YZM 3017
Yazılım Tasarımı ve Mimarisi

Prof. Dr. Hamdi Tolga KAHRAMAN
Arş. Gör. M. Hakan BOZKURT
Arş. Gör. Sefa ARAS

Nesneye Dayalı Tasarım Prensipleri

Yazılım Tasarımı ve Mimarisi

Hatırlatma (Hazırlık) Soruları

1) *Polymorphism, Encapsulation, Inheritance* kavramları nelerdir detaylı açıklayınız? Farkları nelerdir?

2) *Interface* ve *Abstract* sınıflar nelerdir. Kod (C#) ve UML gösterimleri nasıldır?

3) UML' de *Association, Aggregation , Composition, Generalization, Realization Dependency* ve *Use*ilişkilerini açıklayıp , gösterimlerini çiziniz. Interface

Kötü tasarımın belirtileri

- Bir yazılımın kalitesi ölçülürken çoğu zaman yanlış bir yaklaşımla performansı, hatta sadece görsel olarak ne kadar anlamlı olduğu dikkate alınmaktadır. Oysa bu iki faktör kesinlikle kaliteyi ölçmek için yeterli değildir.

- Esneklik, güvenlik, güvenilirlik, ölçeklenebilirlik, modülarite, adaptasyon kabiliyeti gibi çok önemli başka faktörler de yazılım geliştirirken göz önünde tutulmalıdır.

- Bir yazılımın tüm bu karakteristikler bakımından iyi olmasının yolu ise belirli disiplinlere uygun kod yazımından geçer.

Yazılım Tasarımı ve Mimarisi

1.Esnemezlik(Rijidite)

- Bir yazılım sistemi zaman içinde değişen veya yeni gündeme gelen gereksinimleri karşılayabilmelidir. Eğer bir değişim ya da gelişim yapılması isteniyorsa ve sistem bu duruma aşırı direnç gösteriyorsa bu rijit bir sistemdir.
- İyi bir tasarımda değişime karşı olan direncin düşük olması istenir.

2.Kırılganlık(Fragility)

- Amaç, sistemi her zaman olabildiğince esnek tasarlamak ve muhtemel değişikliklere daha az direnç gösteren dolayısıyla daha az kırılgan yazılımlar geliştirebilmektir.
- Aksi takdirde yapılacak bir değişiklik sistemde domino etkisi yaratıp zincirleme şekilde birden çok şeyin değişmesini gerektirebilir.

3.İmmobilite

- Yazılım sistemleri modüler şekilde tasarlanmalıdır. Bir projede kullanılmış olan herhangi bir bileşenin farklı bir projeye taşınıp orada da kullanılabilmesi «yeniden kullanılabilirlik(reusability)» olarak bilinir

Yazılım Tasarımı ve Mimarisi

LCP : Zayıf Bağlaşım Prensibi (Low Coupling Principle)

- Çoğu zaman birbirleriyle ilişki ve iletişim halinde olan nesneler birbirlerini içerebilir ya da çeşitli biçimlerde kullanabilirler. İşte coupling; nesneler arasındaki böylesi ilişkilerin nesneleri birbirlerine ne kadar bağlı kıldığının bir ölçüsüdür.

- Kendi içinde kohezyonu yüksek nesnelerin birbirleriyle ilişkiler kurması halinde bu ilişkideki coupling faktörünün olabildiğince düşük tutulması arzu edilir.

Yazılım Tasarımı ve Mimarisi

- Coupling faktörünün tanımlı 5 seviyesi vardır:

1. **Nil Coupling:**
   - Teorik olarak en düşük dolayısıyla en iyi coupling düzeyidir. Zira bu seviyede bağımlılık söz konusu değildir. Sadece diğer sınıflarla hiçbir ilgisi olmayan, tek başlarına kullanılan sınıflar bu duruma örnek teşkil edebilir.
2. **Expert Coupling:**
   - Herhangi bir sınıf başka bir sınıfa ortak bir arayüzle bağlıysa aralarında export coupling oluşur. Birçok durumda ulaşılmaya çalışılan seviye export coupling seviyesidir.
3. **Overt Coupling:**
   - Bir sınıf başka bir sınıfa ilişkin üyeleri belli bir izin dahilinde kullanıyorsa aralarında overt coupling söz konusudur.

Yazılım Tasarımı ve Mimarisi

4. **Covert Coupling:**
   - Overt coupling'te X sınıfı Y sınıfının private üyelerine erişim izni vermektedir. Covert coupling'te ise herhangi bir izin olmaksızın erişim söz konusudur.

5. **Surreptitious (Gizlice) Coupling:**
   - X sınıfı Y sınıfına ait içsel detayların tümünü biliyorsa 5. Seviyede coupling oluşur. Örneğin, X sınıfının, Y sınıfına ait public veri elemanlarını kullanarak çeşitli işlemler yapıyor olması bu duruma yol açar. Bu çok arzu edilmeyen hatta tasarım açısından tehlikeli kabul edilen bir coupling seviyesidir. Zira bağımlılık çok fazla olacaktır.

- Aşağıdaki örnek incelendiğinde Logger ve Sample nesneleri arasında sıkı bir bağlılığın(bağımlılığın) olduğu görülebilir.

Örnek :

```
class Sample
{
    private Data m_Data;
    public void Foo()
    {
        Logger lg= new Logger();
        lg.Log(m_Data);
        //...
    }
```

Yazılım Tasarımı ve Mimarisi

- log'lama görevini gerçekleştiren fonksiyon olan log(), muhtemelen farklı biçimlerde log'lama yapması gerekecek bir fonksiyondur. Söz gelimi başta sadece dosyaya log'lama yapılıyorken, zaman içinde veri tabanı ya da Windows EvenLog'a da log'lama yapılmak istenebilir. Bu durumda Sample'ın Logger'a olan bağımlılığı yüzünden Logger sınıfında değişiklik yapmak büyük sorunlara neden olacaktır. Oysa aşağıdaki gibi bir tasarım bu sorunlara bir çözüm getirebilir.

```
interface Ilogger
{
    void Log(object obj);
}
class DBLogger : ILogger
{
    //...
}
class FileLogger : ILogger
{
    //...
}
class LogFactory
{
    public static ILogger GetLogger(...)
    {
        if(...)
        {
            return new DBLogger();
        }
        if(...)
        {
            return new FileLogger();
        }
        //...
    }
}
```

Yazılım Tasarımı ve Mimarisi

```
class Sample
{
    private Data m_Data;
    public void SetData(Data dt)
    {
        this.m_Data = dt;
    }
    public void Foo()
    {
        Ilogger lg= LogFactory.GetLogger(...);
        lg.Log(m_Data);
    }
}
```

- Bu şekilde yani interface kullanımıyla coupling(bağlaşım) en aza indirgenmiş olur.
- Bu noktada çoğu zaman programcıların aklını interface veya soyut sınıf tercihi karıştırmaktadır. Bu durumlarda çoklu türetmenin olmadığı C#, Java, VB.NET gibi dillerde interface kullanımı tercih edilmelidir. Ancak çoklu türetmenin olduğu, öte yandan interface'lerin olmadığı C++ dilindeyse aynı şey yine sanal fonksiyonların kullanımıyla gerçekleştirilebilir.

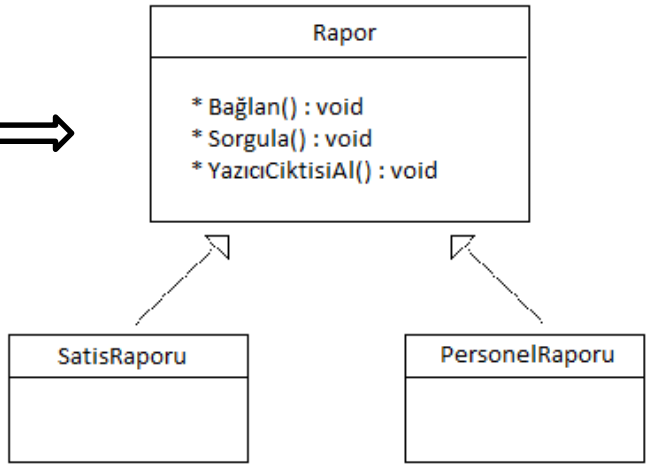
Yeniden Kullanılabilirlik (Reusability) Prensibi

- Yeniden kullanılabilirlik nesne yönelimli programlamanın en önemli özelliklerinden biridir ve çoğu zaman, üstelik de yanlış bir yaklaşımla sadece "inheritance" yani türetmeden ibaret olduğu sanılır. Oysa türetme yeniden kullanılabilirliği sağlama noktasında hem her zaman yeterli hem de doğru olmaz.
- Salt türetme yerine türetmeyle birlikte nesneler arasında kompozisyon kurgulanmalıdır.
- Kompozisyon bir nesnenin başka nesneleri içermesi biçiminde düşünülmelidir.
- **Örneğin** bir raporlama sınıfı yazılacaksa, veri tabanı ve yazıcıdan çıktı alma işlemlerinin bu sınıf içerisinde yazılması, rapor nesnesine gereğinden fazla sorumluluk yükler. Bu sınıftan yapılacak türetmeler de yeniden kullanıma doğru bir örnek teşkil etmeyecektir.

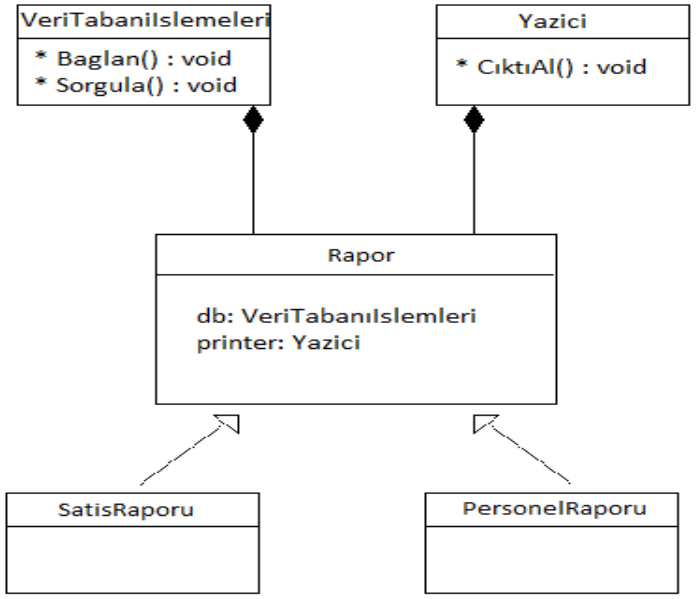
Yazılım Tasarımı ve Mimarisi

Örnek

Şekil 1.4 Yanlış Tasarım



Şekil 1.5 Doğru Tasarım

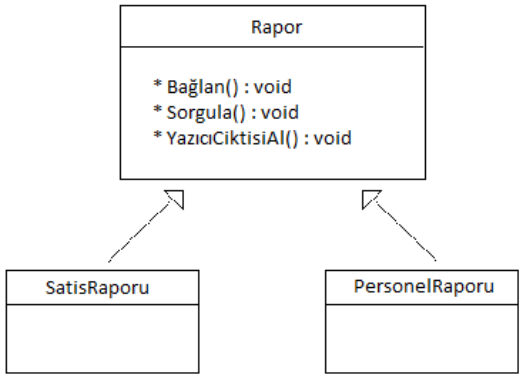


- Doğru olan yaklaşım; rapor sınıfının içinde bu işlemleri kodlamak yerine, veritabanı ve yazıcıdan çıktı alma işlemlerinin ayrı ayrı nesnelere delege edilmesidir. Daha sonra bu nesneler rapor nesnesi içinde yer alarak bir kompozisyon oluşturmalıdır.

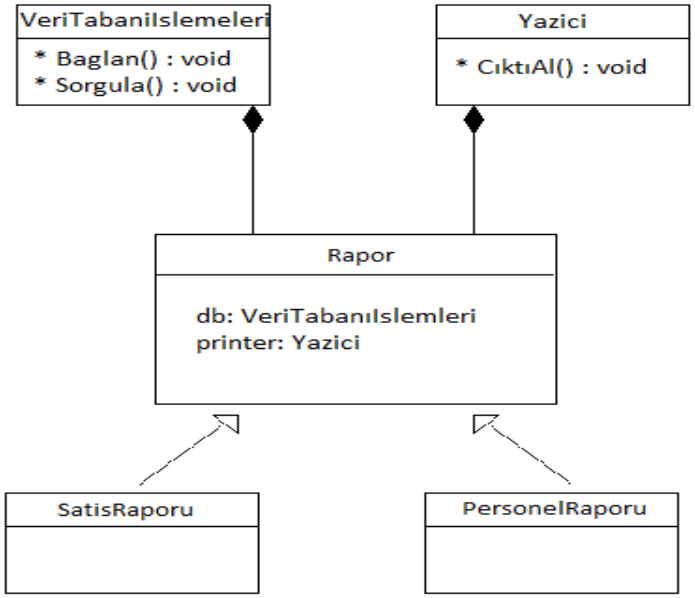
Yazılım Tasarımı ve Mimarisi

Örnek :

Şekil 1.4 Yanlış Tasarım



Şekil 1.5 Doğru Tasarım



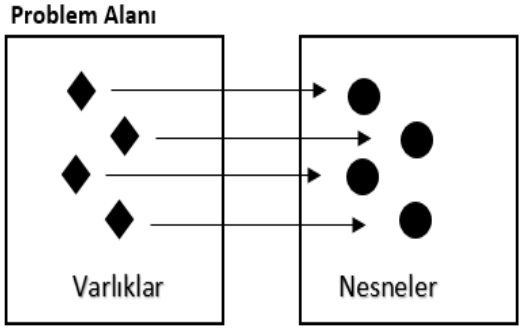
Black Box ve White Box Kavramları

- GoF doktrininde yeniden kullanılabilirlik bu kavramlarla ikiye ayrılarak ele alınmıştır.
- White Box sınıf türetmesine dayalıdır ve salt türetme bazı tasarımsal sakıncalar doğurabilir.
- Black Box ise nesne kompozisyonuna dayalıdır. Yani raporlama temasındaki Rapor sınıfı gibi bir wrapper sınıf, VeriTabanıIslemleri ve Yazici sınıflarına ait nesneleri kendi içinde örneklemekte, böylece bir kompozisyon oluşturarak, “black box reusability” sağlamaktadır.

