TASARIM VE PAKETLEME PRENSİPLERİ

Yazılım Tasarımı ??

Yazılım öncelikle tasarlanmalı

- Hangi dil ?
- Hangi ortam ?
- Hangi Yöntem?
- OOP ise
  - Hangi paketler
  - Hangi sınıflar
  - Sınıf tasarımı ??

Kötü Tasarım Belirtileri

- Esnemezlik (rijidite)
  Değişime karşı gösterilen direnç
- Kırılganlık (Fragility)
  Yapılan güncelleme ile yazılımın çökmesi
- İmmobilite
  Yazılımı oluşturan alt parçalar başka yazılıma taşınabilmeli !!!
  Alt parçalar arasındaki güçlü bağımlılıklar sebebi ile bu işlemin mümkün olmama durumu
- Viskosite – bağımlılık
  - Tasarım viskositesi
    Olası değişiklik sırasında tasarımı koruma eğilimi
    Viskositesi düşük tasarım değişimde bozulacaktır
  - Ortam viskositesi
    Geliştime ortamı ile alakalıdır (IDE)
    IDE ne kadar gelişmiş ise değişimde o kadar kolaylık sağlar

İyi Tasarımın Sağladıkları

- Anlaşılabilir (understandable)
- Esnek (flexible)
- Bakımı kolay (maintainalle)
- Ölçeklenebilir (scalable)

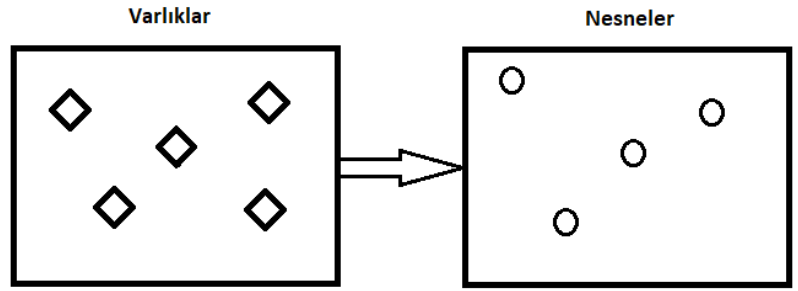
  Büyüyen, gelişen, artan isteklere yanıt vermesi gereken bir sistemin, çalışmanın, işlemin veya yazılımın bu isteklere cevap verme, yönetme ve sorunlarla başa çıkmak yeteneğini

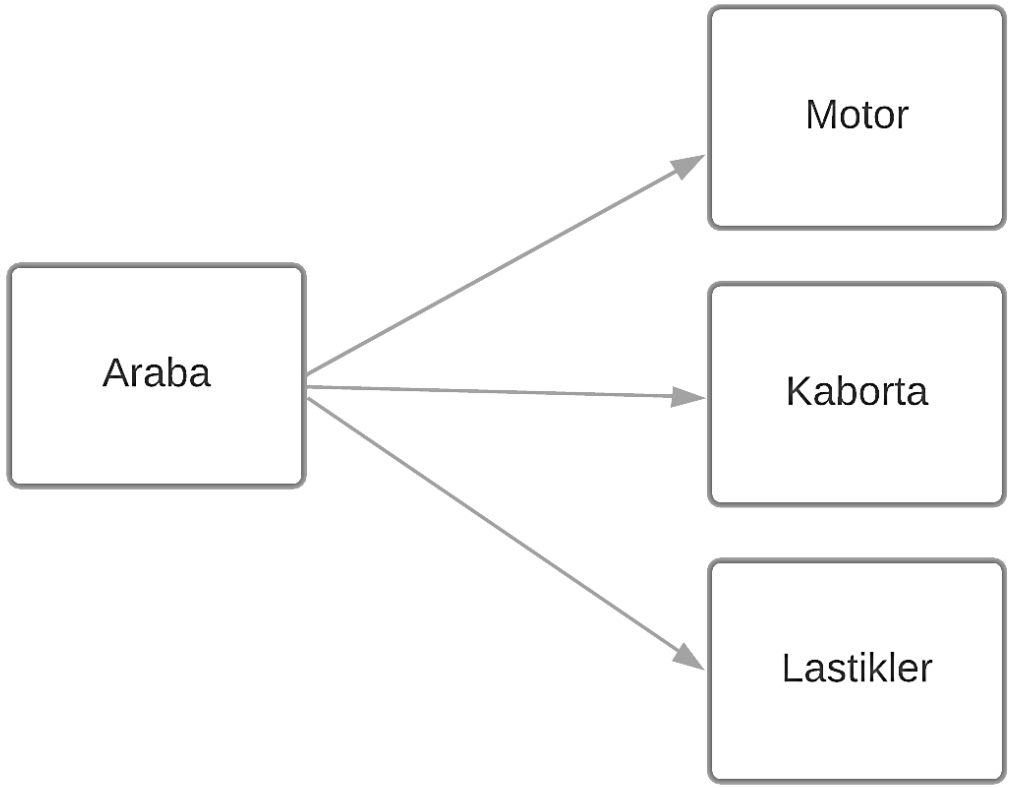
Tasarım prensibleri

- Ayrıştırma
- Uyum (Kohezyon) (artması istenir)
- Bağlantı (Coupling) (artması istenmez)
- Yeniden kullanılabilirlik (Reusability)

