NESNE YÖNELİMLİ PROGRAMLAMA 2(Object Oriented Programming 2/OOP)

Öğr. Gör. Celil ÖZTÜRK
Marmara Üniversitesi
Teknik Bilimler Meslek Yüksekokulu

İçerik

- ✓Tasarım Prensipleri
- ✓SOLID

Tasarım Prensipleri

1.Ayrıştırma (Decomposition)
2. Kohezyon (Cohesion)
3. Tek Sorumluluk Prensibi (Single Responsibility)
4. Zayıf Bağlaşım Prensibi (Low Coupling)
5. Yeniden Kullanılabilirlik prensibi (Reusability)
6. Açık – Kapalı Prensibi (OCP)
7. Liskov Yerine Geçme Prensibi (LSP)
8. Bağımlılığı Ters Çevirme Prensibi (DIP)

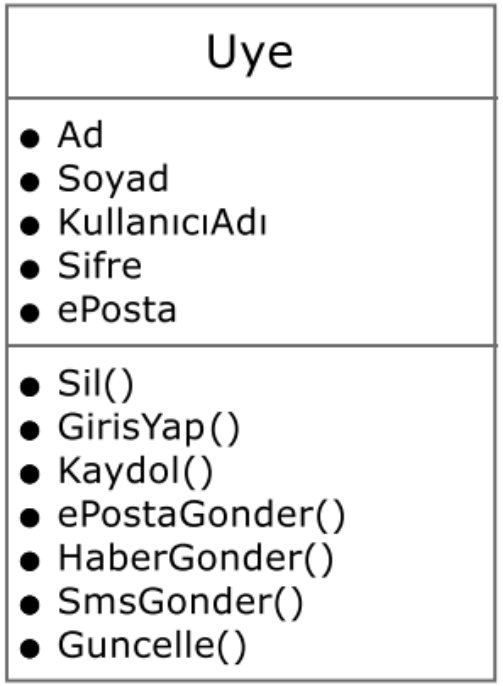
Tek Sorumluluk Prensibi (SRP)

- Tek Sorumluluk Prensibi (Single Responsibility Principle) Kohezyon olgusuyla yakından ilişkili olup, bir modülün (örneğin sınıfın) sadece tek bir sorumluluğu yerine getirmek üzere tasarlanmasını öngörmektedir.
- Kohezyon kavramının özgün bir tanımıdır.
- Zira gereksiz sorumluluklardan arındırılarak sadece belirli bir görevi yerine getirmek üzere tasarlanmış bir sınıf elde etmenin yolu; sınıfın yazılma amacını çarpıtacak yani kohezyonunu düşürecek fonksiyonlara yer vermemektir.

Tek Sorumluluk Prensibi (SRP)

- Bu prensibin uygulanması teoride kolay ancak pratikte zor olmaktadır.
- Sorumluluk ya da görevleri ayrıştırabilmek zordur.
- Doğru bir tasarımda ileride olabilecek bir değişiklikte sadece değişen durumun sorumluluğunu üstlenmiş sınıf değiştirilebilir olması öngörülür.
- "Bir sınıf sadece tek bir sorumluluğu yerine getirmelidir ve yerine getirdiği sorumluluğu iyi yapmalıdır." prensibidir.
- Her sınıf sadece kendisi ile ilgili tek bir sorumluluğu yerine getirir.
- Her sınıfın sorumluluğu farklı olduğu zaman, değişmesi için tek bir sebep olur,o da ihtiyaçların değişmesidir.
- Sınıfların birden fazla sorumluluğunun olması bağımlılığın artmasına neden olur.

Tek Sorumluluk Prensibi (SRP)/Örnek 1

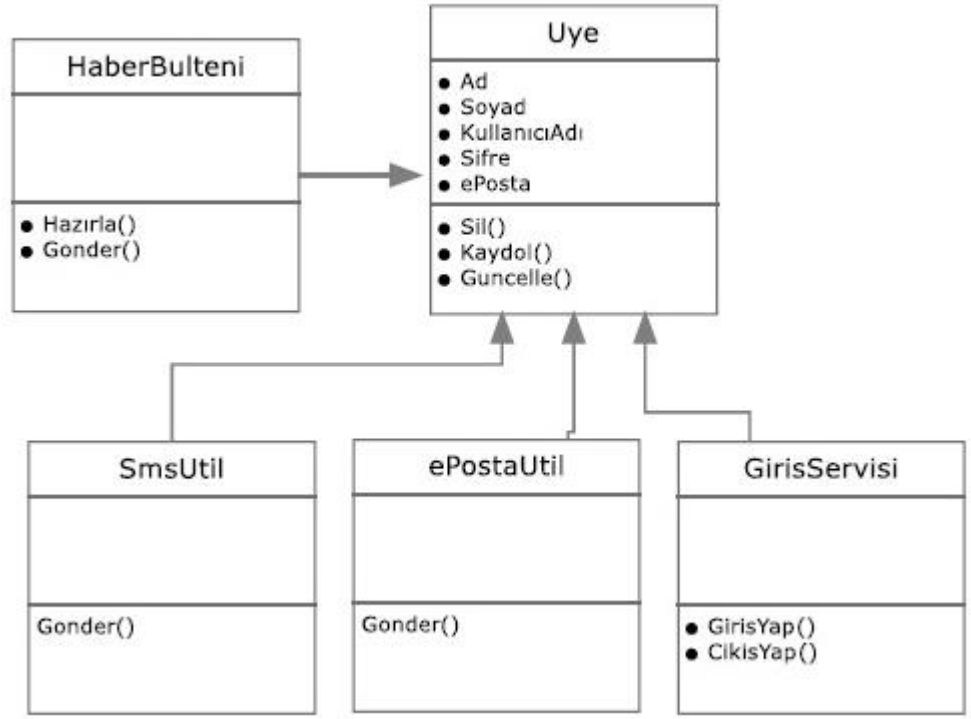


Üye sınıfına gereğinden fazla sorumluluk yüklenmiştir.

- GirisYap(),
- EPostaGonder(),
- SmsGonder(),
- HaberGonder()

fonksiyonları kohezyonun düşmesine neden olmaktadır.

Doğru Tasarım



Tek Sorumluluk Prensibi (SRP) /Örnek 1

```
public class Uye
{
    0 references
    public string Ad { get; set; }
    0 references
    public string Sifre { get; set; }
    1 reference
    public string KullaniciAdi { get; set; }
    1 reference
    public Uye(string kullaniciAdi)
    {
        this.KullaniciAdi = kullaniciAdi;
    }
}
```

```
public static class GirisServisi
{
    //Static bir DBEntityContainer'a ihtiyaç var
    1 reference
    public static bool KullaniciDogrula(string kullaniciAdi, string sifre)
    {
        return true;
    }
    0 references
    public static Uye GirisYap(string kullaniciAdi, string sifre)
    {
        //Kullanici doğrulamaya çalış
        if (KullaniciDogrula(kullaniciAdi, sifre))
            return new Uye(kullaniciAdi);
        else
            return null;
    }
}
```

```
public static class SMSUtil
{
    //SmsObject'e ihtiyaç var
    0 references
    public static void Gonder(Uye uye, string mesaj)
    {
        //Tek kişiye SMS Gönder
    }

    0 references
    public static void TopluGonder(List<Uye> uyeler, string mesaj)
    {
        //Toplı SMS Gönder
    }
}
```