Yazılım Tasarımı ve Mimarisi

1. Ayrıştırma(Decomposition)

- Bir yöntem olarak da nitelendirilebilecek ayrıştırma, sistemdeki karmaşıklıkla(complexity) başa çıkabilmek için yapılır. Buna kısaca *böl ve fethet* denilmektedir.



Şekil 1.1

- Yazılımcının çözümlemeyi gerçekleştirirken en büyük yardımcısı analiz sırasında ortaya çıkan use case tanımlarıdır. UML ile ilgili bölümde de ele alındığı gibi use case tanımlarını isim ve fiilleri ortaya çıkaracak şekilde okumak olası nesnelerin tespitine yardımcı olacaktır.
- Yazılımcı «nasıl?» sorusu yerine «ne(ler)?» sorusunun cevabına odaklanmalıdır. Böylece algoritma düşünmek yerine nesneleri düşünmeye yoğunlaşabilir.

2. Kohezyon(Yapışıklık)

- Yazılım mühendisliğinde kohezyon; bir sınıftaki tanımlı üyelerin birbirlerine mantıksal ilişkilenmesi biçiminde tanımlanabilir. Diğer bir deyişle kohezyon, sınıf içerisindeki üyelerin ya da fonksiyonlar içindeki görevlerin birbirlerine olan mantıksal uzaklığını belirler. Tasarımda kohezyon faktörünün yüksek olması arzu edilir.

- Örneğin, string işlemleri için yazılmış bir sınıf, string'in ekrana basılmasını sağlayan fonksiyonlara sahip olmamalıdır. Her şeyden önce böylesi yanlış bir tercih, o sınıfı sadece belirli GUI sistemleriyle çalışmaya bağımlı kılacaktır. Öte yandan o string sınıfı içinde yazılan ve string işlemleriyle doğrudan ilişkili fonksiyonlar ise olması arzu edilen yüksek kohezyonu yaratacak bir tercihtir.

1. Tasarım Prensipleri

Yazılım Tasarımı ve Mimarisi

3. SOLID

- SOLID (**S**ingle responsibility, **O**pen-closed, **L**iskov substitution, **I**nterface segregation ve **D**ependency inversion) yazılım tasarım prensipleri için kullanılan bir kısaltmadır.

- Yazılım yaparken SOLID uygulandığı taktirde bakımı ve geliştirilmesi kolay yazılım sistemleri oluşturmak mümkündür.

Yazılım Tasarımı ve Mimarisi

SOLID

S SRP Single Responsibility Principle
Her yazılım biriminin (sınıf, nesne, metot) tek bir sorumluluğu olmalıdır.

O OCP Open/Closed Principle
Yazılım birimleri geliştirilmeye açık, değişikliğe kapalı olmalıdır.

L LSP Liskov's Substitution Principle
Alt sınıflardan oluşturulan nesneler üst sınıfların nesneleriyle yer değiştirdiklerinde aynı davranışı göstermek zorundadırlar.

I ISP Interface Segregation Principle
Herşeyi ihtiva eden interface sınıflar yerine belli bir işlemi yapan interface sınıflar oluşturulmalıdır.

D DIP Dependency Inversion Principle
Bağımlılıklar soyut sınıflara doğru olmalıdır

Yazılım Tasarımı ve Mimarisi

SOLID, SRP: Tek Sorumluluk Prensibi (Single Responsibility Principle)

- Kohezyon olgusuyla yakından ilişkili olan bu prensip, bir modülün(örneğin sınıfın) sadece tek bir sorumluluğu yerine getirmek üzere tasarlanmasını öngörür.

- Pratikte sorumluluk ya da görevleri ayrıştırabilmek zordur. Ancak tasarımcısı bunu becerebilirse ileride yaşanacak olası bir değişiklik için sadece değişen durumun sorumluluğunu üstlenmiş sınıfı değiştirmek şansına sahip olur.

- Bir sınıfı değiştirmek için sadece tek bir gerekçeniz olmalıdır. Şayet birden fazla gerekçe söz konusuysa o sınıfın bölünmeye ihtiyacı var demektir. Çünkü değişiklik ihtiyacı daima bir tek nedene dayandırılmalıdır.

SRP: Tek Sorumluluk Prensibi (Single Responsibility Principle)

- Birden fazla değişiklik olduğu zaman bunun etkisi ondan türetilen sınıflar üzerinde de olacaktır ve kontrolsüz gelişim yaşanır, etkileri anlaşılamayabilir. Eklenen her sınıfın tek bir sorumluluğu olması esnek bir tasarım olmasını sağlar.

- Tek sorumluluk prensibi (Single-Responsibility Principle) amacı; bir sınıf, interface, fonksiyon vs. sadece tek bir sorumluluk yerine getirmelidir. Bu sorumluluk sadece kendi ile ilgili işleri yapmalıdır. Bir sınıf, interface, fonksiyon vs. birden fazla sorumluluk yerine getirmeye çalıştığı zaman aşırı büyür ve karmaşıklaşır.

Yazılım Tasarımı ve Mimarisi

SRP: Tek Sorumluluk Prensibi (Single Responsibility Principle)

Örneğin bir projemiz var ve projemizin görevlerinden bir bölümü çek, fatura, senet ile ilgili işlemler yapıyor. Biz ilk aşamada bu işlemleri tek çatı altında toplayarak projemizi geliştirelim.

Tasarımı değerlendiriniz.

```
public class BasvuruIslemleri
  {
      public string Cek { get; set; }
      public string Fatura { get; set; }
      public double Senet { get; set; }


      public void CekIslem(string cekBilgileri)
      {
          //çek işlemleri
      }
      public List<string> CekListesiGetir()
      {
          //çek listesi için
      }
      public void FaturaBas()
      {
          //fatura yazdırmak için
      }
      public bool SenetKontrol()
      {
          //senet geçerliliği kontrolü için
      }
      public void SenetIptalIslemleri(string malzeme)
      {
          //işlemi biten senet'i iptal etmek için
      }
  }
```

Yazılım Tasarımı ve Mimarisi

SRP: Tek Sorumluluk Prensibi (Single Responsibility Principle)

Görüldüğü gibi 3 farklı olay tek bir çatıda toplanmıştır. Bu işlemlerin ileride çok daha geliştiğini ve metod sayımız arttığını düşünelim. Bu sınıf ileride çağrılırken , yeni değişikliklerden etkileneceği ve daha önce çalışan kodu bozma ihtimalini; kod bulma, okuma, geliştirme aşamasında yaşanılan zorlukları göz önüne alındığında SRP yani tek sorumluluk prensibini kullanmamız gerektiğini kolayca anlayabiliyoruz.

```
public class BasvuruIslemleri
  {
      public string Cek { get; set; }
      public string Fatura { get; set; }
      public double Senet { get; set; }


      public void CekIslem(string cekBilgileri)
      {
          //çek işlemleri
      }
      public List<string> CekListesiGetir()
      {
          //çek listesi için
      }
      public void FaturaBas()
      {
          //fatura yazdırmak için
      }
      public bool SenetKontrol()
      {
          //senet geçerliliği kontrolü için
      }
      public void SenetIptalIslemleri(string malzeme)
      {
          //işlemi biten senet'i iptal etmek için
      }
  }
```

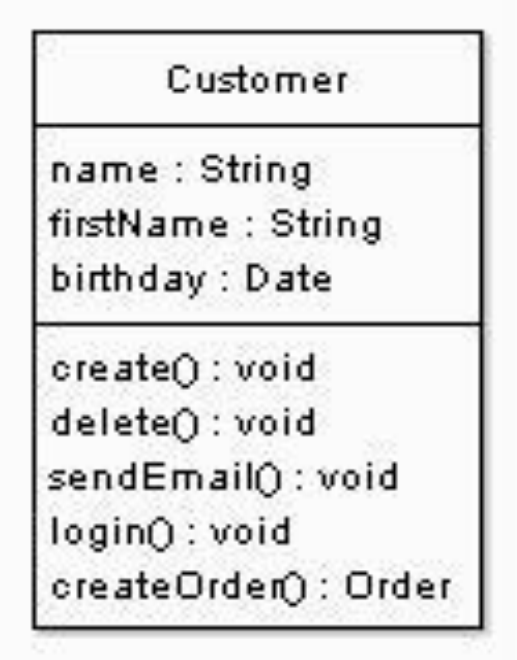
SRP: Tek Sorumluluk Prensibi (Single Responsibility Principle)

Tek sorumluluk prensibine göre sorumluluklar ayrı sınıflara atandı

```
public class BasvuruIslemleri
    {
        public string Cek { get; set; }
        public string Fatura { get; set; }
        public double Senet { get; set; }
    }

 public class Cek
    {
        public void CekIslem(string cekBilgileri)
        {
            //çek işlemleri
        }
        public List<string> CekListesiGetir()
        {
            //çek listesi için
        }
    }

 public class Fatura
    {
        public void FaturaBas()
        {
            //fatura yazdırmak için
        }
    }


 public class Senet
    {
        public bool SenetKontrol()
        {
            //senet geçerliliği kontrolü için
        }
        public void SenetIptalIslemleri(string malzeme)
        {
            //işlemi biten senet'i iptal etmek için
        }
    }
```

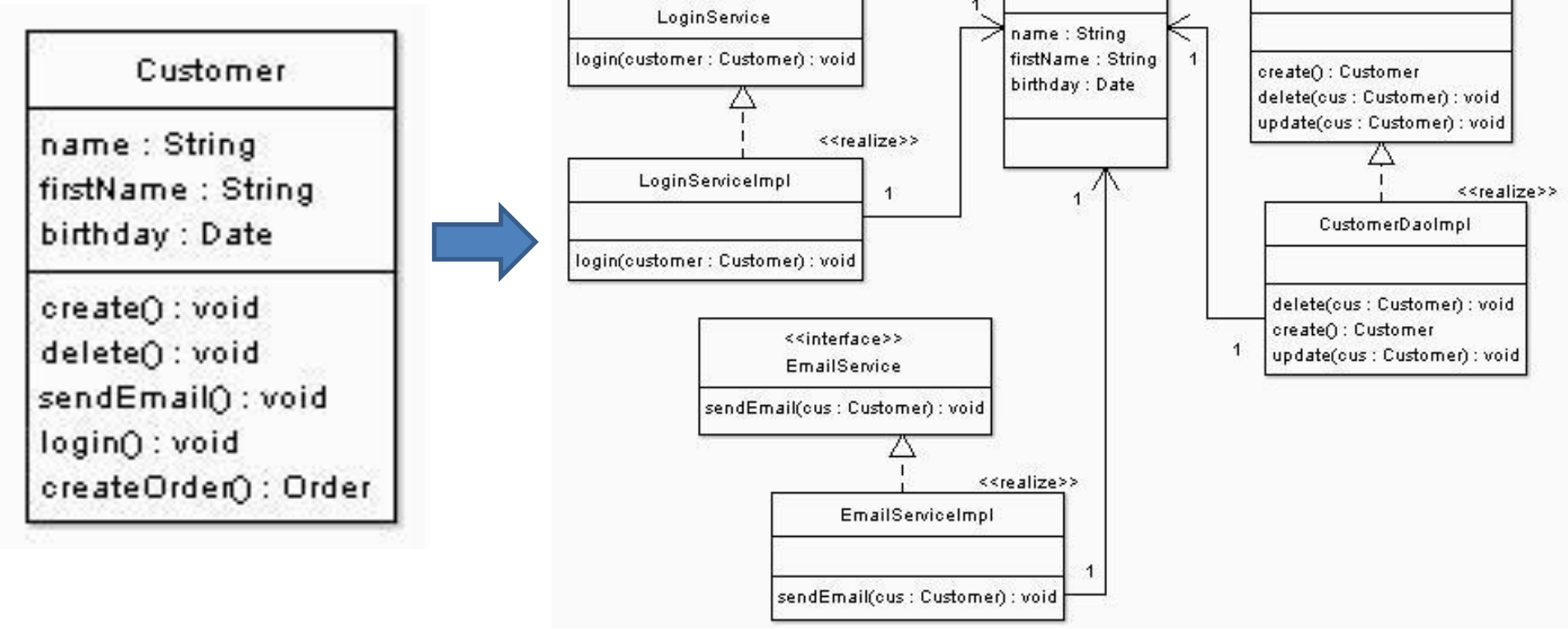
SRP: Tek Sorumluluk Prensibi (Single Responsibility Principle)

- Aşağıda verilen örnek sınıf kendisini bilgi bankasına ekleme, silme, müşteriye e-mail gönderme, login yapma (shop sistemi olabilir) ve sipariş oluşturma işlemlerini yapabilmektedir.



Yazılım Tasarımı ve Mimarisi

SRP: Tek Sorumluluk Prensibi (Single Responsibility Principle)



Customer sınıfı sahip olduğu sorumlulukların başka sınıflara yüklenmesiyle hafiflemiş ve değişikliklere karşı daha dayanıklı bir hale gelmiştir. Bunun yanı sıra bu sınıfın test edilebilirliği yükselmiştir. Bu ve diğer sınıfların değişmek için artık sadece bir sebebi vardır. Bunun sebebi sadece bir sorumluluk sahibi olmalarında yatmaktadır.

Veritabanından bir hafta önce alışveriş yapan kullanıcılarımızı alarak bu kullanıcılara indirimde olan ürünleri e-mail olarak göndermek istediğimizi varsayalım.

```php
<?php

class EmailSender {
  // sorumluluk 1 - asıl sorumlu olduğu bu işlem!
  public function send(array $users, array $products)
  {
    // kullanıcılara indirimde olan ürünleri liste halinde e-mail gönderdiğimiz
  }

  // sorumluluk 2 - ekstra sorumluluk
  public function getUsers()
  {
    return '...'; // bir hafta önce alışveriş yapan kullanıcıların sorgusu.
  }

  // sorumluluk 3 - ekstra sorumluluk
  public function getProducts()
  {
    return '...'; // indirimdeki ürünleri getiren sorgu
  }
}
?>
```

- E-mail gönderme ile sorumlu olan sınıfımıza başka sorumluluklar da yükledik. Son bir haftadır alışveriş gerçekleştiren kullanıcıları da bu sınıf buluyor, indirimdeki ürünleri de bu sınıf buluyor, e-maili de bu sınıf gönderiyor.