Ayrıştırma (Decomposition)

- İlgili varlıklar -> nesneler
- *Nasıl* değil *Neler* sorusu
- Kompleks büyük sistemler – hiyerarşik az komleks sistemler
- Faydaları
  - Az karmaşık, yönetilebilir, anlaşılabilir yapılar
  - Uzmanlaşmış becerilere sahip iş gücünün bölünmesini sağlar.
  - Alt sistemlerin, diğer alt sistemlerden bağımsız olarak değiştirilebilmesi





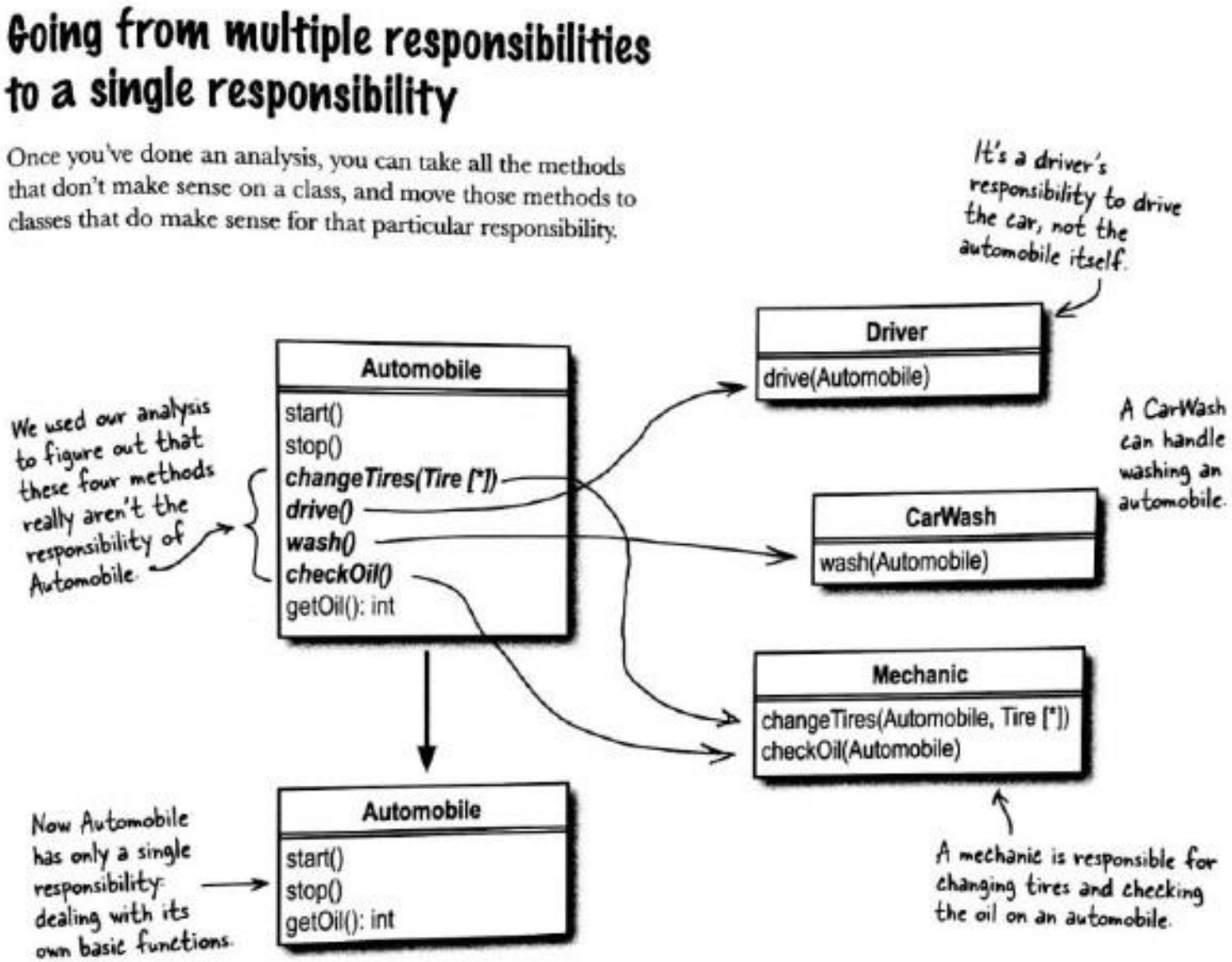
Kohezyon (Uyum)

- Sınıf üyelerinin mantıksal ilişkisi
  - Sınıf içindeki üyelerin ya da metot içindeki görevlerin birbirine mantıksal uzaklığı
- Tasarımdan **yüksek kohezyon** beklenilir
- Yüksek kohezyon yazılım sisteminin
  - Esnekliğini arttırır
  - Bakım ve yeniden kullanılabilirliği kolaylaştırır

```
public class Uye
{
    private String UyeAdi;

    public String getUyeAdi() 

    public void setUyeAdi(String uyeAdi) 

    private void Ekle()

    private void Guncelle()

    private void Yazdir()
    {
        System.out.println("aaa");
    }
}
```