Tek Sorumluluk Prensibi (SRP) /Örnek 2

```
public class User {
   private Long id;
   private String name;
   private String street;
   private String city;
   private String username;

   //Getters, setters

   public void changeAddress(String street,String city) {
    //logic
   }

   public void login(String username) {
    //logic
   }

   public void logout(String username) {
   //logic
   }
}
```

Tek Sorumluluk Prensibi (SRP) /Örnek 2

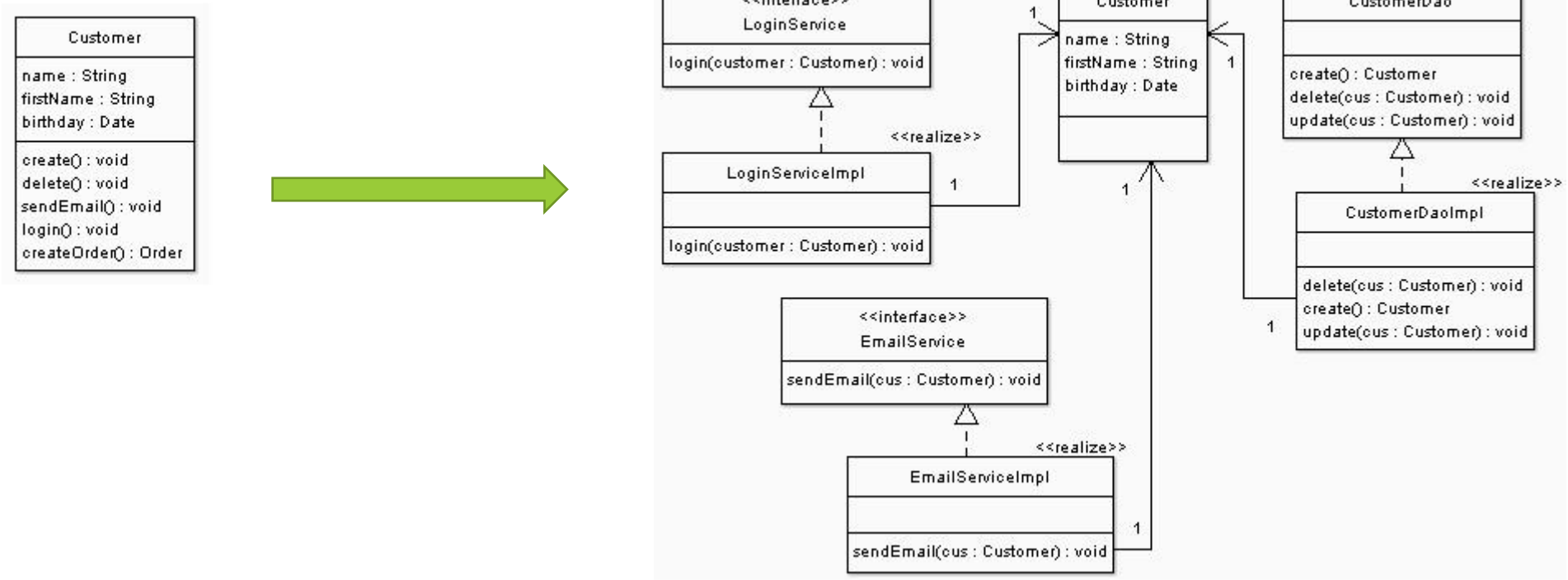
```
public class LoginService{
   public void login(String username) {
       //log-in logic
   }

   public void logout(String username) {
       //log-out logic
   }
}
```

```
public class User {

   private Long id;
   private Address name;

   //Getter,setter
}
```

```
public class AddressService{
  public void changeAddress(Address address) {
      // Sadece addressle ilgileniyorum ve ondan sorumluyum account
daki değişiklikler beni etkilemez.
       //logic
   }
}
```

```
public class Address {

   private String street;
   private String city;
   private String country;
   //Getter,setter
}
```

Tek Sorumluluk Prensibi (SRP) /Örnek 3

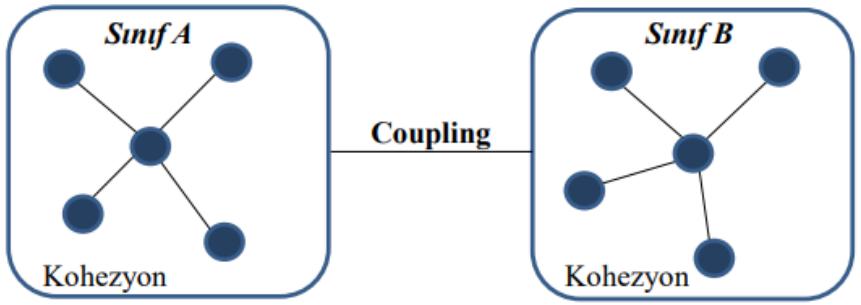


Single Responsibility Principle (SRP) – Tek Sorumluk Tasarım Prensibi – KurumsalJava.com – Özcan Acar

Zayıf Bağlaşım Prensibi (LCP)

- Gerçek dünyada nesneler nadiren tek başlarına bulunmaktadırlar.
- Çoğu zaman birbirleriyle ilişki ve iletişim halinde olan nesneler birbirlerini içerebilir ya da çeşitli biçimlerde kullanabilirler.
- Coupling, nesneler arasındaki ilişkilerin nesneleri birbirlerine ne kadar bağlı kıldığının ölçütüdür.

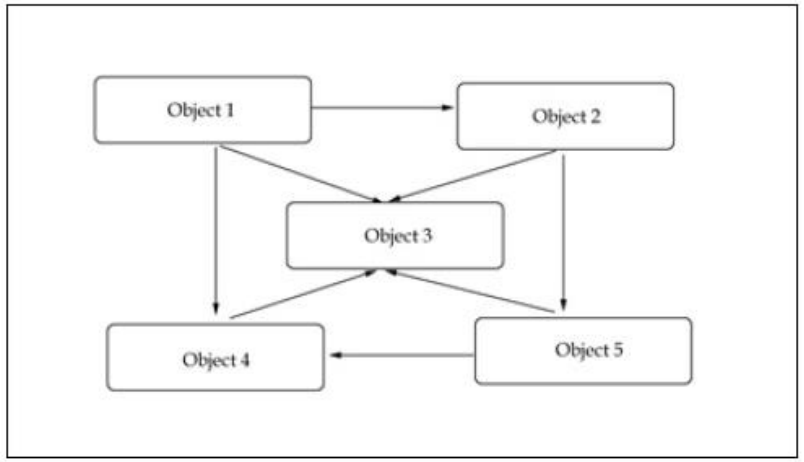
Zayıf Bağlaşım Prensibi (LCP)



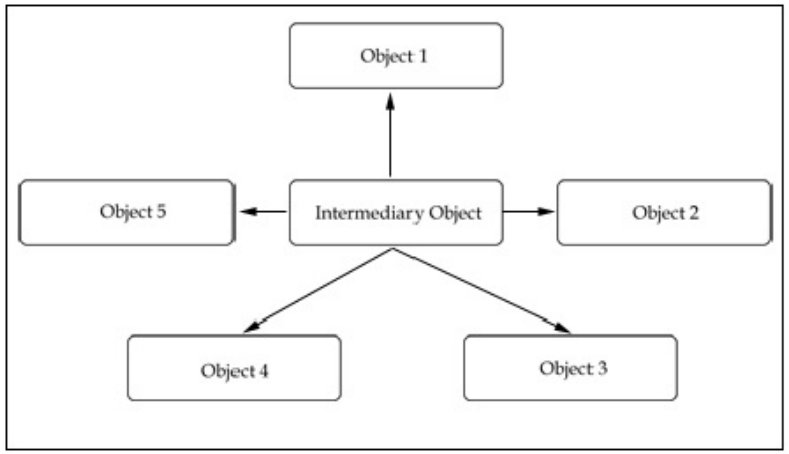
Kohezyon faktörü; sınıf içerisindeki dolayısıyla nesneye ilişkin üyelerin birbirleriyle mantıksal ilişkisi olarak tanımlanmıştı.