- Normal şartlarda sadece e-maili göndermek ile sorumlu olan bu sınıfta sadece e-mail ile ilgili olacak değişikliklerde örneğin farklı bir e-mail servisi kullanmak istediğimizde değişiklik yapmamız gerekirdi.

- Fakat yukarıdaki şekilde oluşturulan bir sınıfta, bundan sonra “son iki hafta boyunca alışveriş yapan kullanıcılara e-mail göndermek istiyorum” dediğimiz zaman da yine bu sınıf üzerinde değişiklik yapmamız gerekecek.

- Kısacası tek sorumluluk prensibini (Single Responsibility) ihlal eden bir sınıf ile karşı karşıyayız!

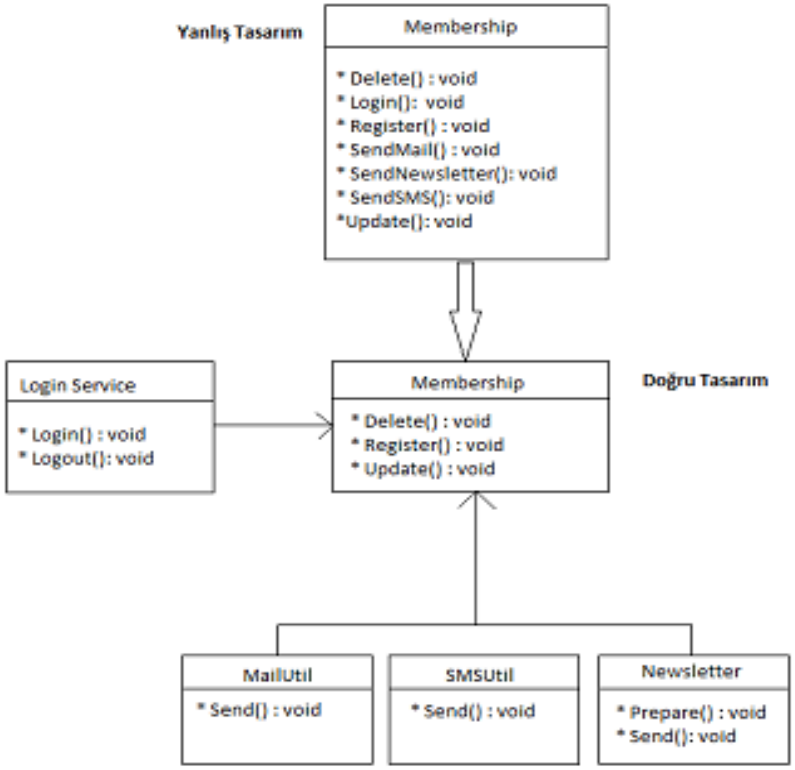
- Aynı işlemleri bir de Single Responsibility Prensibini uygulayarak yazalım:

Yeni kod yapısı incelendiğinde her sınıfın bir tek işten sorumlu olduğunu görebiliriz. Böylece daha anlaşılır ve daha kolay geliştirilebilir sınıflar, yazılımlar geliştirebiliriz.

```php
<?php

class UserRepository {
  public function getUsersByOrders($startDate, $endDate)
  {
    return '..'; // başlangıç ve bitiş tarih aralıkları arasında alışveriş yapa
  }
}

class ProductRepository {
  public function getDiscountedProducts()
  {
    return '..'; // indirimdeki ürünleri getiren method
  }
}

class EmailSender {
  public function send(array $users, array $products)
  {
    // kullanıcılara indirimde olan ürünleri liste halinde e-mail gönderdiğimiz
  }
}
?>
```

Yazılım Tasarımı ve Mimarisi

Şekildeki örnek diyagramdaki ilk tasarım Membership sınıfına gereğinden fazla sorumluluk yükleyen hatalı bir tasarımdır. Çünkü Login(), SendMail(), SendSMS(), ya da SendNewsletter() fonksiyonları kohezyonun düşmesine neden olmaktadır. Yine, Şekil 1.2'de görüleceği üzere, bu prensibin uygulandığı daha doğru bir tasarımda ise Membership sınıfının üstlenmemesi gereken görevler farklı ve doğru nesnelere dağıtılmıştır.



ŞekilTek sorumluluk prensibi açısından yanlış ve doğru tasarımlar

Örnek 2:

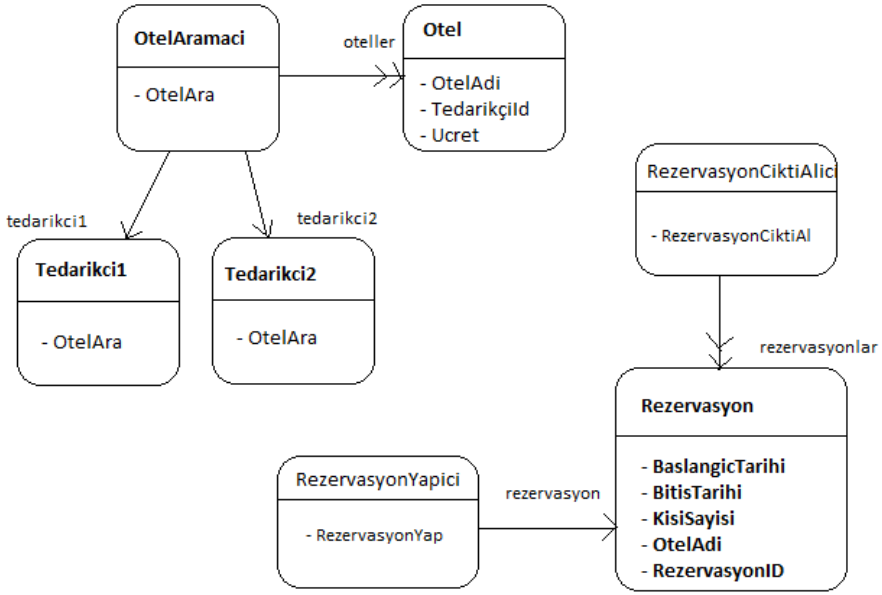
- Bir online otel rezervasyon sitesi yapılacağı düşünülsün: Kullanıcı, şehir ismi ve tarih aralığı seçecek ve ilgili şehirde, o tarihte müsait olan oteller listelenecek. Otellerin arama işlerini tedarikçi firmaların sunduğu web servisler aracılığıyla sağlanacak. Yani yazılımcının otellerle direkt bir bağlantısı yoktur. Şimdi bu tarz bir projenin kodlaması incelensin.

Yazılım Tasarımı ve Mimarisi

```
namespace OtelRezervasyonLibrary
{
    public class OnlineRezervasyon
    {
        public List<Otel> OtelAra(string sehirAdi, DataTime baslangicTarihi, DateTime bitisTarihi,int kisiSayisi)
        {
            List<Otel> sonuc= new List<Otel>();
            sonuc.AddRange(Tedarikci1OtelAra(sehirAdi,baslangicTarihi,bitisTarihi,kisiSayisi));
            sonuc.AddRange(Tedarikci2OtelAra(sehirAdi,baslangicTarihi,bitisTarihi,kisiSayisi));
            sonuc = sonuc.GroupBy(i=>i.OtelAdi).Select(i=> new Otel //En ucuzunu getir
            {
                OtelAdi = i.Key,
                Ucret = i.Min(j=>j.TedarikciID)
            }).ToList();
            return sonuc;
        }
        private List<Otel> Tedarikci1OtelAra(string sehirAdi, DataTime baslangicTarihi, DateTime bitisTarihi,int kisiSayisi)
        {
            //web servisine istek atıldı gibi düşünelim
            return new List<Otel>;
        }
        private List<Otel> Tedarikci2OtelAra(string sehirAdi, DataTime baslangicTarihi, DateTime bitisTarihi,int kisiSayisi)
        {
            //web servisine istek atıldı gibi düşünelim
            return new List<Otel>;
        }
        public bool RezervasyonYap(Otel otel, DataTime baslangicTarihi, DateTime bitisTarihi,int kisiSayisi)
        {
            return true;
        }
        public void RezervasyonYapilanOtelListesiCiktisiAl()
        {
            //burada veritabanına bağlanıp rezervasyon yapılan otellerin listesini çektiğimizi düşünüyoruz.
            //sonra excel'e çıktı alınıyor.
        }
    }
}
```

Yazılım Tasarımı ve Mimarisi

- Yukarıdakine benzer bir tasarımla bütün iş OnlineRezervasyon sınıfının üstüne yıkılmış; gerçek bir proje olsaydı bu sınıf binlerce satır olacaktı. Rezervasyonun yapılması, otellerin aranması ve çıktının alınmasında yapılacak olan değişiklikler bu sınıfın sorumluluklarıdır. Yani bu sınıfın değişmesi için üç farklı sebep var ki, bu da tek sorumluluk prensibine uymamaktadır. Şimdi kodlar tek sorumluluk prensibine göre yeniden düzenlensin ve sorumluluklar sınıflar arasında bölünsün.



- Şekildeki sınıf diyagramında da görüldüğü gibi otel arama, rezervasyon yapma ve çıktı alma sorumlulukları sınıflar arasında dağıtılmıştır. Bu sayede her sınıfın değişmesi için yalnız tek bir sebep vardır.
- Şimdi kaba kodları inceleyiniz.

Yazılım Tasarımı ve Mimarisi

```
namespace OnlineRezervasyonİyi
{
    public class Rezervasyon
    {
        public int RezervasyonID { get; set;}
        public Otel OtelAdi { get; set;}
        public DateTime BaslangicTarihi { get; set;}
        public DateTime BitisTarihi { get; set;}
        public int KisiSayisi { get; set;}
    }

}

namespace OnlineRezervasyonİyi
{
    public class Tedarikci1
    {
        public List<Otel> OtelAra(string sehirAdi, DataTime baslangicTarihi, DateTime bitisTarihi,int kisiSayisi)
        {
            //web servisine istek atıldı gibi düşünelim
            return new List<Otel>;
        }
    }
}

namespace OnlineRezervasyonİyi
{
    public class Tedarikci2
    {
        public List<Otel> OtelAra(string sehirAdi, DataTime baslangicTarihi, DateTime bitisTarihi,int kisiSayisi)
        {
            //web servisine istek atıldı gibi düşünelim
            return new List<Otel>;
        }
    }
}
```

Yazılım Tasarımı ve Mimarisi

```
namespace OnlineRezervasyonİyi
{
    public class OtelAramaci
    {
        Tedarikci1 tedarikci1 = new Tedarikci1();
        Tedarikci2 tedarikci2 = new Tedarikci2();
        List<Otel> oteller;

        public List<Otel> OtelAra(string sehirAdi, DataTime baslangicTarihi, DateTime bitisTarihi,int kisiSayisi)
        {
            oteller = new List<Otel>();
            oteller.AddRange(tedarikci1.OtelAra(sehirAdi, baslangicTarihi, bitisTarihi, kisiSayisi));
            oteller.AddRange(tedarikci2.OtelAra(sehirAdi, baslangicTarihi, bitisTarihi, kisiSayisi));
            oteller = oteller.GroupBy(i => i.OtelAdi).Select(i => new Otel //en ucuzunu getir
            {
                OtelAdi = i.Key,,
                Ucret = i.Min(j => j.Ucret),
                TedarikciID = i.Min(j => j.TedarikciID)
            }).ToList();
            return oteller;
        }
    }
}
```

Kod 1.3

- Kodların tamamını buraya koyulmamıştır. Çünkü amaç; sadece tek sorumluluk prensibine uyan ve uymayan tasarımı basit kod örnekleriyle aktarmaktı. Her sınıfın veya metodun değişmesi için tek bir sebep olmalıdır. Ancak çok küçük projelerde bu prensibi uygulamaya çalışmak, sınıf sayısını artırdığı için tercih edilmez.

OCP : Açık/Kapalı Prensibi (Open/Close Principle)

- Yazılımlar genellikle zaman içinde değişim göstermek zorunda kalır. Değişim engellenemeyecek bir durum olduğuna göre, yazılımcının yapması gereken şey; tasarladığı sistemin bir noktasında ortaya çıkması muhtemel bir değişim halinde, diğer pek çok yerde zincirleme bir değişim olmasını ve kaosu engellemektir.

- Bunun çıkar yolu; esnek bir tasarım yapmaktır.

- Bu durumu, (sistemler gelişime açık, ancak değişime kapalı olmalıdır) cümlesiyle ilk kez Bertrand Meyer formüle etmeye çalışmıştır. Buradaki sistemler ifadesinden kasıt; sınıf, modül veya kütüphane olabilir. Sistemleri mevcut kodları üzerinde değişiklikler yaparak değiştirmeye çalışmak genelde kaosa yol açar. Değişimi sadece yeni kodlar ekleyerek yapabilmek amaçlanmaktadır. Aksi takdirde bu prensip ihlal edilmiş olunur.

OCP : Açık/Kapalı Prensibi (Open/Close Principle)

- Programı geliştirmek, programa yeni bir davranış biçimi eklemek anlamına gelmektedir.

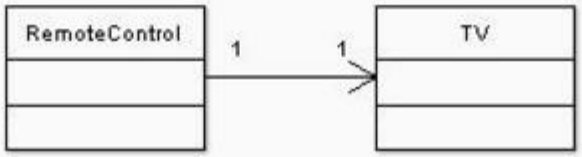
- OCP ye göre programlar geliştirmeye açık olmalıdır, yani programı oluşturan modüller yeni davranış biçimlerini sergileyecek şekilde genişletilebilmelidirler. Bir modüle yeni bir davranış biçimi kazandırılarak düşünülen değişiklik sağlanır.