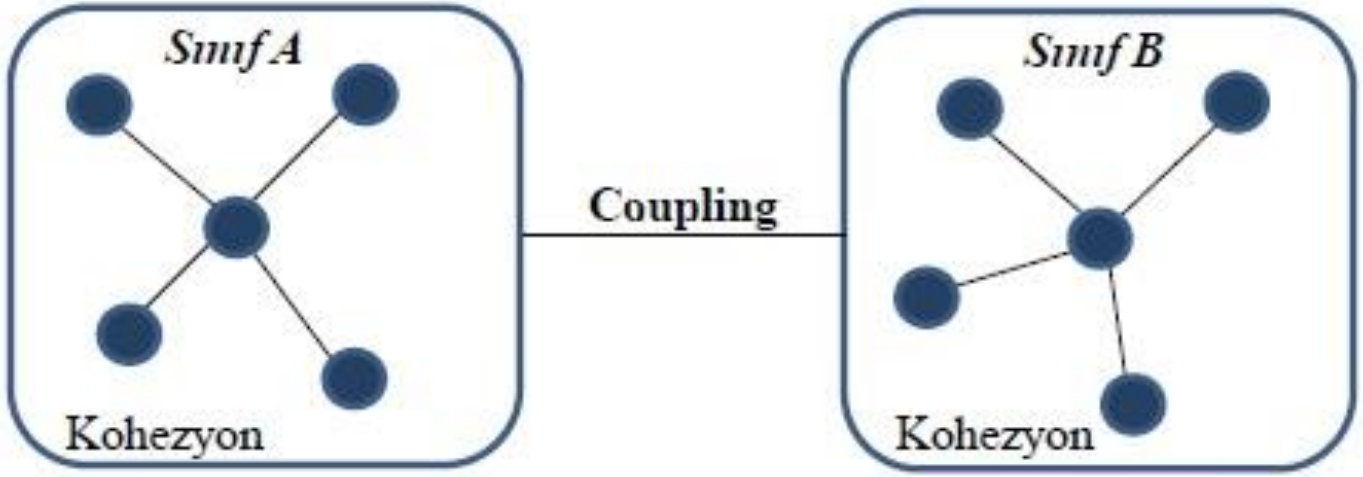
Kohezyon (Uyum)

- Sınıf üyelerinin mantıksal ilişkisi
  - Sınıf içindeki üyelerin ya da metot içindeki görevlerin birbirine mantıksal uzaklığı
- Tasarımdan **yüksek kohezyon** beklenilir
- Yüksek kohezyon yazılım sisteminin
  - Esnekliğini arttırır
  - Bakım ve yeniden kullanılabilirliği kolaylaştırır



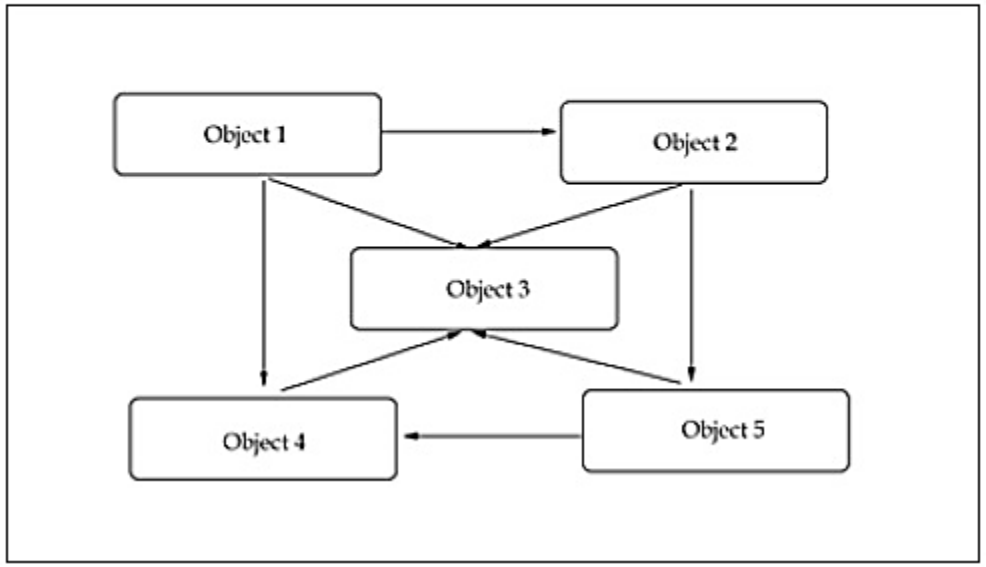
Coupling (bağlantı)

- Nesneler arası ilişkinin nesneleri birbirine ne kadar bağlı kıldığının ölçüsüdür.
- Nesne görevlerinin ne kadar iyi tanımlandığını gösterir
- İyi tasarımdan **küçük coupling** beklenilir