Kendi içinde kohezyonu yüksek sınıfların birbirleriyle ilişkiler kurarken, bu ilişkide coupling faktörü olabildiğince düşük tutulmalıdır.

İstenmeyen Yapı



İstenen Yapı



Zayıf Bağlaşım Prensibi (LCP/Low Coupling…)

- Coupling faktörünün 5 seviyesi vardır.

1. Nil Coupling
2. Export Coupling
3. Overt Coupling
4. Covert Coupling
5. Surreptitious (Gizlice) Coupling

Zayıf Bağlaşım Prensibi (LCP)

Nil Coupling :

- Teorik olarak en düşük dolayısıyla en iyi coupling düzeyidir.
- Zira bu seviyede bağımlılık söz konusu olmamaktadır.
- Diğer sınıflarla hiçbir ilgisi olmayan, tek başlarına kullanılan sınıflar bu duruma örnek teşkil etmektedir.

Zayıf Bağlaşım Prensibi (LCP)

Export Coupling:

- Herhangi bir sınıf başka bir sınıfa ortak bir arayüzle bağlıysa aralarında export coupling oluşur.
- Birçok durumda ulaşılmaya çalışılan ideal seviye export coupling seviyesidir.
- Bu seviyeden sonraki seviyeler yaratılmak istenen low coupling ilkesine zarar vermeye başlar.

Zayıf Bağlaşım Prensibi (LCP)

Overt Coupling :

- Bir sınıf, başka bir sınıfa ilişkin üyeleri belli bir izin dahilinde kullanıyorsa aralarında overt coupling söz konusudur.

Covert Coupling :

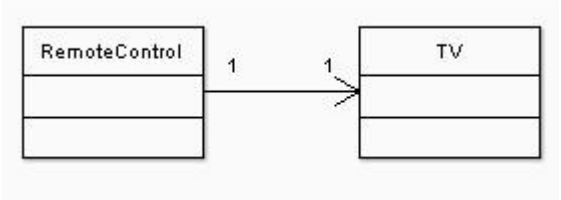
- Bir sınıf, başka bir sınıfa herhangi bir izin vermeden arkadaşlık kurması durumudur

Surreptitious (Gizlice) Coupling :

- Bir sınıf, başka bir sınıfın içsel detaylarının tümünü biliyorsa ve bunları kullanarak işlem gerçekleştiriyorsa bu sınıfların arasında surreptitious coupling oluşur.
- Tasarım açısından tehlikelidir.
- Bağımlılık, prensip gereğince az olması gerekirken, bu seviyedeki coupling'de çok fazladır..

Zayıf Bağlaşım Prensibi (LCP)

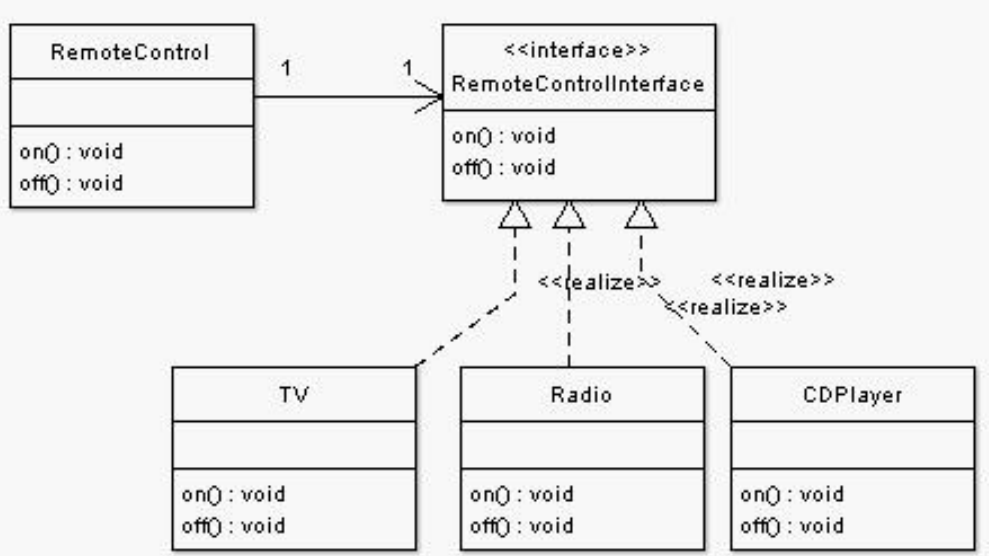
UML'de gösterildiği gibi bir sınıfın içerisinde başka bir sınıftan nesne yaratmak ve o nesnelerle bir çok işlem yaptırmak coupling'i oldukça arttırmaktadır.



RemoteControl sınıfı bünyesinde TV tipinde bir sınıf değişkeni (tv) barındırdığı için kendisini TV sınıfına bağımlı kılar.

```
public class RemoteControl
{
    /**
     * Kontrol edilen televizyon
     */
    private TV tv = new TV();

    /**
     * Televizyonu acmak
     * için kullanilan metot.
     *
     */
    public void tvOn()
    {
        tv.on();
    }


    /**
     * Televizyonu kapatmak
     * için kullanilan metot.
     */
    public void tvOff()
    {
        tv.off();
    }
}
```

Zayıf Bağlaşım Prensibi (LCP)



Resim 2 Esnek bağ için gerekli sınıf yapısı

Loose Coupling (LC) – Esnek Bağ Tasarım Prensibi – KurumsalJava.com – Özcan Acar

Zayıf Bağlaşım Prensibi (LCP)

```java
public interface RemoteControlInterface
{
    /**
     * Bu sınıfı implement eden
     * bir aleti acmak için
     * kullanilan metot.
     *
     */
    void on();


    /**
     * Bu sınıfı implement eden
     * bir aleti kapatmak için
     * kullanilan metot.
     *
     */
    void off();

}
```

```java
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();
    }

}
```

Loose Coupling (LC) – Esnek Bağ Tasarım Prensibi – KurumsalJava.com – Özcan Acar

Zayıf Bağlaşım Prensibi (LCP)

```java
public class TV implements RemoteControlInterface
{

    /**
     * Televizyonu acmak için
     * kullanilan metot.
     *
     */
    public void on()
    {
        System.out.println("TV acildi.");
    }

    /**
     * Televizyonu kapatmak için
     * kullanilan metot.
     *
     */
    public void off()
    {
        System.out.println("TV kapandi");
    }
}
```

```java
public class Test
{

    public static void main(String[] args)
    {
        RemoteControlInterface rci = new TV();
        RemoteControl control = new RemoteControl(rci);
        control.on();
        control.off();
    }
}
```

TV acildi.