- Bu yeni kod yazılarak gerçekleştirilir (bu yüzden bu işleme değiştirme değil, genişletme denir), mevcut kodu değiştirerek değil! Eğer kendinizi bir müşteri gereksinimini mevcut kod üzerinde değişiklik yaparken bulursanız, biliniz ki OCP prensibine ters düşüyorsunuz.

- Kod üzerinde yapılan değişiklik, bir sonraki gereksinimlerinde ayni şekilde implemente edilmesini zorunlu kılacaktır. Bu durum, kodun zamanla içinden çıkılmaz ve çok karmaşık bir yapıya dönüşmesine neden olur.

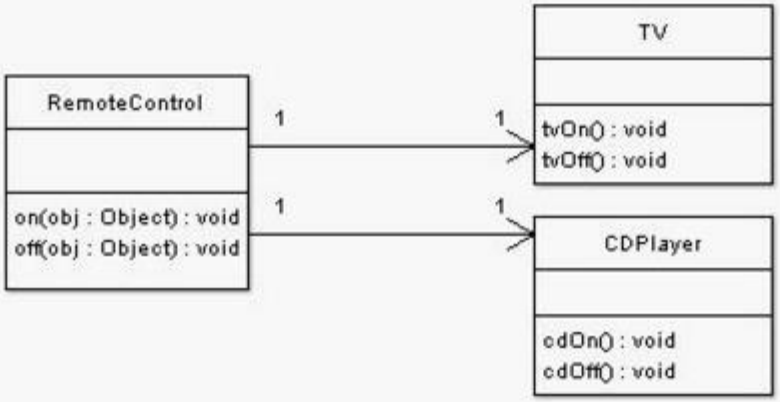
Yazılım Tasarımı ve Mimarisi

OCP prensibinin nasıl uygulanabileceğini RemoteControl – TV örneği üzerinde inceleyelim.



Resim 1 RemoteControl ve TV sınıfları

Resim 1 de görüldüğü gibi RemoteControl sınıfı TV sınıfını kullanarak işlevini yerine getirmektedir. Eğer RemoteControl sınıfını TV haricinde başka bir aleti kontrol etmek için kullanmak istersek, örneğin CDPlayer Resim 2 deki gibi değişiklik yapmamız gerekebilir. Bu noktada esnek bağ prensibini unutarak, resim 2 de yer alan çözümün bizim için yeterli olduğunu düşünelim.



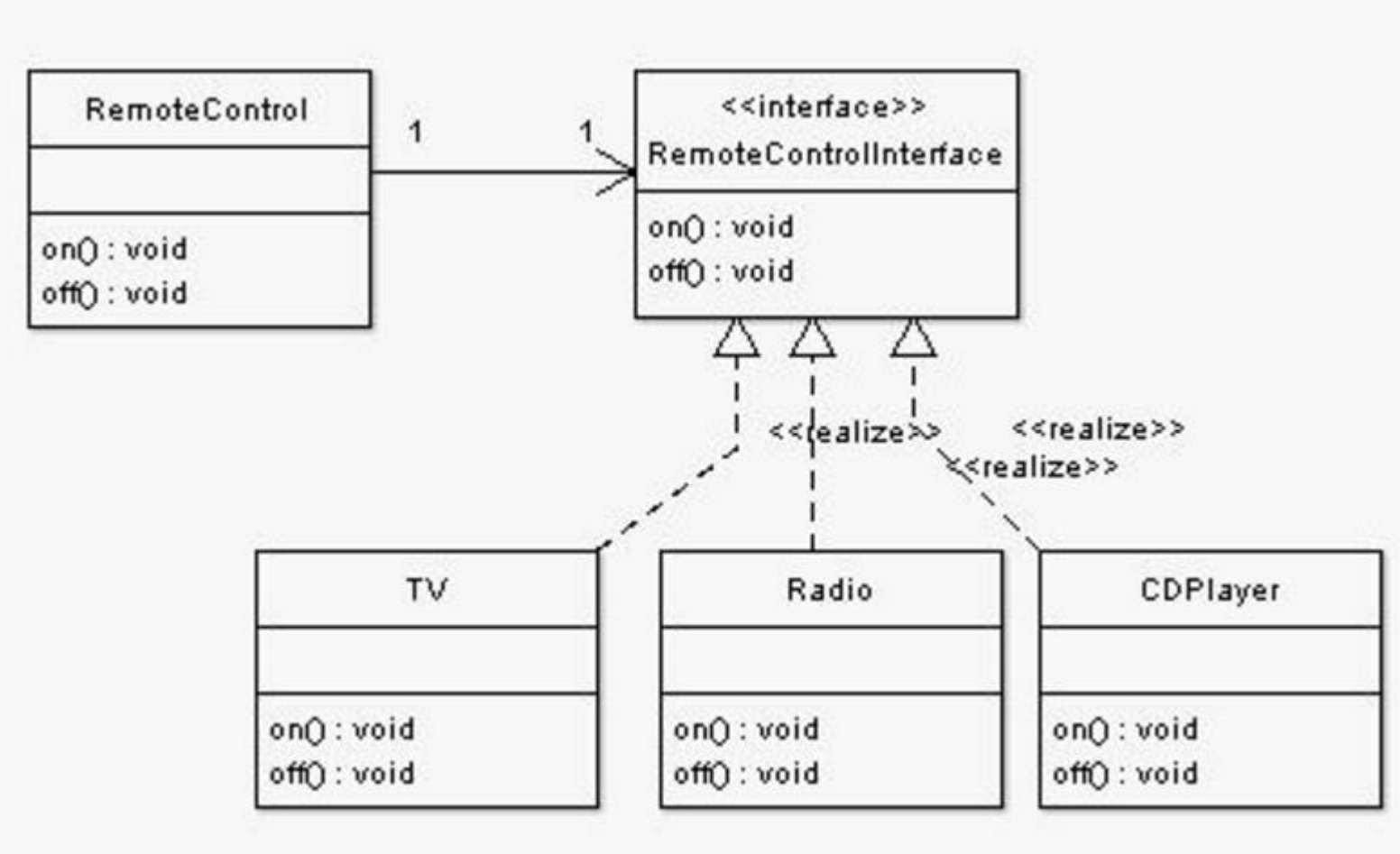
Resim 2 RemoteControl sınıfı TV ve CDPlayer sınıflarından olan nesneleri kontrol etmektedir.

- Bu şekilde oluşturulan bir tasarım OCP prensibine ters düşmektedir, çünkü her yeni eklenen cihaz için on() ve off() metotlarında değişiklik yapmamız gerekmektedir.

- OCP böyle bir modifikasyonu kabul etmez. OCP ye göre mevcut çalışır kod kesinlikle değiştirilmemelidir. Onlarca cihazın bulunduğu bir sistemde on() ve off() metotlarının ne kadar kontrol edilemez ve bakımı zor bir yapıya bürüneceği çok net olarak bu örnekte görülmektedir.

- Bunun yanı sıra RemoteControl sınıfı TV ve CDPlayer gibi sınıflara bağımlı kalacak ve başka bir alanda kullanılması mümkün olmayacaktır.
- TV ve CDPlayer sınıflar üzerinde yapılan tüm değişiklikler RemoteControl sınıfını doğrudan etkileyecek ve yapısal değişikliğe sebep olacaktır.

```
10. public class RemoteControl
11. {
12. 
13.     /**
14.      * Aleti acmak
15.      * için kullanilan metot.
16.      *
17.      */
18.     public void on(Object obj)
19.     {
20.         if(obj instanceof TV)
21.         {
22.             ((TV)obj).tvOn();
23.         }
24.         else if(obj instanceof CDPlayer)
25.         {
26.             ((CDPlayer)obj).cdOn();
27.         }
28.     }
29. 
30. 
31.     /**
32.      * Aleti kapatmak
33.      * için kullanilan metot.
34.      */
35.     public void off(Object obj)
36.     {
37.         if(obj instanceof TV)
38.         {
39.             ((TV)obj).tvOff();
40.         }
41.         else if(obj instanceof CDPlayer)
42.         {
43.             ((CDPlayer)obj).cdOff();
44.         }
45. 
46.     }
47. }
```



Şekilde yer alan çözüm OCP ye uygun yapıdadır, çünkü kod üzerinde değişiklik yapmadan programa yeni davranışlar eklemek mümkündür. OCP prensibi, esnek bağ prensibi kullanılarak uygulanabilir. OCP ye uygun RemoteControl sınıfının yapısı şu şekilde olmalıdır:

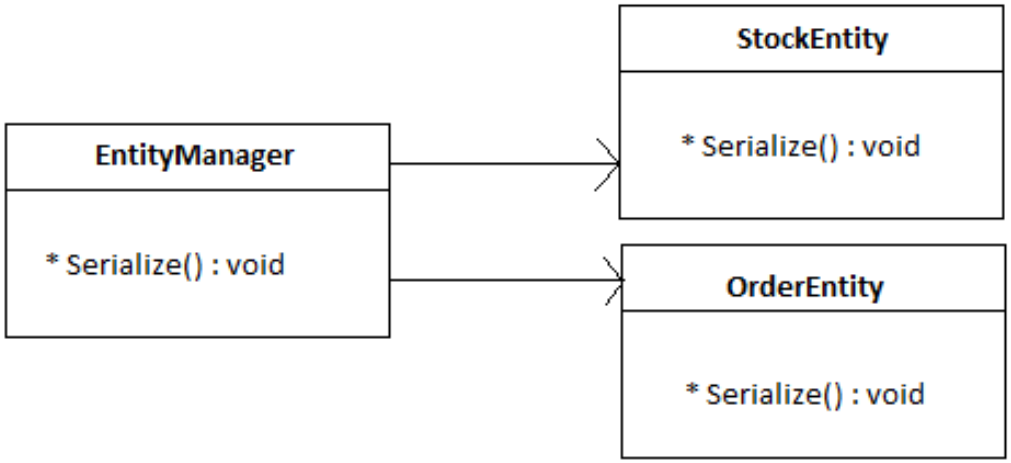
OCP ye uygun RemoteControl sınıfının yapısı şu şekilde olmalıdır:

on() ve off() metotları sadece RemoteControlInterface tipinde olan bir sınıf değişkeni üzerinde işlem yapmaktadır. Bu sayede if/else yapısı kullanmadan RemoteControlInterface sınıfını implemente etmiş herhangi bir alet üzerinde gerekli işlem yapılabilmektedir. Bu örnekte on() ve off() metotları değişikliğe kapalı ve tüm program geliştirmeye açıktır, çünkü RemoteControlInterface interface sınıfını implemente ederek sisteme yeni aletleri eklemek mümkündür. Sisteme eklediğimiz her alet için on() ve off() metotları üzerinde değişiklik yapmak zorunluluğu ortadan kalkmaktadır. Uygulanan OCP ve esnek bağ prensibi ile RemoteControl başka bir alanda her tip aleti kontrol edebilecek şekilde kullanılır hale gelmiştir.

```
public class RemoteControl
{
    /**
     * Delegasyon islemi için RemoteControlInterface
     * tipinde bir sınıf degiskeni tanimliyoruz.
     * Tüm islemler bu nesnenin metodlarina
     * delege edilir.
     */
    private RemoteControlInterface remote;

    /**
     * Sinif konstuktörü. Bir nesne oluşturma islemi
     * esnasında kullanilacak RemoteControlInterface
     * implementasyonu parametre olarak verilir.
     *
     * @param _remote RemoteControlInterface
     */
    public RemoteControl(RemoteControlInterface _remote)
    {
        this.remote = _remote;
    }

    /**
     * Aleti acmak
     * için kullanilan metot.
     *
     */
    public void on()
    {
        remote.on();
    }


    /**
     * Aleti kapatmak
     * için kullanilan metot.
     */
    public void off()
    {
        remote.off();
    }

}
```

Örnek

- Başka bir örnek ise şekil'de gösterilen EntityManager sınıfıdır. EntityManager sınıfı her sistemdeki her bir entity için (veritabanı tablolarını ifade eden varlık sınıfı anlamında) tabloya kaydetme işlevi gören ortak bir **Serialize()** fonksiyonuna sahiptir.