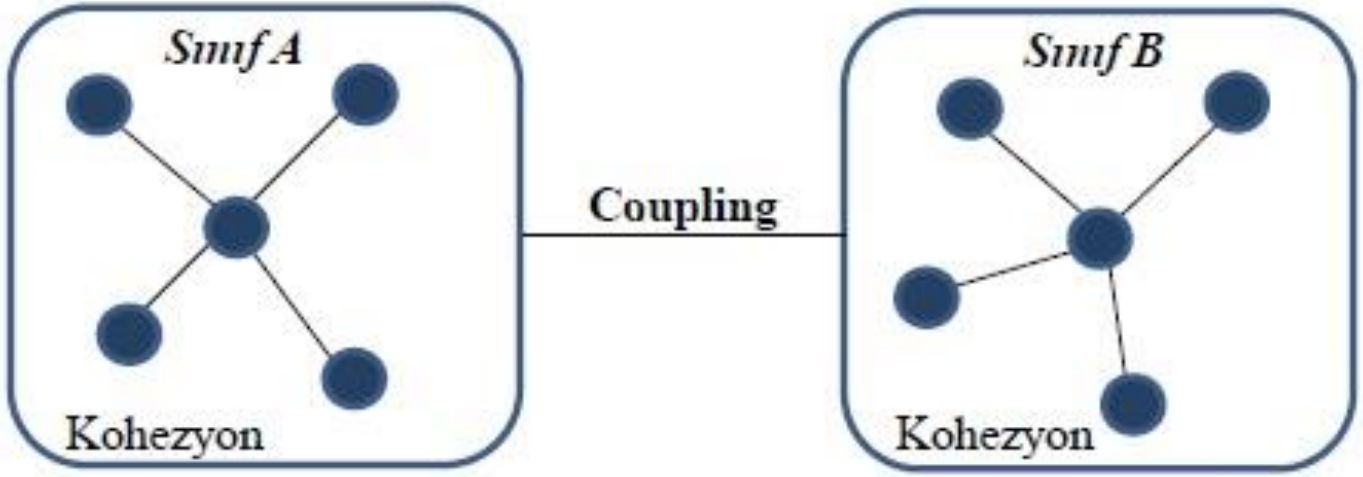
İstenmeyen durum

Tightly coupled bir tasarım…



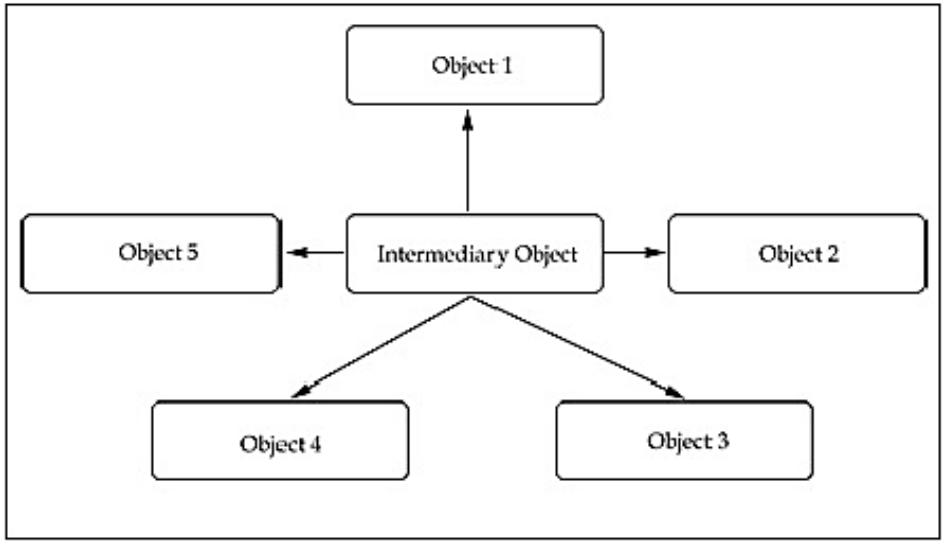
Coupling (bağlantı)

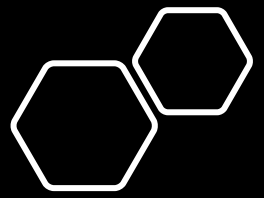
- Nesneler arası ilişkinin nesneleri birbirine ne kadar bağlı kıldığının ölçüsüdür.
- Nesne görevlerinin ne kadar iyi tanımlandığını gösterir
- İyi tasarımdan **küçük coupling** beklenilir



İdeal durum

Loosely coupled bir tasarım…





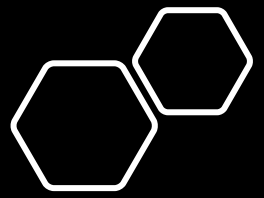
Coupling (Bağlaşım)

- Coupling faktörünün 5 seviyesi:
  1. **Nil Coupling**
  2. Export Coupling
  3. Overt Coupling
  4. Covert Coupling
  5. Surreptitious (Gizlice) Coupling

Nil Coupling

- Teorik olarak en düşük ve en iyi coupling düzeyidir.
- Bu seviyede bağımlılık söz konusu değil
- Diğer sınıflarla hiçbir ilgisi olmayan, tek başlarına kullanılan sınıflardır
- Gerçek hayatta pek mümkün değil

Aykut Taşdelen, C++, Java ve C# ile UML ve Dizayn Paternleri, Pusula Yayıncılık, İstanbul, 2014



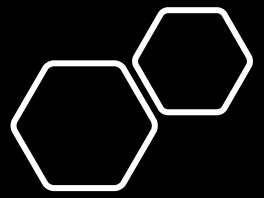
Coupling (Bağlaşım)

