilirler.
- Nesneler gerçek tipleri değil arayüz tipleri bilirler.
- Nesneler birbirlerinin gerçekleştirme detaylarını bilmemelidirler.

- Bir nesne olabildiğince kendisinden hizmet alan az sayıda nesneyi bilmeli.
- Nesnenin metodu, kendisinden çağrı yapabileceği nesneler;
 - nesnenin instance değişkenleri
 - nesnenin metotlarına gönderilen nesneler
 - nesnenin metodunda oluşturulan nesneler

- Demeter'e göre;
 - her yazılım birimi diğer birimlerle ilgili sınırlı bilgiye sahip olmalı
 - sınırlı bilgi birimin yakınındaki birimlerle ilgili olmalı
 - her birim sadece yakınıyla konuşmalı
- Data alma hizmet al

- `String city = order.getCustomer().getAddress().getCity();`
- Aşırı Bağımlılık (Coupling): Kodun çalışması için Order'ın Customer'ı, Customer'ın Address'i, Address'in de City'yi bildiğini teyit etmesi gerekir.
- Kırılganlık: Eğer gelecekte Address sınıfındaki `getCity()` metodu `getLocation().getCity()` olarak değişirse, bu zinciri kullanan her yer patlar.
- Gizliliğin İhlali: Order nesnesi, Customer nesnesinin iç yapısı (adres bilgisi gibi) hakkında çok fazla bilgiye sahip olmuş olur.

- `String city = order.getDeliveryCity();`
- Order sınıfı: `customer.getCity()` çağrısını yapar.
- Customer sınıfı: `address.getCity()` çağrısını yapar.
- Address sınıfı: Kendi içindeki city bilgisini döner.

```
class Motor {  
    public void calistir() {  
        System.out.println("Motor çalıştırıldı.");  
    }  
}
```

```
class Araba {  
    private Motor motor;  
  
    public Araba() {  
        this.motor = new Motor();  
    }  
  
    public Motor getMotor() { // Demeter  
        Yasasına Aykırı: Dış nesneler motor nesnesine  
        doğrudan erişebilir.  
        return motor;  
    }  
}
```

```
class Surucu {  
    public void arabayiCalistir(Araba araba) {  
        araba.getMotor().calistir(); // Surucu,  
        Araba'nın içindeki Motor nesnesini doğrudan  
        kullanıyor.  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Araba araba = new Araba();  
        Surucu surucu = new Surucu();  
        surucu.arabayiCalistir(araba);  
    }  
}
```

```
class Motor {  
    public void calistir() {  
        System.out.println("Motor çalıştırıldı.");  
    }  
}
```

```
class Araba {  
    private Motor motor;  
  
    public Araba() {  
        this.motor = new Motor();  
    }  
  
    public void arabayiCalistir() { // Arabayı  
        çalıştırmak için dışarıya sadece gerekli metodu  
        açıyoruz.  
        motor.calistir();  
    }  
}
```

```
class Surucu {  
    public void arabayiCalistir(Araba araba) {  
        araba.arabayiCalistir(); // Sürücü artık  
        arabayla konuşuyor, motorun iç detaylarını  
        bilmiyor.  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        Araba araba = new Araba();  
        Surucu surucu = new Surucu();  
        surucu.arabayiCalistir(araba);  
    }  
}
```

GRASP (GENERAL RESPONSIBILITY ASSIGNMENT SOFTWARE PATTERNS-GENEL SORUMLULUK ATAMA YAZILIM KALIBLARI)

- Yazılımda sorumluluk atama ile ilgili genel geçer şablonlar(prensipier) demektir. Nesne yönelimli tasarımda sorumlulukların nasıl dağıtılacağını belirleyen rehber kurallar sunar.
- Nesne tasarımı ve sorumluluk atamasında dokuz temel pattern'dir.
 - Controller,
 - Creator,
 - Indirection,
 - Information Expert,
 - High Cohesion,
 - Low Coupling,
 - Polymorphism
 - Protected Variations,
 - Pure Fabrication.

- Controller(Kontrol edenin belirlenmesi):** Nesnelerin yaptıkları iş itibari ile nesneler arasında olası bir iş kırılımı olması olasılığı var ise; o iş kırılımlarını kontrol edecek bir controller yazılmalıdır. Aktiviteyi (eventlar) control eder.
- Creator(Oluşturucu):** Bir nesneyi kimin oluşturması gerektiğine karar verir. Eğer B nesnesi A'yı içeriyorsa, kullanıyorsa veya verisine sahipse, A'yı B oluşturmalıdır.
- Indirection(Dolaylı,endirekt bağımlılık):** Nesne gruplarının kendine has yaptığı işlerde diğer nesnelere bağımlılığının azaltılmalıdır. İki nesne arasında olan iletişimi direk bağımlı yapma araya ara eleman (interface veya sınıf) koy.
- Information Expert(Bilginin uzmanı):** Nesnelere sorumluluk atarken; bir işi , o işin uzmanı olan nesneye atama gerekliliğidir. Temel nesne (encapsulation) tanımını vurgular ve veri bağımlılıklarını ortadan kaldırmayı hedefler.
- High Cohesion(Yüksek birliktelik):** Nesnelerin sorumluluklarının; birbiriyle çok alakalı olan işler ile ilgi olması gerektiğidir. Fonksiyonel birlikteliği vurgular.

- Low Coupling(Düşük bağımlılık):** Bir nesnenin yaptığı iş itibari ile diğer nesnelere en az bağımlı olması gerektiğidir. Information Expert gibi veri bağımlılığını ortadan kaldırmayı amaçlar. Veriyi gezdirmek yerine en yakın yerde işlemeyi hedefler.
- Polymorphism(Çok çeşitlilik):** Bir nesnenin ortak özelliklerinin, çok çeşitli olarak kullanılabilecek şekilde tasarlanmasıdır. If yerine poly kullan.
- Protected Variations (Güvenli,birliktelik oluşturan Çeşitlemeler):** Nesneleri, birliktelik oluşturacak şekilde ('yabancılarla konuşmayan' gibi), olabildiğince kullanıma açık, değiştirilebilmeye kapalı olarak tasarlamamanın gerektiğidir. (Örnek : Open-Closed prensibi). Değişenlerle değişmeyenleri olabildiğince ayırmalıyız.
- Pure Fabrication(Saf fabrikasyon):** Bazen "Bilgi Uzmanı" prensibi, sınıfın bağıllığını (cohesion) bozar. Bu durumda, gerçek dünyada karşılığı olmayan ama tasarımı rahatlatan yapay sınıflar oluşturulur. Örneğin Veritabanına kayıt yapan bir DAO veya Repository sınıfı gerçek dünyada yoktur ama kod organizasyonu için şarttır.

Kaynakça

- <https://dijitalseruven.com/>
- <https://www.merhabadunya.net/>
- Akın Kaldıroğlu ders notları