TV kapandi

Loose Coupling (LC) – Esnek Bağ Tasarım Prensibi – KurumsalJava.com – Özcan Acar

Yeniden Kullanılabilirlik prensibi (Reusability)

- Yeniden kullanılabilirlik nesne yönelimli programlamanın en önemli özelliklerinden biridir ve çoğu zaman, yanlış bir yaklaşımla sadece kalıtımdan ibaret olduğu sanılır.
- Salt kalıtım yerine, kalıtım ile birlikte nesneler arasında bir ilişki kurulmalı
  - Örneğin; Kompozisyon(Composition) ve Agregasyon(Aggregation) kurgulanmalıdır.

Aggregation vs. Composition

- Aggregation: Sahip olunan nesnenin, sahip olan nesneden bağımsız bir şekilde var olabilmesine denir.
- Bu ilişkide nesnenin yok olması diğer nesneyi etkilemez.
- Örnek, dersin bitmesi ile ders nesnesinin yok olması öğrenci nesnesini etkilemez.

Aggregation vs. Composition

- Aggregation:

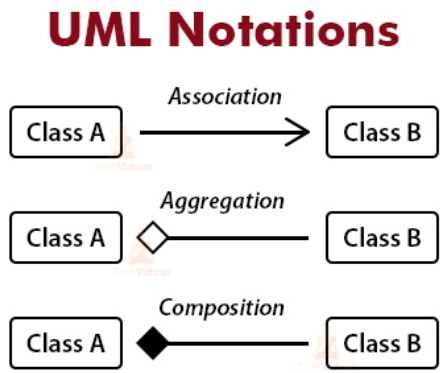
```java
// Java program to illustrate
//the concept of Aggregation.
import java.io.*;
import java.util.*;

// student class
class Student
{
    String name;
    int id ;
    String dept;

    Student(String name, int id, String dept)
    {

        this.name = name;
        this.id = id;
        this.dept = dept;

    }
}
```

```java
class Department
{

    String name;
    private List<Student> students;
    Department(String name, List<Student> students)
    {

        this.name = name;
        this.students = students;

    }

    public List<Student> getStudents()
    {
        return students;
    }
}
```

```java
class Institute
{

    String instituteName;
    private List<Department> departments;

    Institute(String instituteName, List<Department> departments)
    {
        this.instituteName = instituteName;
        this.departments = departments;
    }

    // count total students of all departments
    // in a given institute
    public int getTotalStudentsInInstitute()
    {
        int noOfStudents = 0;
        List<Student> students;
        for(Department dept : departments)
        {
            students = dept.getStudents();
            for(Student s : students)
            {
                noOfStudents++;
            }
        }
        return noOfStudents;
    }

}
```

Aggregation vs. Composition

- Composition: Nesnelerim yaratımları ve var olmaları birbirleri ile bağlantılıdır.
- Bir nesne diğerinde bağımsız olarak kullanılamaz.



Java Association - Aggregation and Composition in Java - TechVidvan

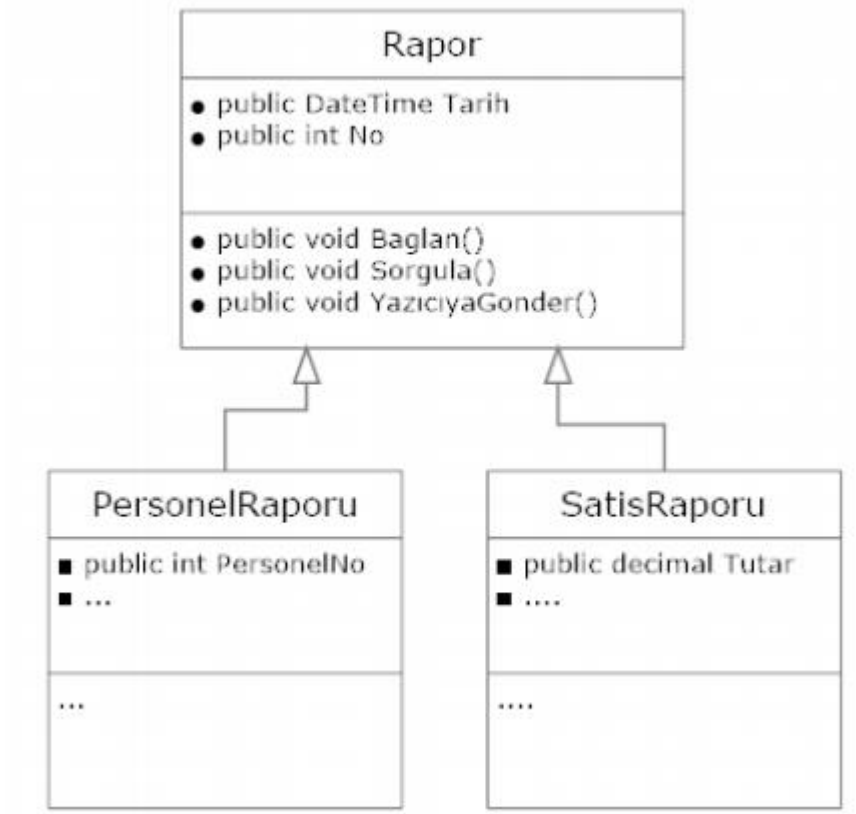
Aggregation vs. Composition

- Composition:

```java
// Java program to illustrate
// the concept of Composition
import java.io.*;
import java.util.*;

// class book
class Book
{

    public String title;
    public String author;

    Book(String title, String author)
    {

        this.title = title;
        this.author = author;
    }
}
```

```java
class Library
{

    // reference to refer to list of books.
    private final List<Book> books;

    Library (List<Book> books)
    {
        this.books = books;
    }

    public List<Book> getTotalBooksInLibrary(){

       return books;
    }

}
```

Yeniden Kullanılabilirlik prensibi (Reusability)