Şekil. Yanlış Tasarım

Yazılım Tasarımı ve Mimarisi

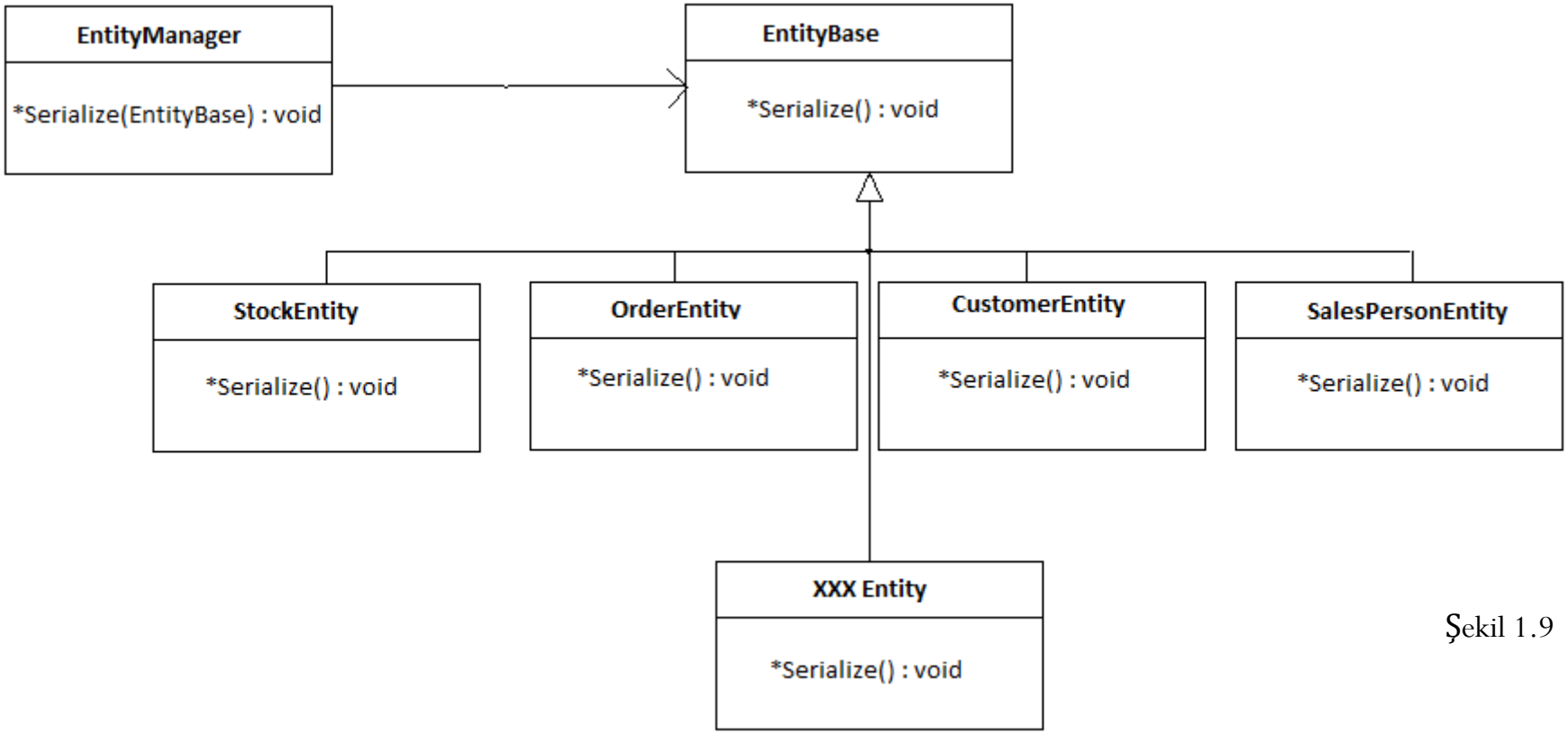
- Bu fonksiyon parametresine aldığı Object sınıfı türündeki referansın dinamik türüne bakarak ilgili entity sınıfının **Serialize()** fonksiyonunu çağırmaktadır. Örneğin,

```
[Java,C#]

class EntityManager
{
    public void Serialize(Object Obj)
    {
        //C#'ta instanceof yerine is kullanılır
        if(obj instanceof StockEntity)
        {
            ((StockEntity) obj).Serialize();
        }
        if(obj instanceof OrderEntity)
        {
            ((OrderEntity) obj).Serialize();
        }
        //Yanlış tasarım çünkü sonradan eklenecek
        //entity'ler için yeni if'ler yazılması gerekir.
    }
}
```

Kod 1.7

Yazılım Tasarımı ve Mimarisi

- Zaman içinde yeni tablolar ve dolayısıyla yeni entity sınıflarının eklenmesi gündeme geldiğinde bu kez programcı ister istemez EntityManager::Serialize() fonksiyonuna yeni if'ler eklemek zorunda kalacak yani bu fonksiyonu değiştirmek zorunda kalacaktır. Bu durum söz konusu prensibin ihlal edilmesi anlamına gelir. Oysa doğru tasarım tüm entity sınıflarının **EntityBase** gibi bir taban sınıf ya da interface'ten implemente edilmesini ve **EntityManager::Serialize()** fonksiyonunun da bu taban sınıf ya da interface türünde bir referans parametresi almasını gerektirir. Bu durum şekil 1.9'da gösterilmiştir.



Şekil 1.9

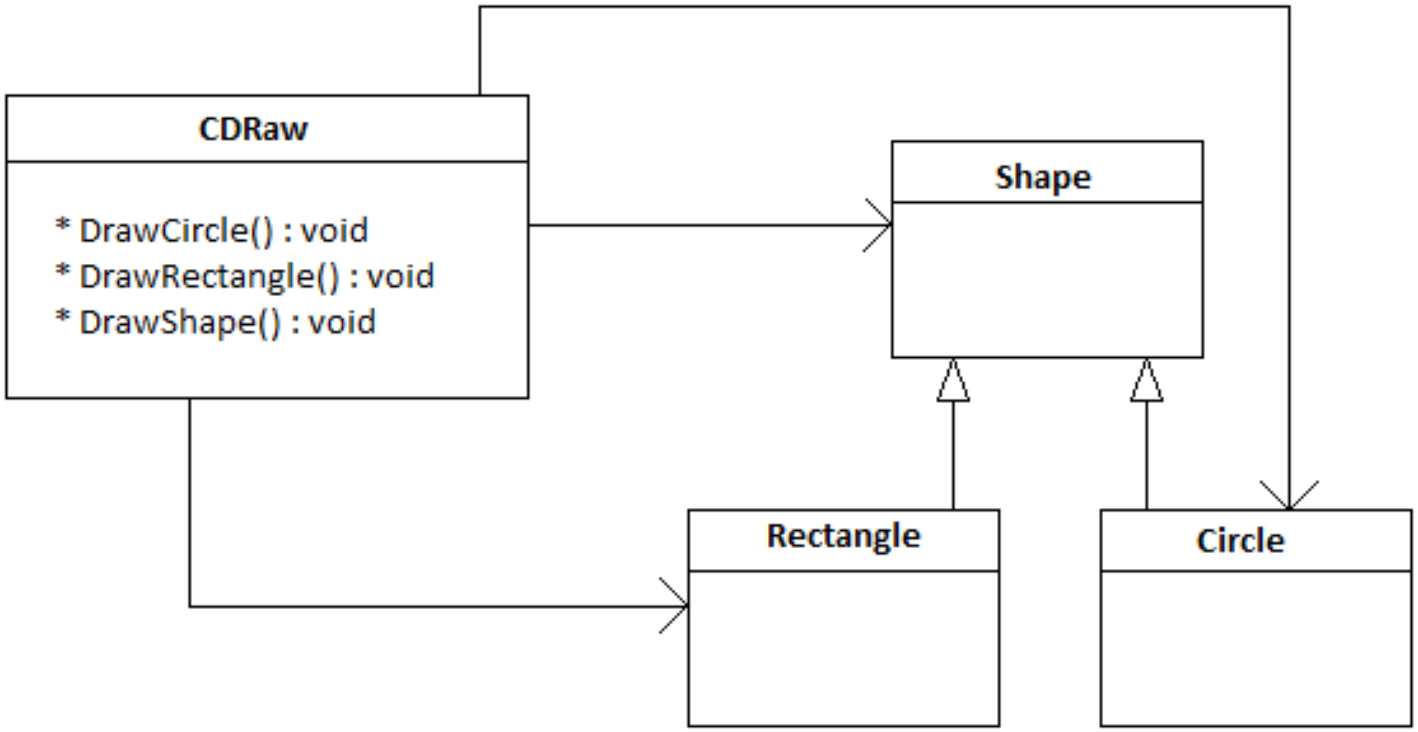
Yazılım Tasarımı ve Mimarisi

Böylece yeni entity sınıfları eklense de **Serialize()** fonksiyonunda değişiklik gerekmeyecektir.

```
[Java,C#]

class EntityManager
{
    public void Serialize(EntityBase obj)
    {
        obj.Serialize();
    }
}
```

Örnek

- Bu prensibi ihlal eden bir yanlış tasarım üzerinden hareket etmek gerekirse, **CDRaw** sınıfı, **Shape** taban sınıfından türetilmiş çeşitli sınıflarla farklı şekilleri çizebilen fonksiyonlara sahip bir sınıfı temsil ediyor olsun.

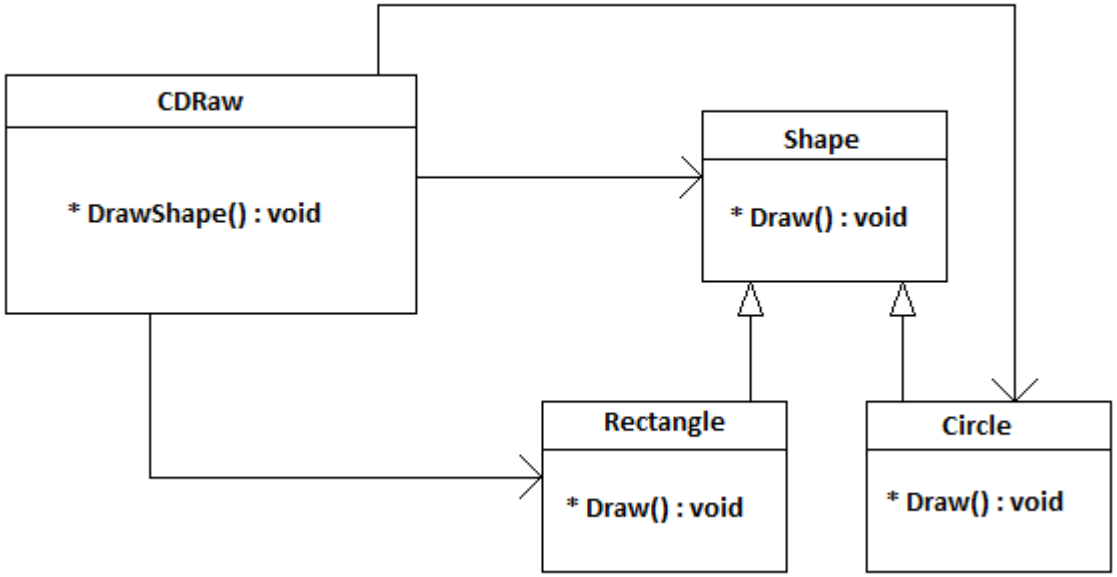


Open-Closed Prensibi açısından yanlış tasarım

Yazılım Tasarımı ve Mimarisi

- **CDRaw** sınıfına ilişkin **DrawShape()** fonksiyonunun parametre biçiminde alacağı olası bir enum değerine veya nesne türüne göre, hangi şeklin çizileceğinin belirlenmesi tipik bir kötü tasarım örneğidir. Böylesi bir kötü tasarım, yeni şekil sınıflarının kütüphaneye eklenmesiyle ciddi sorunlara neden olacaktır. **CDRaw**'a ya yeni fonksiyonlar eklemek ya da **DrawShape()** içinde değişiklikler yapılmak zorunda kalınacaktır.

- Doğru tasarım ise **Shape**'in soyut bir sınıf ya da interface olmasını gerektirir. **Draw()** da saf sanal bir fonksiyon olup, farklı türemiş Shape sınıflarında farklı biçimlerde implemente edilmeli ve **DrawShape()** ile kullanılmalıdır. Bu yapıda polimorfizmden faydalanılması gerektiği açıktır.



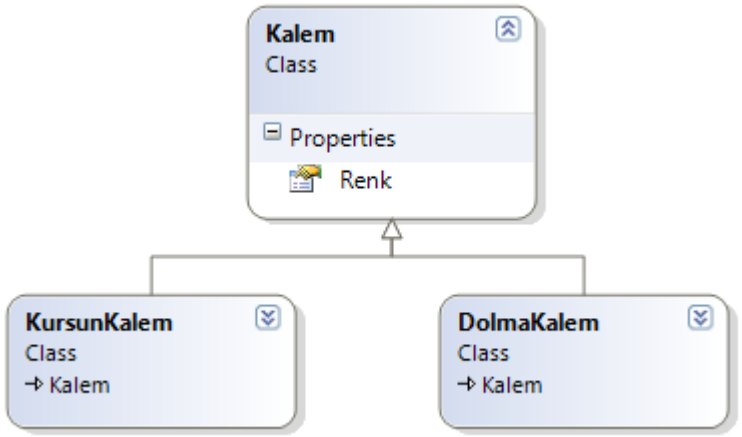
Open-Closed Prensibi açısından doğru tasarım

LSP : Liskov Yerine Geçme Prensibi (Liskov Substitution Principle)

- Türemiş sınıf nesnelerinin taban sınıf nesnesi yerine geçmesini öngörür. Daha açık bir ifadeyle taban sınıf türündeki nesne üzerinde operasyon yapacak şekilde geliştirilmiş bir fonksiyon, bu sınıftan türeyen farklı sınıflara ait nesneler üzerinde de aynı operasyonu yapabilir.

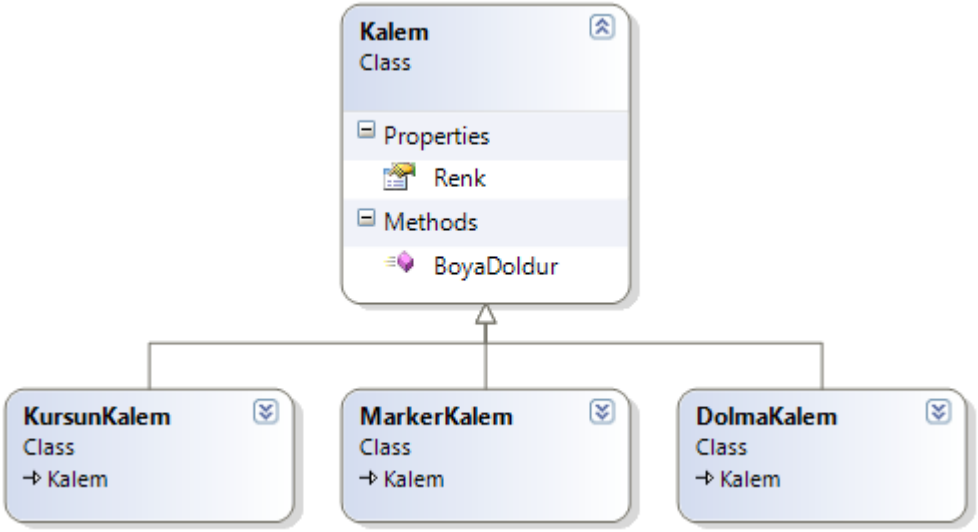
- Arabaya klima tuşu koyup onu işlevsiz bırakmak, bu prensibe tam anlamıyla terstir. Dummy code dediğimiz de tam olarak budur. Türetilen sınıfların, türeyen sınıftaki işlevselliği tam olarak yerine getirdiğinden emin olmalıyız. Yani bir program türetilen sınıfı kullanırken, bir anda türeyen sınıfları kullanmaya geçebilmeli ve sistem bundan etkilenmemelidir.