- Coupling faktörünün 5 seviyesi:
  1. Nil Coupling
  2. **Export Coupling**
  3. Overt Coupling
  4. Covert Coupling
  5. Surreptitious (Gizlice) Coupling

Export Coupling

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

Aykut Taşdelen, C++, Java ve C# ile UML ve Dizayn Paternleri, Pusula Yayıncılık, İstanbul, 2014



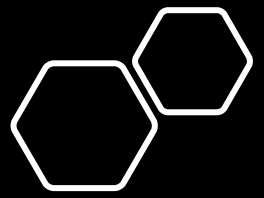
Coupling (Bağlaşım)

- Coupling faktörünün 5 seviyesi:
  1. Nil Coupling
  2. Export Coupling
  3. **Overt Coupling**
  4. Covert Coupling
  5. Surreptitious (Gizlice) Coupling

Overt Coupling

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

Aykut Taşdelen, C++, Java ve C# ile UML ve Dizayn Paternleri, Pusula Yayıncılık, İstanbul, 2014



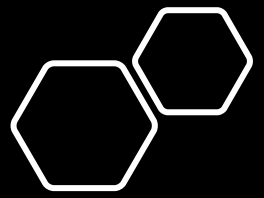
Coupling (Bağlaşım)

- Coupling faktörünün 5 seviyesi:
  1. Nil Coupling
  2. Export Coupling
  3. Overt Coupling
  4. **Covert Coupling**
  5. Surreptitious (Gizlice) Coupling

Covert Coupling

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

Aykut Taşdelen, C++, Java ve C# ile UML ve Dizayn Paternleri, Pusula Yayıncılık, İstanbul, 2014



Coupling (Bağlaşım)

- Coupling faktörünün 5 seviyesi:
  1. Nil Coupling
  2. Export Coupling
  3. Overt Coupling
  4. Covert Coupling
  5. **Surreptitious (Gizlice) Coupling**

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.

Aykut Taşdelen, C++, Java ve C# ile UML ve Dizayn Paternleri, Pusula Yayıncılık, İstanbul, 2014

- Burada kumanda nesnesi görevini yapabilmek için televizyon nesnesine ihtiyaç duymaktadır
- Yani kumanda televizyona bağımlıdır.
- Tasarım kırılgan olup bu bağımlılığım zararları aşağıdaki gibidir:
  - Tv olmazsa kumanda bir işe yaramaz.
  - Tv değiştiğinde kumanda bu değişimden direk etkilenir.
  - Kumanda sadece televizyonu kontrol edebilir başka aletleri kontrol edemez

Dikkat!!! Yanlış Tasarım!!!

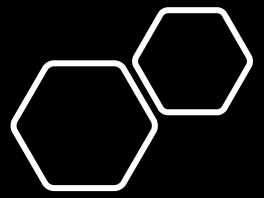
```
public class Televizyon
{
    1 reference
    public void Ac()
    {
        Console.WriteLine("Televizyon açıldı...
    }
}
```

```
public class Kumanda
{
    private Televizyon tv;
    1 reference
    public Kumanda()
    {
        tv = new Televizyon();
    }
    1 reference
    public void TelevizyonAc()
    {
        tv.Ac();
    }
}
```

Overt Coupling var…

Export Coupling'e dönüştürmek istiyoruz.

```
private void Form1_Load(object sender, EventArgs e)
{
    Kumanda k = new Kumanda();
    k.TelevizyonAc();
}
```



- Burada kumanda nesnesi görevini yapabilmek için televizyon nesnesine ihtiyaç duymaktadır
- Yani kumanda televizyona bağımlıdır.
- Tasarım kırılgan olup bu bağımlılığım zararları aşağıdaki gibidir:
  - Tv olmazsa kumanda bir işe yaramaz.
  - Tv değiştiğinde kumanda bu değişimden direk etkilenir.
  - Kumanda sadece televizyonu kontrol edebilir başka aletleri kontrol edemez
- ÇÖZÜM
  - Kumanda ile Televizyonun sınıfsal ve nesnesel bağını kopart
  - Kumanda içerisinde IKumanda interface'i ile bağ kur.
  - Televizyon sınıfı da IKumanda interface'ini implement etsin.
  - Hatta Radyo sınıfını da ekle. O da Ikumanda interface'ini implement etsin.

Televizyon.java

```
public class Televizyon implements IKumanda
{
    @Override
    public void Ac()
    {
        System.out.println("Televizyon açıldı...");
    }
}
```

Kumanda.java

```
public class Kumanda
{
    private IKumanda kumanda;

    public Kumanda(IKumanda _kumanda)
    {
        super();
        this.kumanda = _kumanda;
    }

    public void Ac()
    {
        kumanda.Ac();
    }
}
```