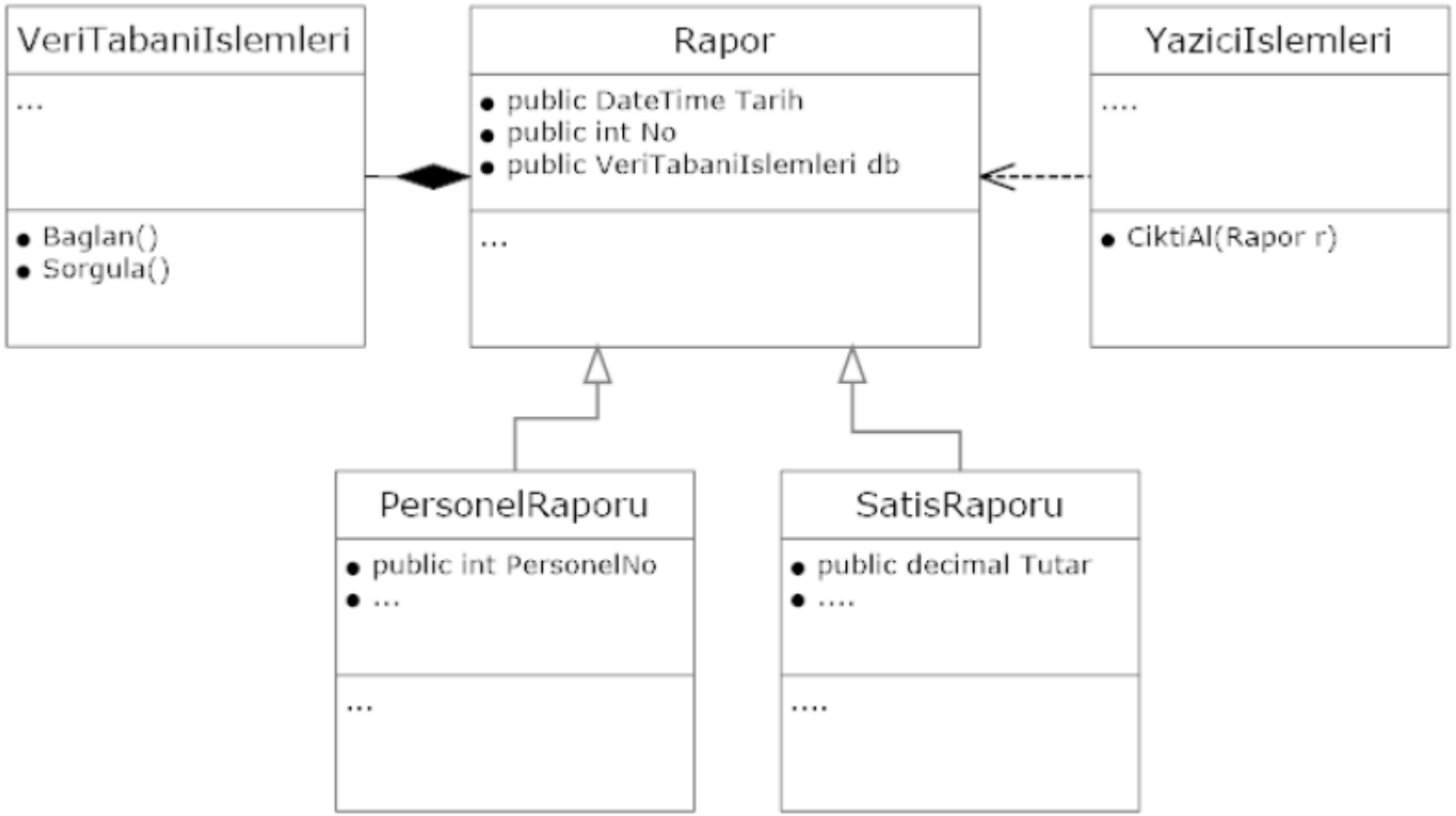
- Rapor sınıfına gereğinden fazla sorumluluk yüklenmiştir.

Bu sınıftan türetilen sınıflar yeniden kullanıma doğru bir örnek teşkil etmemektedirler.

- Doğru olan yaklaşım Rapor sınıfının içinde bu davranışları barındırmak yerine, veri tabanı ve yazıcı işlemlerinin ayrı sınıflarda yapılması ve bu sınıflarla bir ilişki (composition) sağlanmasıdır.

Yeniden Kullanılabilirlik prensibi (Reusability)

İyileştirilmiş tasarım



Yeniden Kullanılabilirlik prensibi (Reusability)

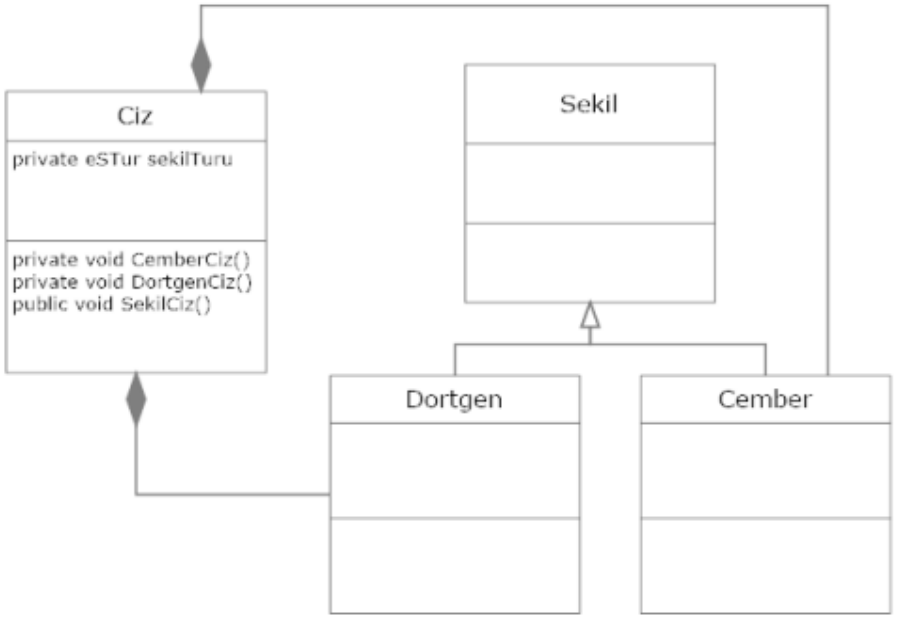
- Veritabanı işlemleri ve yazıcı işlemleri için 2 tane yeni utility sınıfı eklenmiştir.
- Utility sınıfları sayesinde sisteminde reusability'si artmıştır.
- Rapor sınıfı ve utility sınıfları arasında bağımlılık (coupling) oluşmuştur.
- Hem yazıcı hem de veritabanı sınıflarıyla Overt Coupling yani nesneler üzerinden bir coupling gerçekleştirilmiştir.

Açık/Kapalı Prensibi (OCP)

- Yazılım birimleri geliştirilmeye açık, değişikliğe kapalı olmalıdır.
- Değişim gerektirmeyen yazılımın ömrü bitmiştir.
- Yazılımlar, kullanıcı beklentileri değiştikçe değişime uğramaktadır.
- Bu noktada yapılması gereken şey; yazılım sisteminin herhangi bir yerindeki değişimin başka yerlerde de zincirleme değişim gerektirmeyecek şekilde tasarlanmasıdır(esnek olmalıdır).
- Böylece ufak bir değişimde oluşabilecek kargaşa engellenmiş olur. Değişim kargaşasını önlemek için esnemez tasarımlardan uzak durulmalıdır

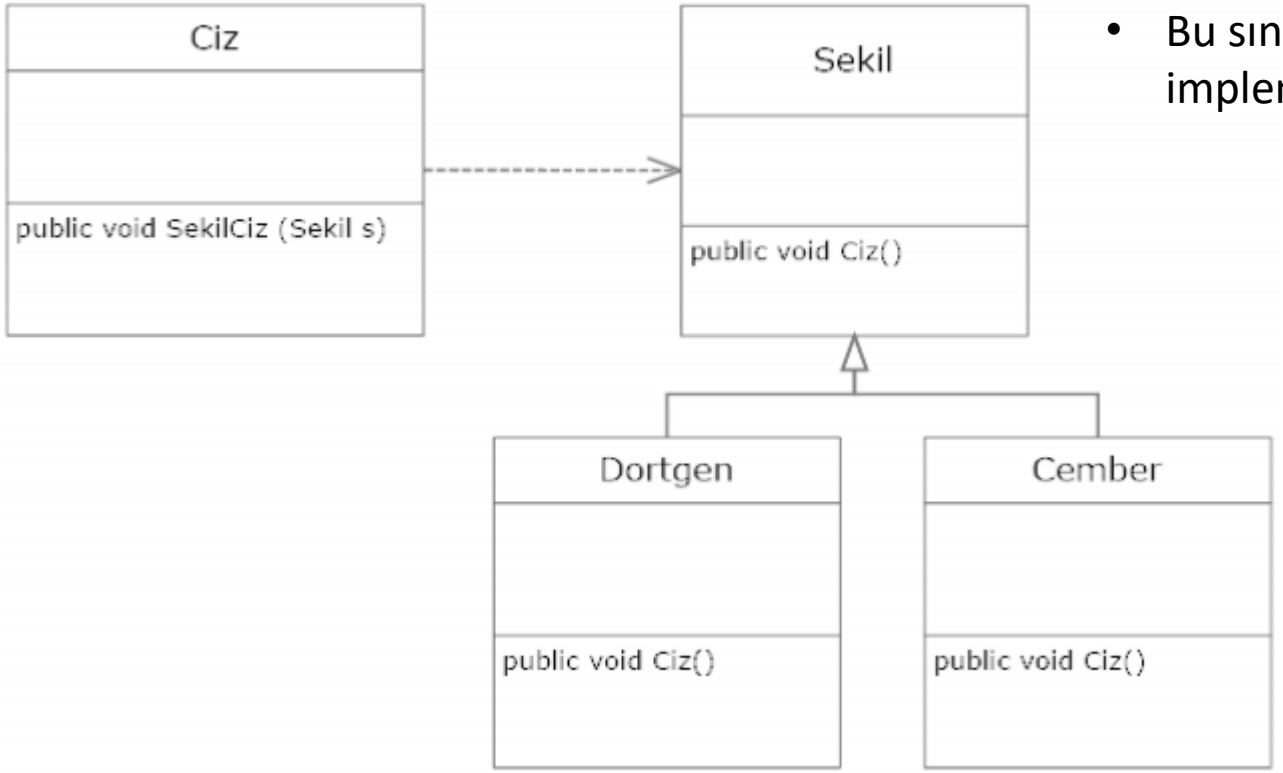
Açık/Kapalı Prensibi (OCP)