7. LSP : Liskov Yerine Geçme Prensibi (Liskov Substitution Principle)



- Kalem sınıfından türeyen iki adet sınıfımız bulunmaktadır.
- İmplementasyon aşamasında KursunKalem ve DolmaKalem nesneleri Kalem gibi davranış gösterebilir.
- Bir adet sınıf daha bu hiyerarşiye ekleyelim. Kalemimiz MarkerKalem olsun.
- MarkerKalem ve DolmaKalem'ler bittikleri zaman tekrar doldurulabilir kalemlerdir

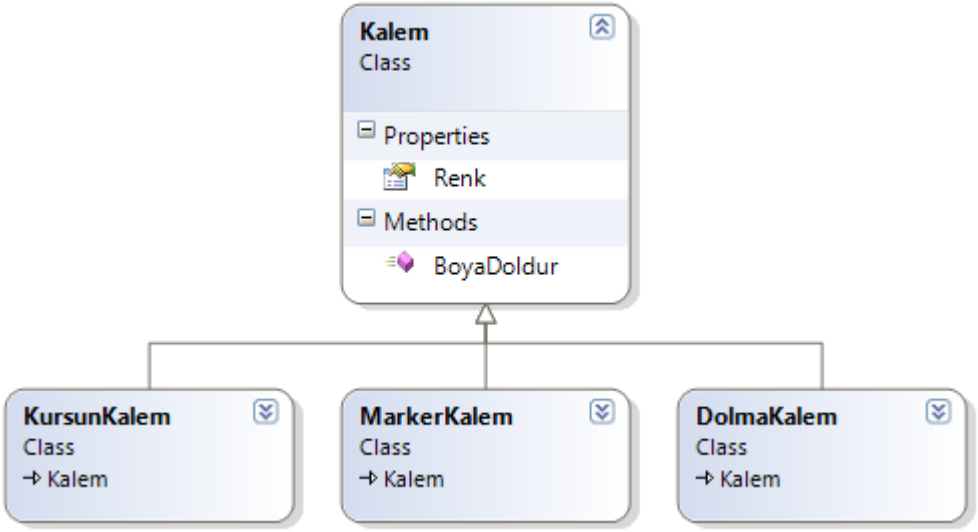
7. LSP : Liskov Yerine Geçme Prensibi (Liskov Substitution Principle)



- O zaman boya doldur işlemini nereye koymalıyız? İki alt sınıf da bu işlemi yapıyorsa bu işlemi üst sınıf olan Kalem sınıfına mı koymalıyız?

- Yukarıdaki uml aslında işimizi gayet iyi görür. BoyaDoldur methodunu sanal (virtual) yapar ve KursunKalem sınıfında da bu işlevi ezer (override) ve metodun içini boş bırakır veya istisnai durum notu düşeriz. (notimplementedexception fırlatmak). Diğer kalem türlerinde yapmak istediğimiz ne ise onu yaparız.

7. LSP : Liskov Yerine Geçme Prensibi (Liskov Substitution Principle)

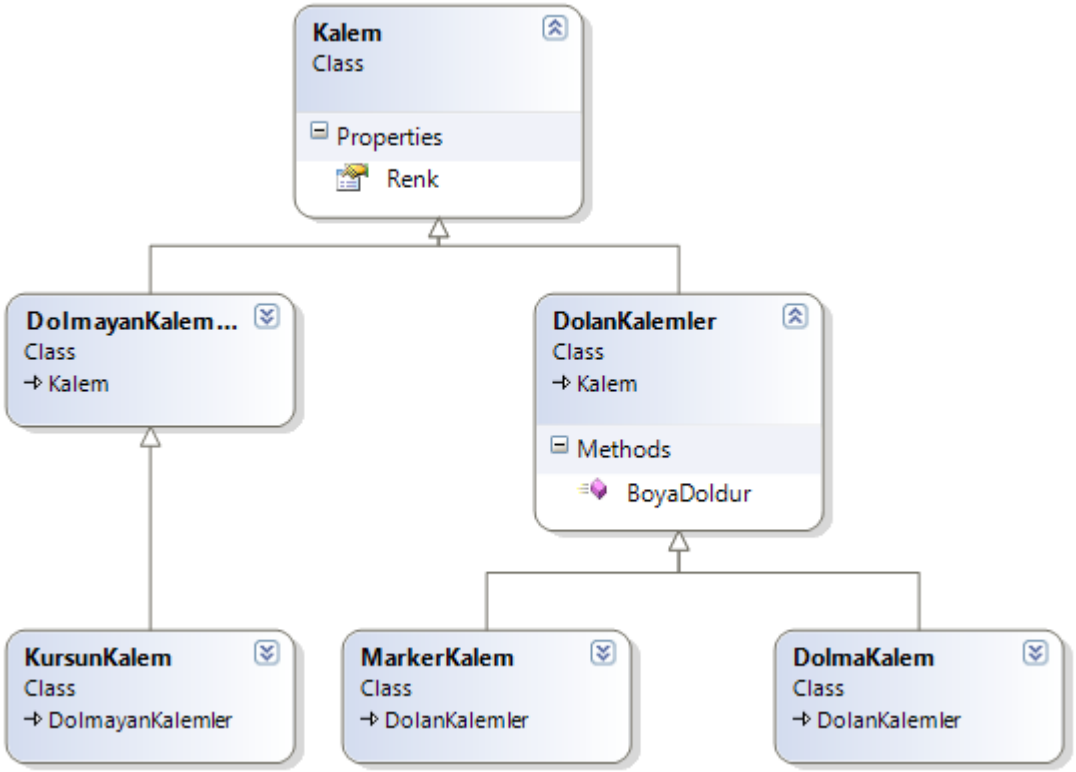


- Liskov prensibi diyor ki madem alt sınıfın üst sınıfın yerine geçemiyor. O zaman alt sınıfı ya oradan çıkar ya da olması gerektiği gibi bir ara sınıf koy.

- Alt sınıfı çıkartırız ancak bu bizim kalem özelliklerini tekrar tanımlamamıza neden olur. İkinci ifade daha mantıklı gözüküyor. Ara sınıf getirerek BoyaDoldur işlemini bu sınıfa verebiliriz.

- Şu şekilde olabilir mi?

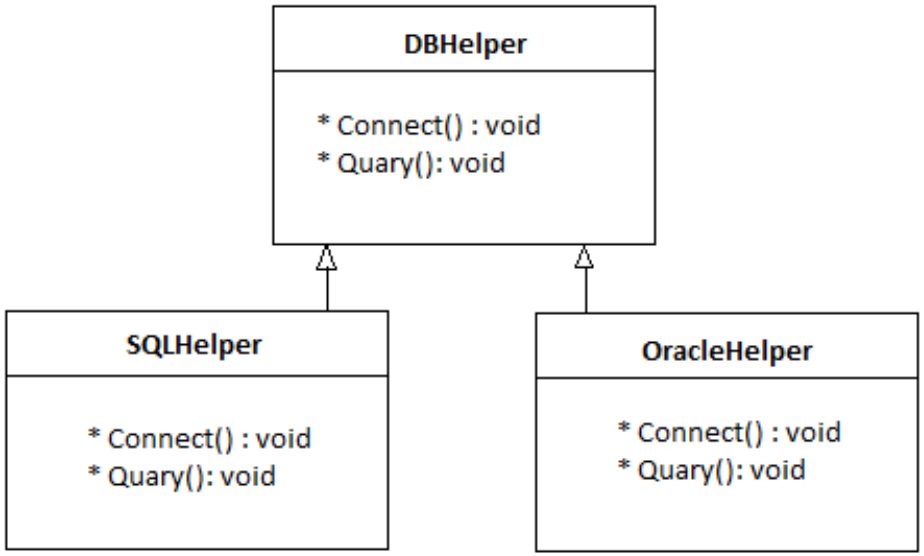
LSP : Liskov Yerine Geçme Prensibi (Liskov Substitution Principle)



- Artık KursunKalem,MarkerKalem,DolmaKalem nesneleri Kalem nesnesi gibi davranabilir. Bu çözümü ara sınıf koymak yerine bir interface tanımlayıp DolanKalemlere uygulatılabilir.

Örnek:

Aşağıdaki örnekte veri tabanından bağımsız bir veri katmanı yazılmak istenmiştir. Bu yapı yeni veri tabanları için yeni sınıfların eklenebilirliği fakat değişim gerekmemesiyle hem Açık/Kapalı prensibiyle uyumlu hem de Baglan() ve Sorgula() fonksiyonlarının DBHelper sınıfı türünde aldığı parametre (Liskov ilkesine göre) bu sınıftan türeyen iki farklı sınıf türünde referansın geçilebilmesiyle de Liskov prensibiyle uyumludur.



Yazılım Tasarımı ve Mimarisi

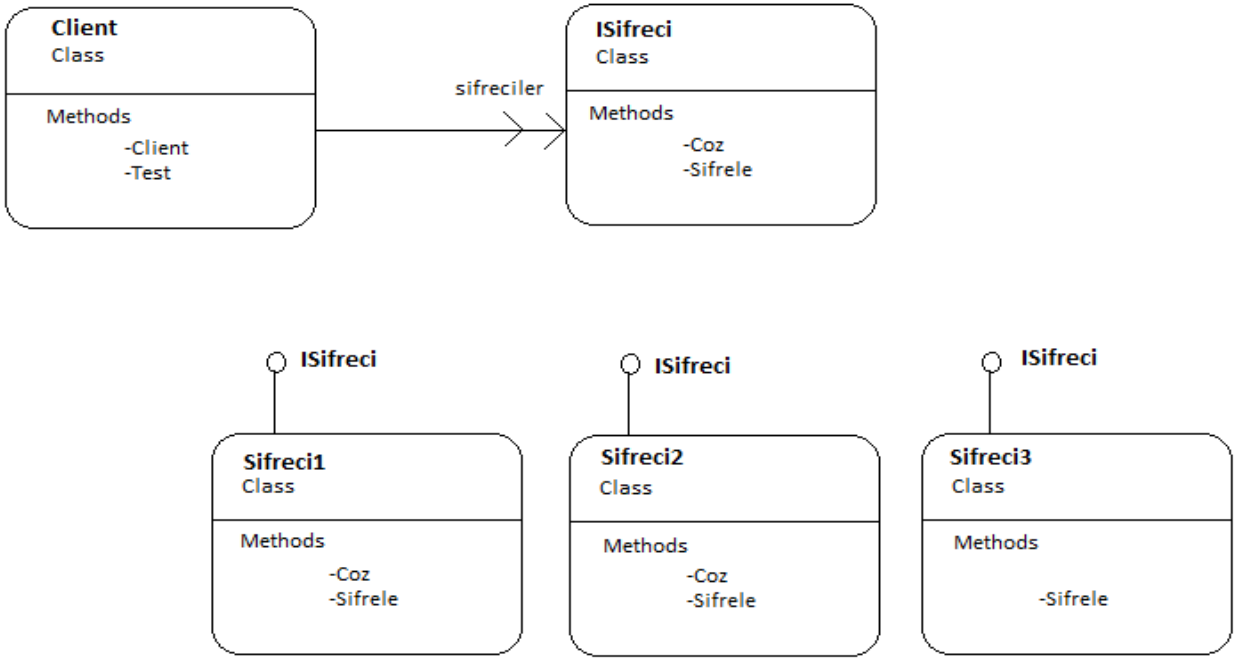
```
public abstract class DBHelper
{
    public abstract void Connect();
    public abstract void Query();
}

public class SQLHelper : DBHelper
{
    public override void Connect()
    {
        Console.WriteLine("SQL'e bağlan");
    }
    public override void Query()
    {
        Console.WriteLine("T-SQL");
    }
}

public class OracleHelper:DBHelper
{
    public override void Connect()
    {
        Console.WriteLine("Oracle'a bağlan");
    }
    public override void Query()
    {
        Console.WriteLine("PL-SQL");
    }
}
```

```
////////////////////////////////////////

class Program
{
    static void Baglan(DBHelper db)
    {
        db.Connect();
    }
    static void Sorgula(DBHelper db)
    {
        db.Query();
    }
    static void main()
    {
        SQLHelper sql = new SQLHelper();
        Baglan(sql);
        Sorgula(sql);

        OracleHelper ora = new OracleHelper();
        Baglan(ora);
        Sorgula(ora);
    }
}
```

Yazılım Tasarımı ve Mimarisi

Örnek

- E-Ticaret uygulamalarında, müşterinin kredi kartı bilgisini alıp şifreleyeceğinizi düşünün. Aynı şekilde, şifreleme işine kullanıcı kayıt işleminde şifre alınırken de ihtiyaç olacak. Bunları sağlayan birkaç şifreleme algoritması oluşturduğunuz veya hazırlarını kullanacağınızı varsayın. Sıradaki işlem bunları soyutlamak olacaktır; çünkü direkt bağımlılığın zararlarından bahsedilmişti. Bütün şifreci sınıfları entegre edildikten sonra hangisinin daha karmaşık şifrelendiğini test etmek için, aynı metin sırasıyla bütün şifreci sınıfları içinde işleme sokularak şifrelenecektir.
- Aşağıda hızlıca kurulmuş bir yapı var. Ancak oluşturulan yapıda fark ediliyor ki **Sifreci3** sınıfı sadece şifreliyor ama çözemiyor. Yani bir Hash algoritması olduğu için geri dönüş olmuyor. Çözümleme kısmında kodlarda görülüyor. Bu durumda kodda if-else blokları olmak zorundadır.