IKumanda.java

```
public interface IKumanda
{
    public void Ac();
}
```

Radyo.java

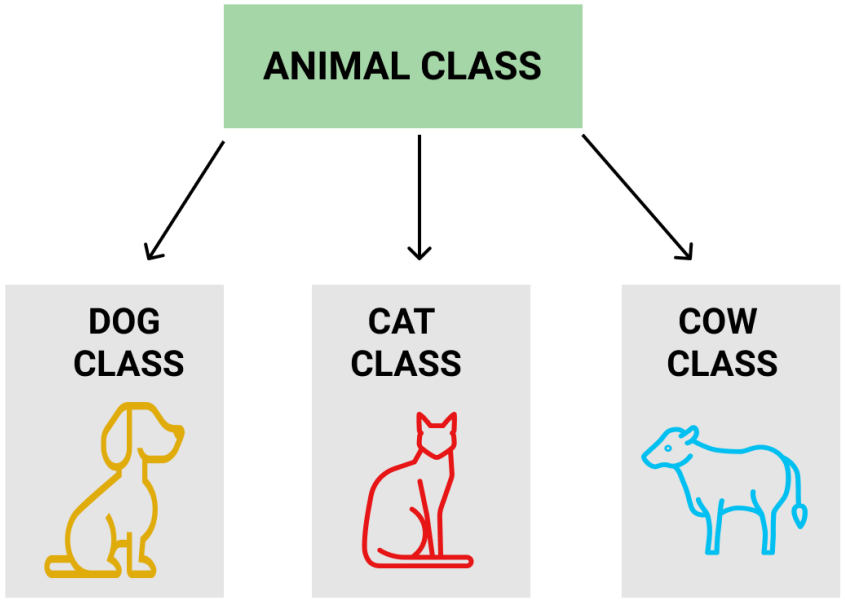
```
public class Radyo implements IKumanda
{
    @Override
    public void Ac()
    {
        System.out.println("Radyo açıldı...");
    }
}
```

KumandaTest.java

```
public static void main(String[] args)
{
    // TODO Auto-generated method stub
    Televizyon t = new Televizyon();
    Kumanda k = new Kumanda(t);
    k.Ac();

    Radyo r = new Radyo();
    Kumanda k2 = new Kumanda(r);
    k2.Ac();
}
```

Reusability

- Mirasın (is a) ya da has-a ilişkileri ile sağlanıla bilen bir özelliktir.
- Bir sınıfın sağladığı özelliği başka sınıfta kullanabilmek
- Aynı kodları tekrar tekrar yazmamalı