- **Bu prensibe göre, sistemlerdeki değişimleri kodları değiştirerek gerçekleştirmek yerine yeni kod blokları eklenerek yapılması öngörülmektedir.



- Yandaki UML ile gösterilen Dortgen ve Cember sınıfları Sekil sınıfından türetilmiştir. Bu sınıflar Ciz sınıfı ile composition yapmaktadır.
- Ciz sınıfı verilen parametreye bağlı olarak şekli çizecek fonksiyonu çalıştıracaktır.
- Böyle bir tasarım esnemezdir ve dolayısıyla kırılgandır.
- Sekil sınıfından yeni bir şekil sınıfı türetildiğinde Ciz sınıfında da değişmesi gerekecektir.

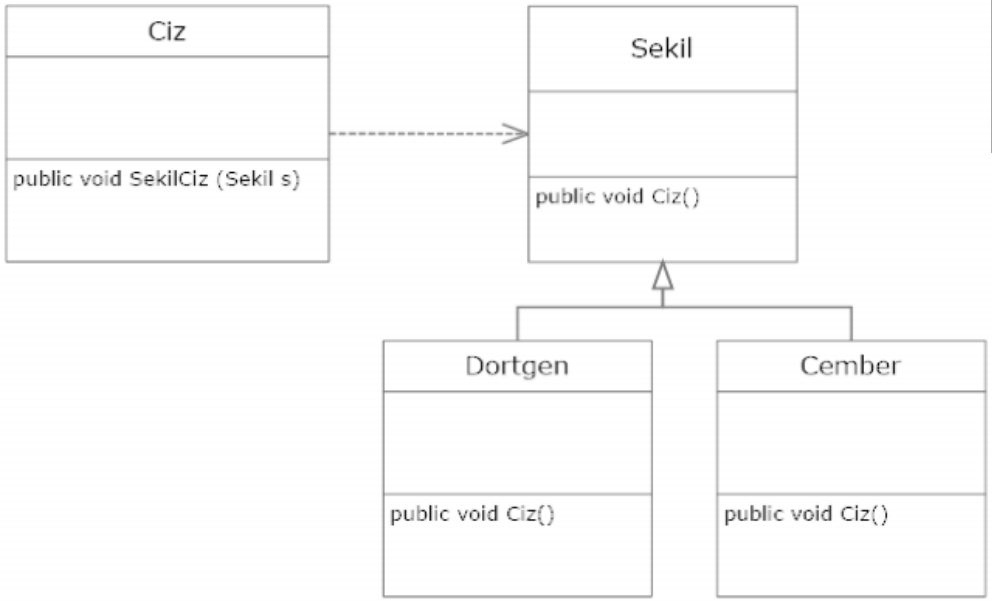
Açık/Kapalı Prensibi (OCP)



- Doğru tasarım Sekil sınıfının soyut sınıf ya da ara yüz şeklinde tanımlanarak yapılabilir.
- Bu sınıftan türetilen sınıflar da kendi ihtiyaçlarına göre implemente edilmelidir.

Açık/Kapalı Prensibi (OCP)

```
public abstract class Sekil {

    public abstract void Ciz();
}
```




```
public class Dortgen extends Sekil {

    @Override
    public void Ciz() {

    }

}
```

```
public class Cember extends Sekil {

    @Override
    public void Ciz() {

    }

}
```

```
public class Ciz
{
    private Sekil sekil;

    public Ciz(Sekil _sekil)
    {
        this.sekil = _sekil;
    }

    public void SekilCiz()
    {
        this.sekil.Ciz();
    }
}
```

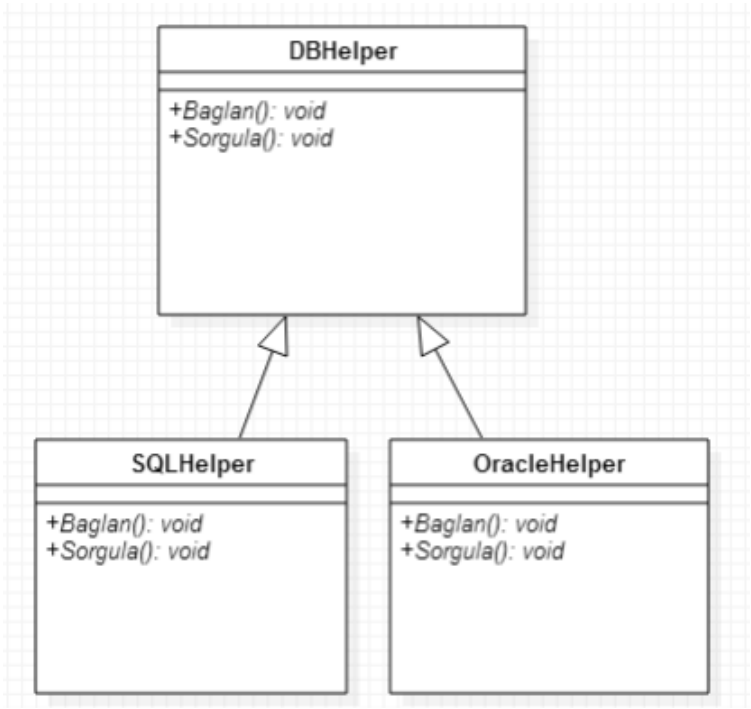
Liskov Yerine Geçme Prensibi (LSP)

- Barbara Liskov tarafından formüle edilen ve Açık-Kapalı prensibi ile yakından ilgili olan bu prensip, türemiş sınıf nesnelerinin taban sınıf nesnesi yerine geçmesini öngörür.
- Alt sınıflardan oluşturulan nesneler üst sınıfların nesneleriyle yer değiştirdiklerinde aynı davranışı göstermek zorundadırlar.
- 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 yapabilmelidir.