Yazılım Tasarımı ve Mimarisi

```
namespace SifrelemeKotu
{
    public İnterface ISifreci
    {
        string Sifrele(string sifrelenecekMetin);
        string Coz(string cozulecekMetin);
    }
}

namespace SifrelemeKotu
{
    //tamamen kaba koddur
    public class Sifreci1:ISifreci
    {
        public string Sifrele(string sifrelenecekMetin)
        {
            return sifrelenecekMetin + "sifre";
        }
        public string Coz(string cozulecekMetin)
        {
            return cozulecekMetin.Substring(0,cozulecekMetin.Length - 5);
        }
    }
}
```

Yazılım Tasarımı ve Mimarisi

```
namespace SifrelemeKotu
{
    public class Sifreci2:ISifreci
    {
        public string Sifrele(string sifrelenecekMetin)
        {
            return sifrelenecekMetin + "%";
        }
        public string Coz(string cozulecekMetin)
        {
            return cozulecekMetin.Split('%')[0];
        }
    }
}

namespace SifrelemeKotu
{
    public class Sifreci3:ISifreci
    {
        public string Sifrele(string sifrelenecekMetin)
        {
            return sifrelenecekMetin.Substring(0, sifrelenecekMetin.Length - 3);
        }
        public string Coz(string cozulecekMetin)
        {
            //Burada bir geri dönüş(return) yok fakat türetildiği sınıfta vardır.
            throw new NotImplementedException();
        }
    }
}
```

Kod 1.12

Yazılım Tasarımı ve Mimarisi

```
namespace SifrelemeKotu
{
    public class Client
    {
        private List<ISifreci> sifreciler = null;
        public Client(List<ISifreci> sifreciler)
        {
            this.sifreciler = sifreciler;
        }
        public void Test()
        {
            List<string> sifreliMetinler = new List<string>;
            List<string> cozulenMetinler = new List<string>;
            sifreciler.ForEach(i =>
            {
                //Eğer sifreci3 değilse şifrel. Çünkü şifreci3 bir hash algoritmasıdır
                //yani geri dönüş yoktur.
                //yanlış tasarım nedeniyle buraya bir if eklenmek zorunda kalındı.
                if(!(i is Sifreci3))
                    sifreliMetinler.Add(i.Sifrele("deneme"));
            });
            int sayi = 0;
            for(int i = 0; i < sifreciler.Count; i++)
            {
                if(!(sifreciler[i] is Sifreci3))
                {
                    cozulenMetinler.Add(sifreciler[i].Coz(sifreliMetinler[sayi]));
                    sayi++;
                }
            }
        }
    }
}
```

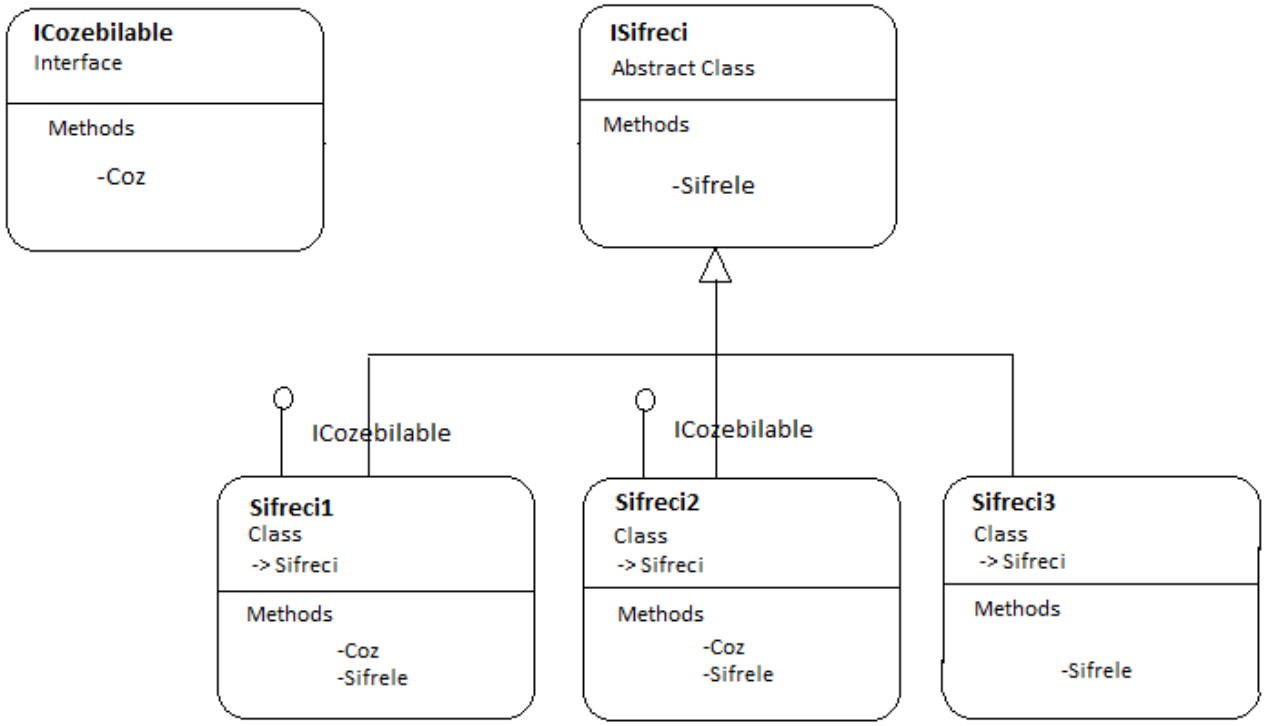
Yazılım Tasarımı ve Mimarisi

Özetle;

- Bir adet ISifreci sınıfı var ve bu sınıfla client arasında gevşek bağlılık vardır. Kendinden türeyen sınıfların Sifrele ve Coz diye iki ayrı yeteneği yerine getirmesi beklenir. Ancak bu tasarımda geri dönüşü olmayan (örneğin; MD5, SHA) bir şifreleme algoritması da ISifrele sınıfından türemiştir. Bu yüzden Coz işlevini yerine getiremez ve metodun çağrıldığı zaman, ya hata fırlatılır ya da boş değer dönülür. Aynı zamanda yukarıdaki kod *açık kapalı prensibine* de uymuyor. Yeni bir şifreci geldiğinde yeni *if-else*' ler gelebilir, yani değişime kapalı olması gereken kısımlar değişebilir. *Liskov*'a uygun hale getirmek için arayüzlere bölüp şifreleme işini çözebilen ve çözmeyen diye ayırmak gerekiyor.

Yazılım Tasarımı ve Mimarisi

Çözüm Yapısı:



Not: *Burada Abstract sınıfa sadece şifrele özelliği koyulduğu ve;*
eğer türetilen sınıfların şifre çözme özelliği varsa, Coz() fonksiyonunun Icozebilable interface' i kullanılarak eklendiği görülüyor.

Yazılım Tasarımı ve Mimarisi

- Yeni tasarımda çözme işini sağlayan sınıflar ayrı bir interface' den türetildi ve şifreci sınıfı soyut sınıf yapıldı. İçine sadece şifreleme metodu koyuldu.
- Artık şu biliniyor ki Sifreci sınıfından türeyen bütün sınıflar, kesinlikle şifreleme yeteneğini yerine getirirler.

- Bütün ***ICozebilable*** interface'inden türeyen sınıflar ise bir metni çözebilirler.
- Bu sayede *Liskov*'un prensibine uygun kod yazılmış oldu; çünkü artık alt sınıflar üst sınıfların bütün metotlarını ve propertylerini gerçekledi.
- Aynı zamanda open close prensibine de uygun bir kod; çünkü herhangi bir client sınıf yazılması durumunda artık bir if-else yapmasına gerek kalmayacak. Yeni bir sınıfın eklenmesi client'ı değiştirmeyecek.

Yazılım Tasarımı ve Mimarisi

DIP : Bağımlılığı Ters Çevirme Prensibi(Dependency Inversion Principle)

- Bu prensip, yüksek seviyeli sınıfların, aşağı seviyeli sınıflarla doğrudan bir bağımlılığının olmamasını öngörür.

- Öncelikle buradaki yüksek ve aşağı seviyeli sınıf(modül) kavramının ne anlam ifade ettiğini aydınlatalım.

- Aşağı seviyedeki modüller uygulamadaki temel işlevleri yerine getiren sınıflardır. Örneğin bir stok takip uygulamasında kota bilgilerini alan Finder sınıfı ya da sayfada bu bilgileri göstermeye yarayan Renderer sınıfı gibi sınıflar aşağı seviyeli modüllere örnektir. Öte yandan bu sınıfları kullanarak çalışan Stock sınıfı ise yüksek seviyeli bir modül örneği oluşturur. Yüksek seviyeli modüller aşağı seviyeli modülleri anlamlı kılan ve her biri belirli bir işlev yerine getirmek üzere tasarlanmış, üstelik çoğunlukla da birbirlerinden bağımsız yani otomatik modüllerdir.

Yazılım Tasarımı ve Mimarisi

Örnek:

- Aşağıdaki örnek sınıfta bu anlamda yanlış yapılmış bir tasarım örneği verilmiştir. Stock isimli sınıf, yarattığı Finder nesnesiyle veritabanından kota bilgilerini alıp, yarattığı Renderer nesnesiyle de bu bilgileri Html formatında render etmektedir.

```
[C#, Java]

public class Stock
{
    public string GetInfoAsHtml()
    {
        Finder fnd = new Finder();
        StockInfo[] stocks = fnd.FindQuoteInfo(...);

        Renderer ir = new Renderer(RenderFormat.Html);
        return ir.RenderQuoteInfo(stocks);
    }
}
```

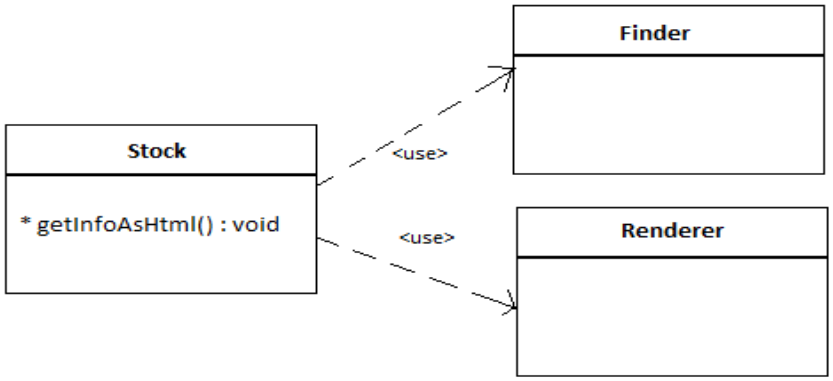
- Buradaki sorun; **Stock** sınıfının **Finder** ve **Renderer** sınıflarını doğrudan kullanmasıyla ortaya çıkan ve bağımlılığı yüksek tasarımdır. Tasarım bu haliyle stokların veri tabanında değil de farklı bir veri kaynağında bulunmak istenmesi ya da verilerin farklı formatlarda render edilebilmesinin istenmesi halinde sorunlar çıkartacaktır.

Yazılım Tasarımı ve Mimarisi

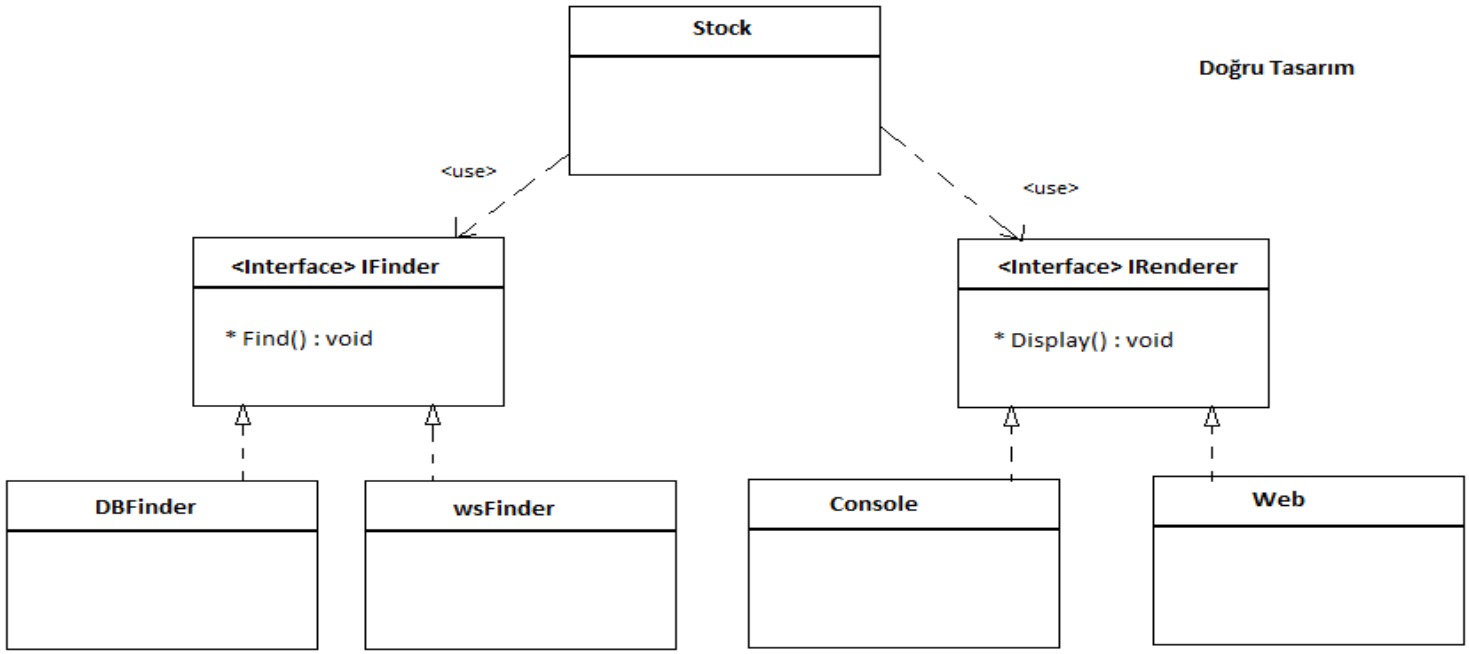
- İşte bu noktada "Dependency Inversion" prensibi kullanılarak bu bağımlılık yok edilebilir yani tersine çevrilebilir. Her şeyden önce **Finder** ve **Renderer** sınıfları doğrudan kullanılmak yerine arayüzleri üzerinden kullanılmalıdır. Bu amaçla **IFinder** ve **Irenderer** arayüzleri yazılmıştır. Söz konusu sınıflar bu arayüzleri implemente etmelidir.

```
[C#]

interface IFinder
{
    string Find(string name);
}
interface IRenderer
{
    void Display(string content);
}
class DBFinder : IFinder
{
    public string Find(string name)
    {
        return name + "isimli ürün veritabanında bulundu";
    }
}
```