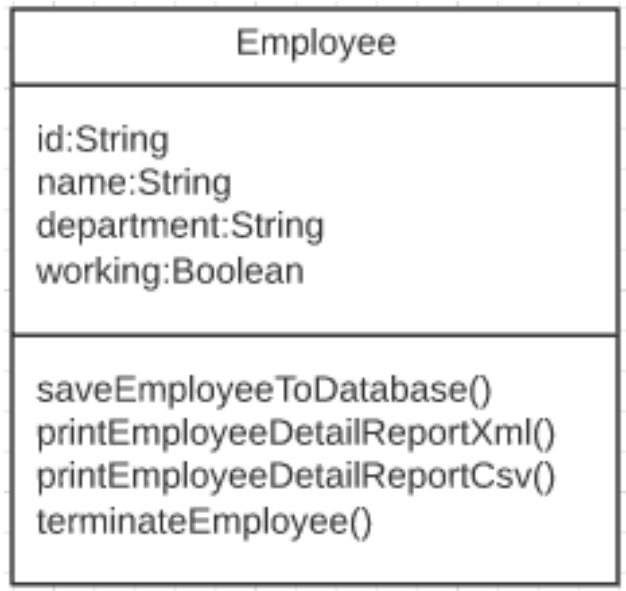


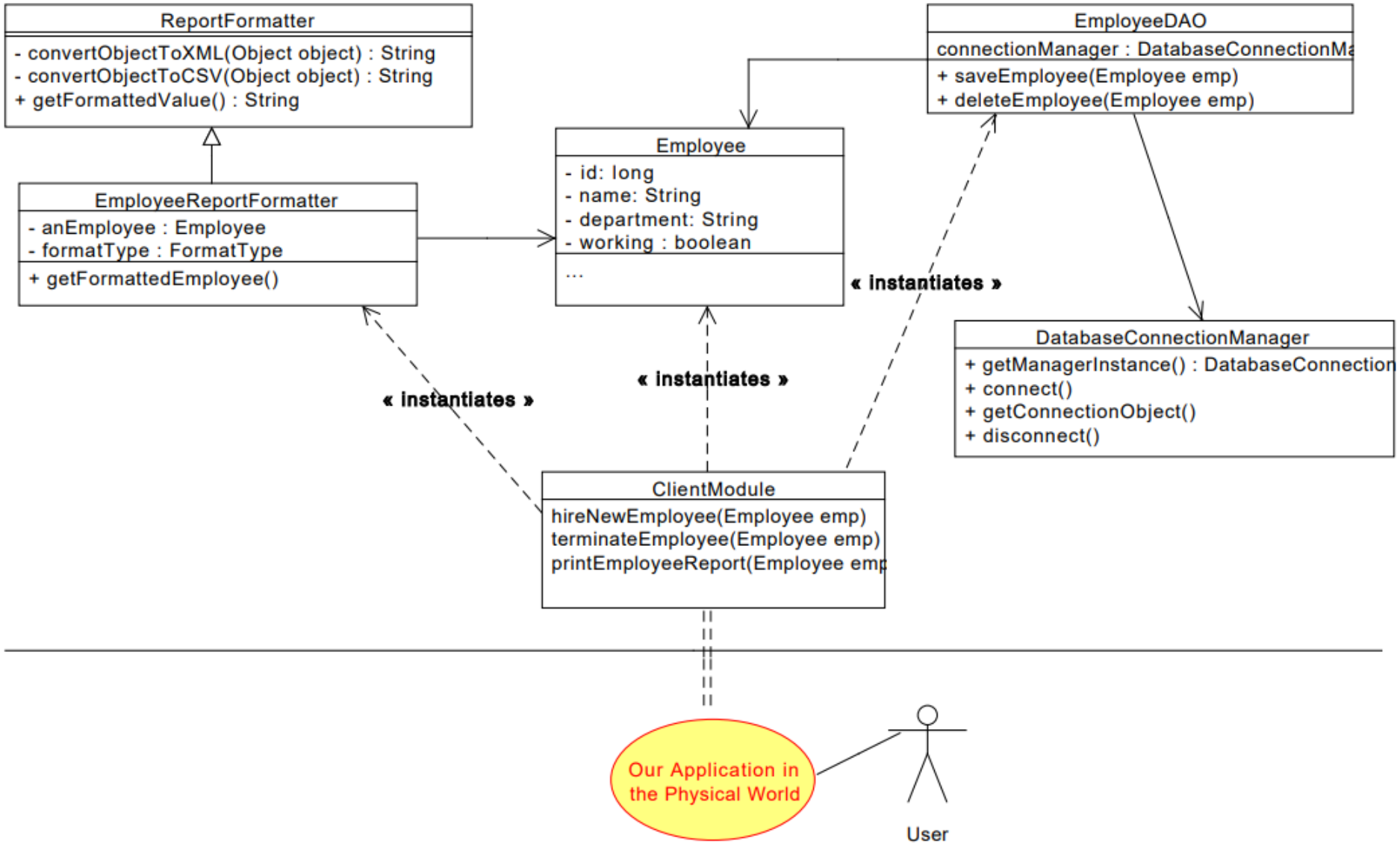
SOLİD prensibleri (Robert Martin)

- Single Responsibility principle (tek sorumluluk prensibi)
- Open/close principle (açık/kapalı prensibi)
- Liskov substitution principle (liskov yerine geçme prensibi)
- Interface segregation principle (arayüz ayırma prensibi)
- Dependency inversion principle (bağımlılığı ters çevirme prensibi)

Single Responsibility principle (tek sorumluluk prensibi)

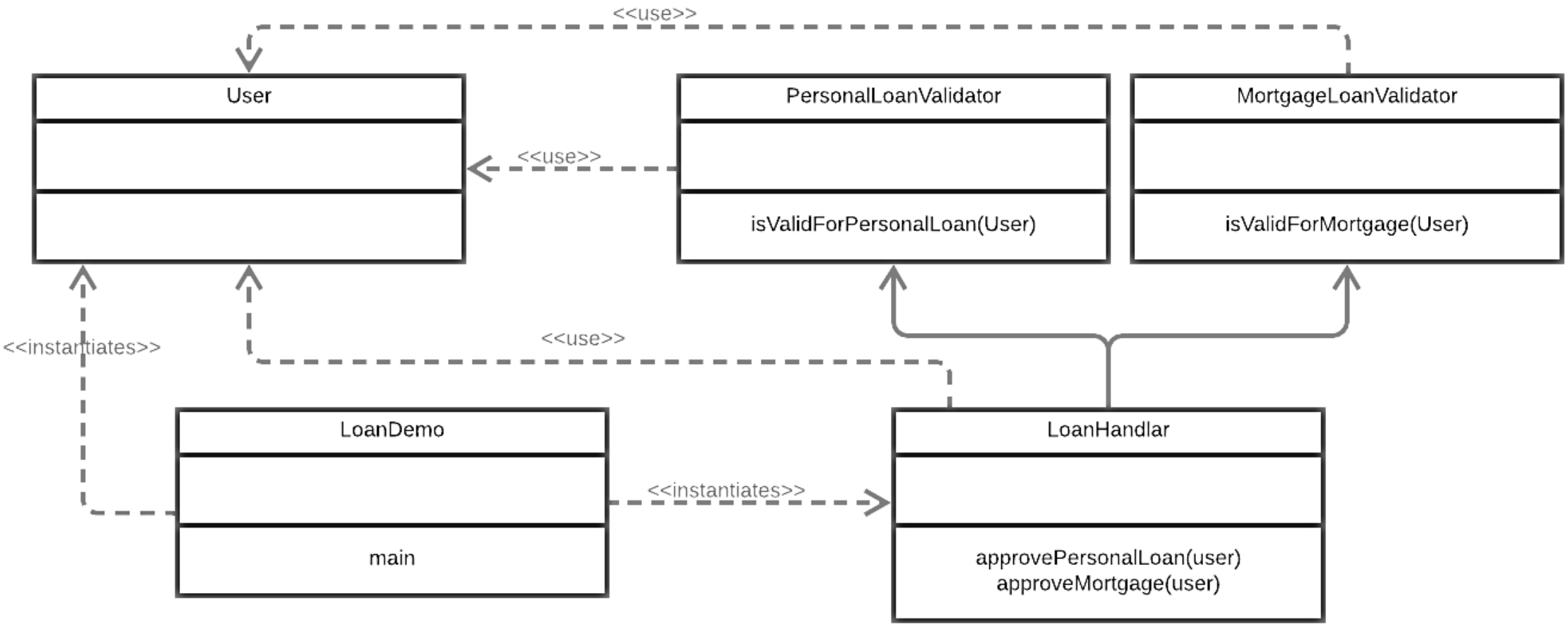
- İyi yapılmış bir modellemede ileride yaşanacak olası değişiklikte sadece değişen durumun sınıfı değişir **geri kalan tasarımda değişiklik olmaz**
  - Her sınıfın sadece bir amacı olmalı
  - **Birden fazla amaç yüklemek tasarım problemidir**
- Bir sınıfın boyutu en optimum ne olmalı??
  - SORUMLULUK BELİRLEYİCİ
- **Bir sınıfı değiştirmek için sadece tek bir gerekçeniz olmalı**
  - Birden fazla gerekçe varsa o sınıfın bölünmesi gerekir
- Dont Repeat Yourself (DRY)
  - Aynı özellikleri birden fazla sınıfta olması gerekiyorsa, bunu ayrı bir sınıf yap