Liskov Yerine Geçme Prensibi (LSP)

- **LSP’ye göre herhangi bir sınıf kullanıcısı, bu sınıfın alt sınıfları kullanmak için özel bir efor sarf etmek zorunda kalmamalıdır. Onun bakış açısından üst sınıf ve alt sınıf arasında farklılık yoktur. Üst sınıf nesnelerinin kullanıldığı metotlar içinde alt sınıftan olan nesneler aynı davranışı sergilemek zorundadır.
- Bu prensip türemiş sınıf türündeki nesnelerin taban sınıfa ait nesnelere atanması halinde gerçekleşen otomatik tür dönüşümünden (upcast) faydalanır.

Liskov Yerine Geçme Prensibi (LSP)

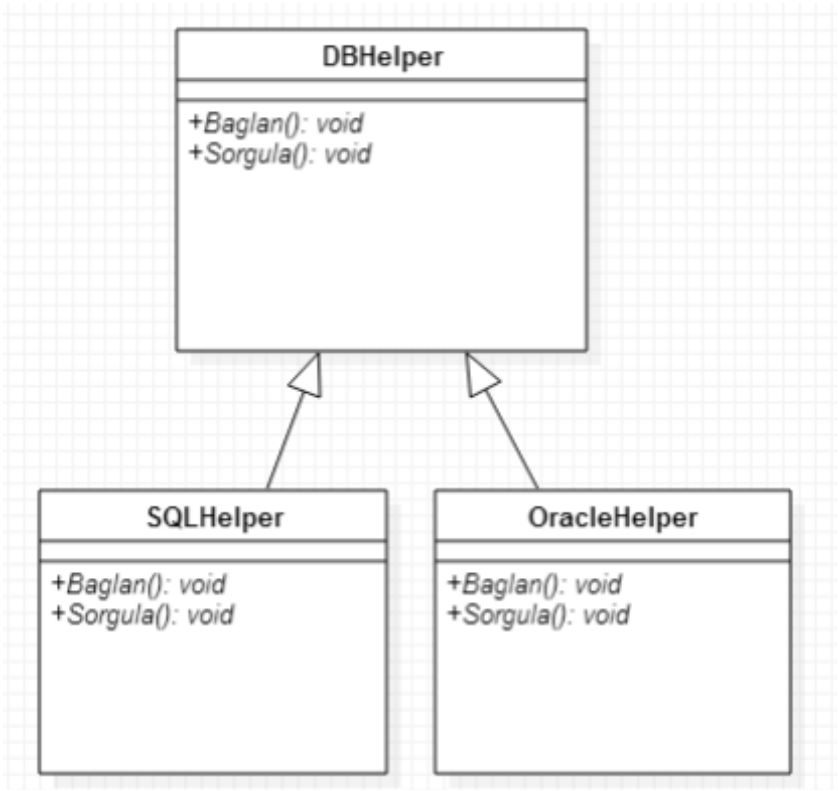



```java
public abstract class DBHelper
{
    public abstract void Baglan();

    public abstract void Sorgu();
}
```

```java
public class OracleHelper extends DBHelper{

    @Override
    public void Baglan() {
        System.out.println("Oracle'a bağlan");
    }

    @Override
    public void Sorgu() {
        System.out.println("Sorgu...");
    }

}
```

```java
public class SQLHelper extends DBHelper {

    @Override
    public void Baglan() {
        System.out.println("Sql'e bağlan");
    }

    @Override
    public void Sorgu() {
        System.out.println("Sorgu...");
    }
}
```

Liskov Yerine Geçme Prensibi (LSP)




```
public class Abstractsinifornek {

    public static void Baglan(DBHelper dbh)
    {
        dbh.Baglan();
    }
    public static void Sorgu(DBHelper dbh)
    {
        dbh.Sorgu();
    }
    public static void main(String[] args) {
       SQLHelper sql= new SQLHelper();
       Baglan(sql);
       Sorgu(sql);
       //sql.Baglan();
       //sql.Sorgu();

       OracleHelper oracle=new OracleHelper();
       Baglan(oracle);
       Sorgu(oracle);
    }
}
```

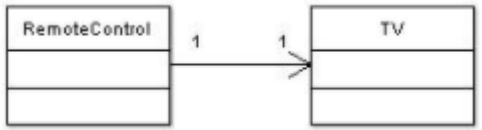
Bağımlılığı Ters Çevirme Prensibi (DIP)

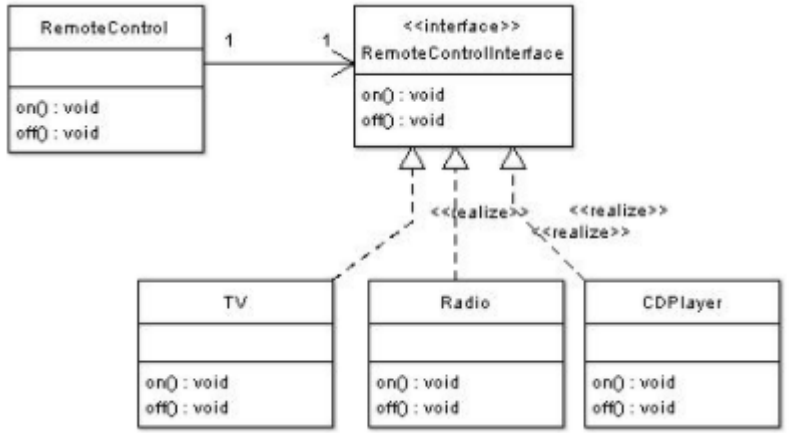
- Bağımlılığı Ters Çevirme prensibi (Dependency Inversion Principle) yüksek seviyeli sınıfların, düşük seviyeli sınıflarla doğrudan bir bağımlılığının olmamasını öngörmektedir.
- **Bu prensibe göre somut sınıflara olan bağımlılıklar soyut sınıflar ve interface sınıflar kullanılarak ortadan kaldırılmalıdır, çünkü somut sınıflar sık sık değişikliğe uğrarlar ve bu sınıflara bağımlı olan sınıflarında yapısal değişikliğe uğramalarına sebep olurlar.
- Bağımlılığın artmaması için

– yüksek seviyeli sınıflar ile

– düşük seviyeli sınıfların arasına bir ara yüz ya da soyut sınıf sokulması gerekmektedir.

Bağımlılığı Ters Çevirme Prensibi (DIP)





Görülen yapı DIP prensibine ters düşmektedir, çünkü **RemoteControl** sınıfı somut bir sınıf olan **TV** sınıfına bağımlıdır.
TV bünyesinde meydana gelen her değişiklik doğrudan RemoteControl sınıfını etkileyecektir. Ayrıca RemoteControl sınıfını TV sınıfı olmadan başka bir yerde kullanılması mümkün değildir.