Yazılım Tasarımı ve Mimarisi



Yanlış Tasarım



Doğru Tasarım

Yazılım Tasarımı ve Mimarisi

```
class wsFinder : IFInder
{
    public string Find(string name)
    {
        return name + "isimli ürün web servis aracılığıyla bulundu";
    }
}
class ConsoleRender : IRenderer
{
    public void Display(string content)
    {
        Console.WriteLine(content);
    }
}
class WebRender : IRenderer
{
    public void Display(string content)
    {
        Console.WriteLine("<b>" + content + "</b>");
    }
}
```

Kod 1.16

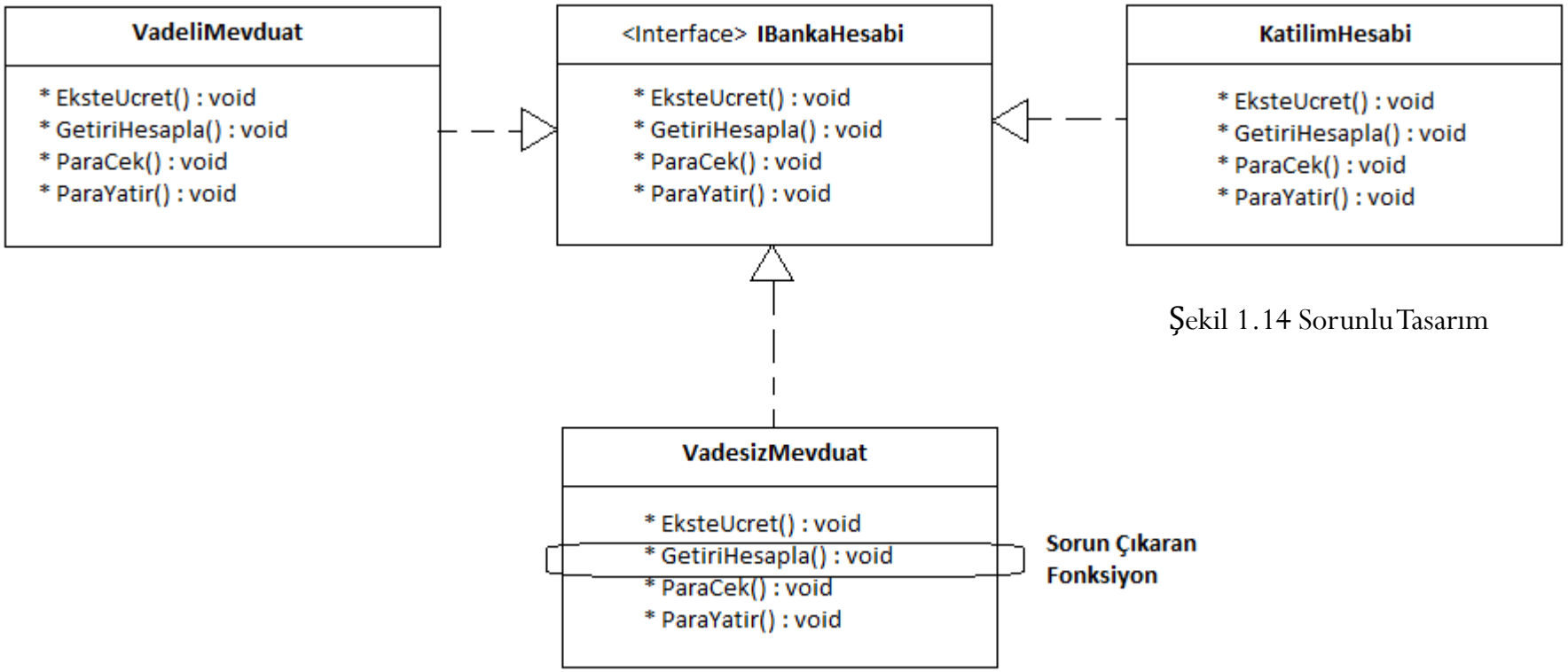
```
class Stock
{
    private IFinder fnd;
    private IRenderer rnd;

    public Stock(IFinder, f, IRenderer r)
    {
        fnd = f;
        rnd = r;
    }
    public void DisplayStockInfo(string name)
    {
        rnd.Display(fnd.Find(name));
    }
}
class Program
{
    static void Main(string[] args)
    {
        Stock stk = new Stock(new DBFinder(), new WebRender());
        stk.DisplayStockInfo("Buzdolabı");
    }
}
```

- Doğru tasarımda **Stock** sınıfı bu arayüzler türünde iki veri elemanı içermeli ve ilgili işlemleri bu nesneler üzerinden yapmalıdır ki, böylece ileride farklı arama ve render etme biçimlerinin geliştirilmesine ihtiyaç duyulursa bu sınıf kolayca yeni durumlara adapte edilebilsin. Böylece programcı farklı Renderer veya Finder türevleri geliştirirse bunu kolayca kütüphaneye ekleyebilir ve Stock sınıfı aşağı seviyeli modüller olan bu sınıfları bilmeksizin çalışabilir.

Yazılım Tasarımı ve Mimarisi

ISP: Arayüz Ayırma Prensibi(Interface Segregation Principle)

- Ortak bir arayüzü gerçekleyen (impelemente eden) sınıflarla oluşturulan yapılarda, bazı sınıfların söz konusu arayüzün zorunlu kıldığı tüm fonksiyonları içermemesi, başka bir deyişle arayüzdeki bazı fonksiyonları implemente etmemesi isteniyor olabilir.
- Detaylı açıklamak gerekirse; gerçek projelerde yazılması gereken kodların karmaşıklığı öylesine artar ki bağımlılıktan kurtulmak için arayüz (*Interface*) kullanılır. Ancak bir arayüze çok fazla yetenek yüklenirse o arayüzü gerçekleyen sınıflar için bazen sıkıntılı durumlar oluşur. Eğer arayüz fazla uzun olursa kullanılmayan işlevler olması durumu ortaya çıkar. Arabaya klima tuşu koyup işlevsiz bırakma örneğindeki gibi.
- Aşağıdaki örnek bankacılık framework yapısını inceleyiniz.



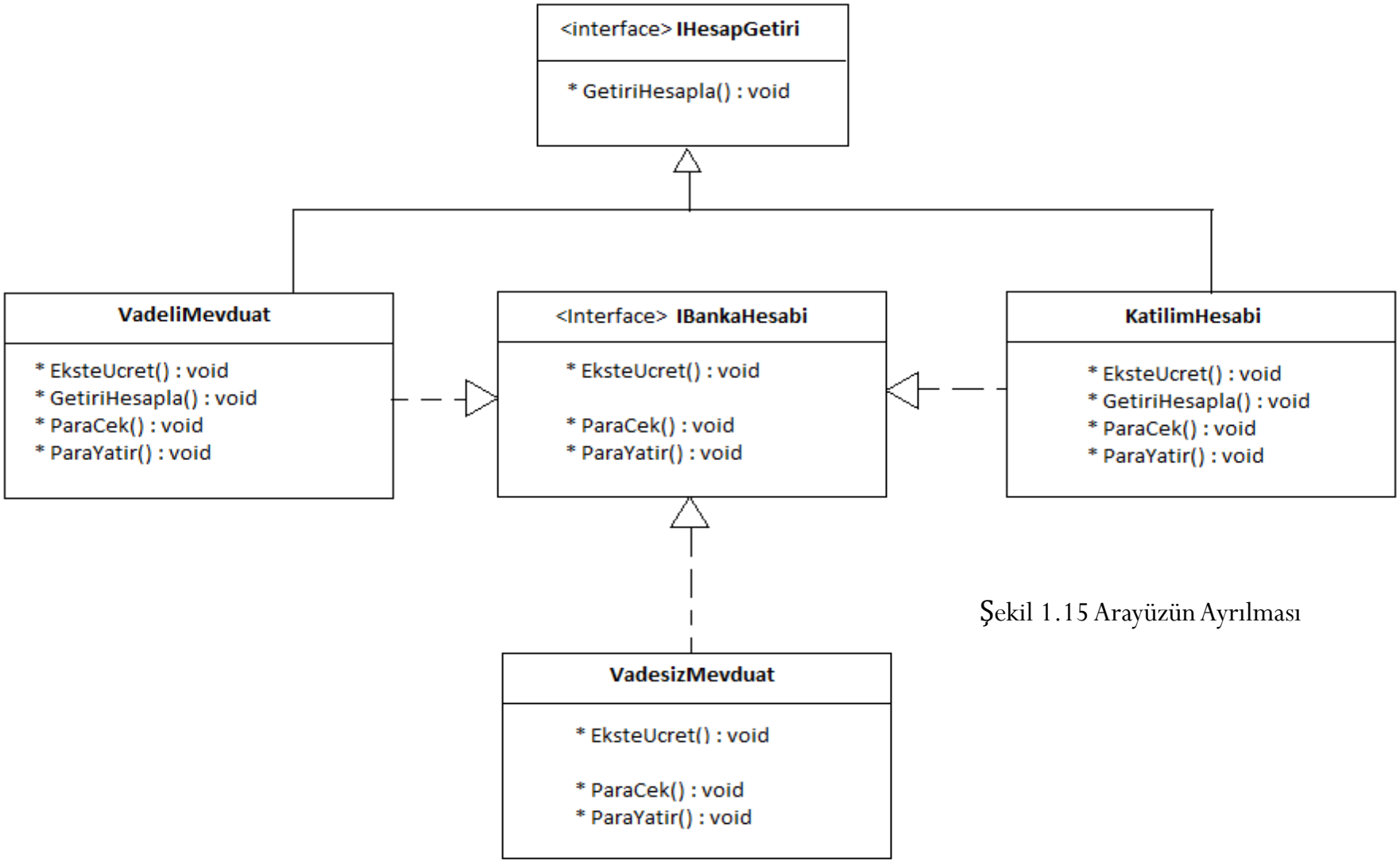
Şekil 1.14 Sorunlu Tasarım

Yazılım Tasarımı ve Mimarisi

- **Örneğin**, bir bankacılık Framework'u geliştiriliyor olsun. Tüm hesap türleri için kullanılan ortak bir arayüz söz konusu olabilir. IBankaHesabi isimli bu arayüz ParaCek(), ParaYatir() gibi fonksiyonların yanı sıraGetiriHesapla() isimli bir fonksiyon daha içeriyor olsun. Şayet bu arayüzü sadece VadeliMevduat ve KatilimHesabi sınıfları gerçekleyecek olsaydı her iki sınıfta da GetiriHesapla() faiz veya kar payı hesabı yaptırılacak şekilde yazılabilirdi. Ancak aynı arayüzü VadesizMevduat sınıfı da desteklemek sorundaysa GetiriHesapla() isimli fonksiyonun bu sınıfta da "mecburen" yer alacak olması, aşılması gereken bir sonu ortaya çıkarır.

- Bu sorunu aşmanın yolu; arayüzü ayırmaktır. Yani IbankaHesabi isimli tek bir arayüz yerine duruma göre iki veya daha çok arayüz oluşturmak ve sınıfları da bunları gerçekleyecek şekilde yazmaktır.

Yazılım Tasarımı ve Mimarisi



Şekil 1.15 Arayüzün Ayrılması

Yazılım Tasarımı ve Mimarisi

```
[C#,Java]
//Aşağıdaki örnek kodu C# programcıları "implements"
//anahtar sözcüğü yerine ":" kullanarak yazmalıdır.

public interface IHesapGetiri
{
    void GetiriHesapla();
}
public interface IBankaHesabi
{
    void EksteUret();
    void ParaCek();
    void ParaYatir();
}
public class VadesizMevduat implements IBankaHesabi
{
    public void EksteUret()
    {
    }
    public void ParaCek()
    {
    }
    public void ParaYatir()
    {
    }
}
```

```
public class KatilimHesabi implements IBankaHesabi, IHesapGetiri
{
    public void GetiriHesapla()
    {
        //Kar payı hesaplanacak
    }
    public void EksteUret()
    {
    }
    public void ParaCek()
    {
    }
    public void ParaYatir()
    {
    }
}
```

Yazılım Tasarımı ve Mimarisi

Örnek Problem:

Örnek amaçlı bir Türkçe-İngilizce sözlük programı tasarlayınız.

- Program kullanıcıdan çevrilmek istenen kelimeyi alır, veritabanında gerekli aramayı yapıp çevirisini kullanıcıya döndürür.
- Eğer birebir karşılık gelen kelime bulunamıyorsa ilgili kelimeyi içeren sözcükler (varsa) döndürülür.

    - *Örneğin prens kelimesi aranıyor olsun. İngilizce kelime veritabanında prince sözcüğü olmasın, princess sözcüğü olsun. Prens kelimesi aratıldığında prince sözcüğü olmadığı için geriye princess sözcüğü döndürülür.*

Böyle basit bir sistem için gerekli kodları ve UML diyagramlarını hazırlayınız.
Hazırlamış olduğunuz kodlarda nesneye dayalı tasarım prensibine uymayan kısımları belirtip; düzeltilmiş hallerini gösteriniz.

Notlar:

- Uygulama için 15-20 kelimelik bir veritabanı oluşturulması gerekecektir. Server olarak MS Sql ya da not defteri kullanılabilir.
- Birden fazla sınıf kullanılacaktır.
- Veritabanı işlemleri ayrı bir sınıfta gerçeklenmelidir.
- Kontrol amaçlı olarak veritabanına prince sözcüğünü eklenmesin. Princess sözcüğü eklensin.

Yazılım Tasarımı ve Mimarisi

KAYNAKLAR

1. C# ile Tasarım Desenleri ve Mimarileri, Pusula Yayıncılık, Ocak 2015, Ali Kaya ve Engin Bulut.
2. Yazılım Mimarının El Kitabı, C++ Java ve C# ile Uml ve Dizayn Paternleri, Pusula Yayıncılık, Eylül 2014, Aykut Taşdelen