Bağımlılığı Ters Çevirme Prensibi (DIP)

Notification sınıfımız yüksek seviye bir sınıf olmasına rağmen, daha düşük seviyeli olan Email ve SMS sınıflarına bağımlıdır.

```
public class Email {

    public void sendEmail() {
        //Send emeail
    }
}
```

```
public class SMS {
    public void sendSMS() {
        //Send sms
    }
}
```

```
public class Notification {

    private Email email = new Email();
    private SMS sms = new SMS();

    public void sender() {

        email.sendEmail();
        sms.sendSMS();
    }

}
```

Bağımlılığı Ters Çevirme Prensibi (DIP)

```java
public class Notification {

    private List<Message> messages;

    public Notification(List<Message> messages) {
        this.messages = messages;
    }

    public void sender() {
        for (Message message : messages) {
            message.sendMessage();
        }
    }
}
```

```java
public interface Message {
    void sendMessage();
}
```

```java
public class Email implements Message {

    @Override
    public void sendMessage() {
        sendEmail();
    }

    private void sendEmail() {
        //Send email
    }
}
```

```java
public class SMS implements Message

    @Override
    public void sendMessage() {
        sendSMS();
    }

    private void sendSMS() {
        //Send sms
    }
}
```

**E-mail ve sms sınıflarıyla doğrudan bağlantılı olmayan ve soyut olarak tanımladığımız Message arayüzünü kullanan bir yapı haline getirdik. Yüksek seviye bir sınıfın alt seviye sınıflara olan bağımlılığını ortadan kaldırarak artık soyut katman üzerinden işlemleri yapabiliyoruz.

Arayüz Ayrımı Prensibi/Interface Segregation Principle (ISP)

- Arayüz ayrım prensibi olarak da bilinir.
- Sorumlulukların hepsini tek bir arayüze toplamak yerine daha özelleştirilmiş birden fazla arayüz oluşturmalıyız.
- Sınıflar, ihtiyaç duymadıkları metotların bulunduğu Interface'lere bağlı olmaya zorlanmamalıdır.

Arayüz Ayrımı Prensibi/Interface Segregation Principle (ISP)

- Çeşitli mesaj tipleri (Email, SMS vb.) göndermek isteyen bir uygulama için IMessage isimli bir interface yarattığımızı varsayalım.

```
interface IMessage
{
    1 reference
    IList<String> ToAddresses { get; set; }
    1 reference
    string MessageBody { get; set; }
    1 reference
    string Subject { get; set; }
    1 reference
    bool Send();
}
```

Arayüz Ayrımı Prensibi/Interface Segregation Principle (ISP)

**Imessage üzerindeki tüm üye ve metotlar kullanılıyor

```
public class SmtpMessage : IMessage
{
    1 reference
    public string MessageBody { get; set; }
    1 reference
    public string Subject { get; set; }
    1 reference
    public IList<string> ToAddresses { get; set; }
    1 reference
    public bool Send()
    {
        //Gönderme işlemi
        return true;
    }
}
```

**Sorun var: Subject kullanılmıyor

```
public class SmsMessage : IMessage
{
    2 references
    public IList<String> ToAddresses { get; set; }
    2 references
    public string MessageBody { get; set; }
    2 references
    public bool Send()
    {
        //Gönderme işlemi
        return true;
    }
    2 references
    public string Subject
    {
        get
        {
            throw new NotImplementedException();
        }
        set
        {
            throw new NotImplementedException();
        }
    }
}
```

Arayüz Ayrımı Prensibi/Interface Segregation Principle (ISP)

Arayüzler ayırıldı.
BccAddress özelliği eklendi.

```
interface INewMessage
{
    0 references
    IList<String> ToAddresses { get; set; }
    0 references
    string MessageBody { get; set; }
    0 references
    bool Send();
}
```

```
interface IEmailMessage : INewMessage
{
    0 references
    string Subject { get; set; }
    0 references
    IList<String> BccAddresses { get; set; }
}
```

```
public class SmsMessage : INewMessage
{
    2 references
    public IList<String> ToAddresses { get; set; }
    2 references
    public string MessageBody { get; set; }
    2 references
    public bool Send()
    {
        //Gönderme işlemi
        return true;
    }
}
```

```
public class SmtpMessage : IEmailMessage
{
    1 reference
    public string MessageBody { get; set; }
    1 reference
    public string Subject { get; set; }
    1 reference
    public IList<String> BccAddresses { get; set; }
    1 reference
    public IList<string> ToAddresses { get; set; }
    1 reference
    public bool Send()
    {
        //Gönderme işlemi
        return true;
    }
}
```

```
public class SmsMessage : INewMessage
{
    2 references
    public IList<String> ToAddresses { get; set; }
    2 references
    public string MessageBody { get; set; }
    2 references
    public bool Send()
    {
        //Gönderme işlemi
        return true;
    }
}
```

SOLID

- **S** — Single-responsibility principle
- **O** — Open-closed principle
- **L** — Liskov substitution principle
- **I** — Interface segregation principle
- **D** — Dependency Inversion Principle

Kaynaklar

- Java ve Java Teknolojileri, *Tevfik KIZILÖREN* – Kodlab Yayınları
- Aykut Taşdelen, C++, Java ve C# ile UML ve Dizayn Paternleri, Pusula Yayıncılık, İstanbul, 2014 • Eric Freeman, Head First Design Patterns, O'Reilly Media,2004 • Stephen Stelting & Olav Maassen, Applied Java™ Patterns, Prentice Hall PTR ,2001 • http://www.AlgoritmaveProgramlama.comDr Öğr. Üyesi Zehra Aysun ALTIKARDEŞ Nesne Yönelimli Programlama 2 Ders notları
- https://docs.oracle.com/cd/E17802_01/j2se/j2se/1.5.0/jcp/beta1/apidiffs/java/awt/FlowLayout.html
- Yazılım Mühendisliği CBU-Dr. Öğr. Üyesi Deniz Kılınç Yazılım Mimarisi ve Tasarımı Ders Notları
- Yazılım Kalitesi ve Kötü Tasarım Belirtileri | by Ramazan Ümit Bülbül | Medium
- Coupling ve Cohesion Kavramları Nedir? – KodEdu
- Java Association - Aggregation and Composition in Java – TechVidvan
- Association, Composition and Aggregation in Java – GeeksforGeeks
- Liskov Substitution Principle (LSP) – Liskov'un Yerine Geçme Prensibi – KurumsalJava.com – Özcan Acar
- Dependency Inversion Principle (DIP) – Bağımlılıkların Tersine Çevrilmesi Prensibi – KurumsalJava.com – Özcan Acar
- SOLID Nedir ? Solid Yazılım Prensipleri Nelerdir ? | by Gökhan Ayrancıoğlu | Medium

Kaynaklar

- Coupling ve Cohesion Kavramları | by Mehmet Serkan Ekinci | Medium
- Yazılım Çorbası: Cohesion Nedir? - Odaklılık Diyebiliriz (yazilimcorbasi.blogspot.com)
- https://www.geeksforgeeks.org/cohesion-in-java/
- Nesneye Dayalı Programlama'da Temel Tasarım Prensipleri (itu.edu.tr)
- Yazılım Tasarım Prensipleri 1 (wordpress.com)
- https://gokhana.medium.com/single-responsibility-prensibi-nedir-kod-%C3%B6rne%C4%9Fiyle-soli%CC%87d-c8b1602be602#:~:text=Single%20responsibility%20prensibi%20s%C4%B1n%C4%B1flar%C4%B1m%C4%B1z%C4%B1n%20iyi,yaln%C4%B1zca%20bir%20i%C5%9Fi%20olmas%C4%B1%20gerekir.
- http://www.kurumsaljava.com/2009/10/14/single-responsibility-principle-srp-tek-sorumluk-prensibi/
- Composition ile Aggregation Arasındaki Fark | Mehmet Kordacı (mehmetkordaci.com)
- Association vs. Aggregation vs. Composition | #Include <Karabük> (includekarabuk.com)
- Birliktelik, Münasebet ve Oluşum (Association, Aggregation and Composition) – Bilgisayar Kavramları (bilgisayarkavramlari.com)