Employee
id:String
name:String
department:String
working:Boolean
saveEmployeeToDatabase()
printEmployeeDetailReportXml()
printEmployeeDetailReportCsv()
terminateEmployee()

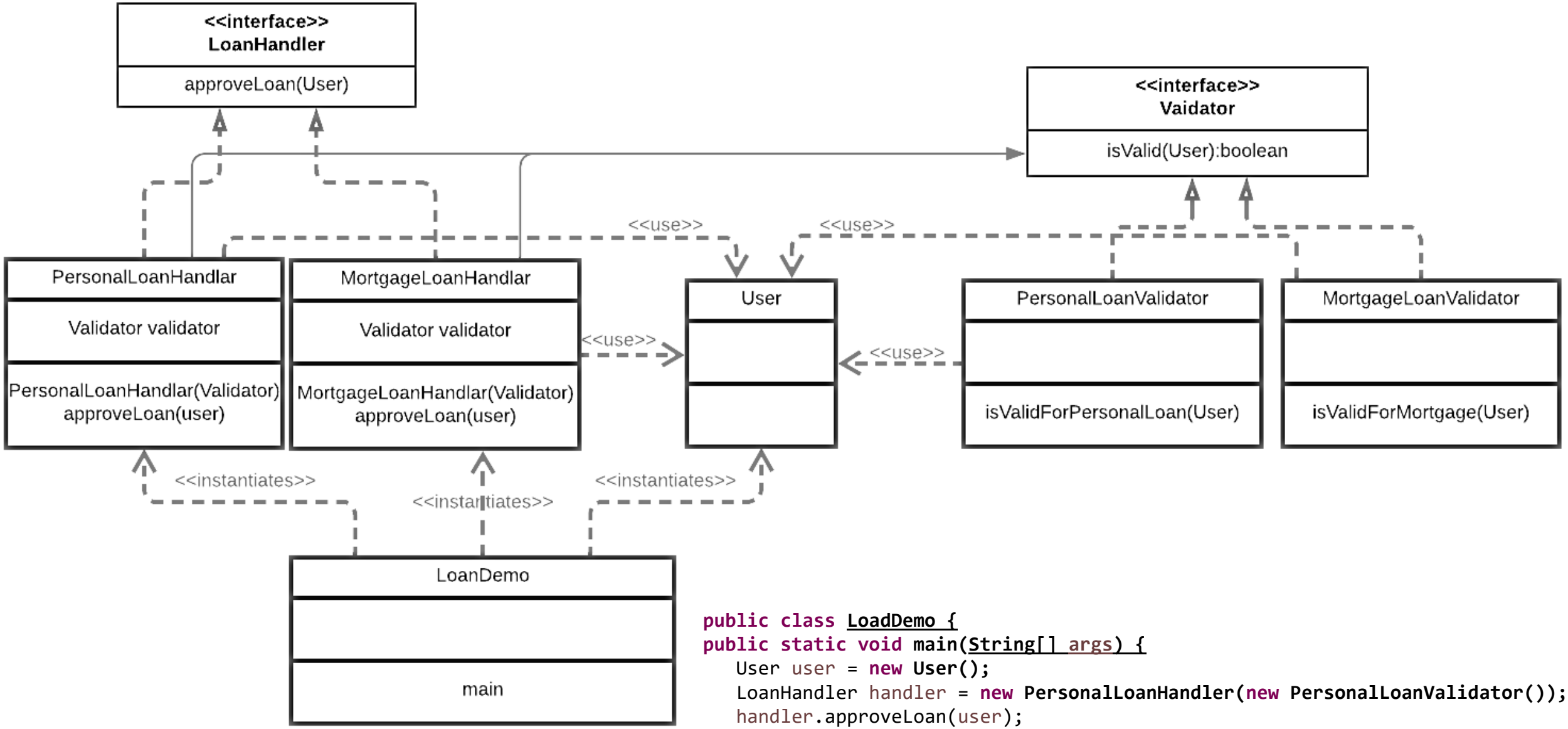

ReportFormatter
- convertObjectToXML(Object object) : String
- convertObjectToCSV(Object object) : String
+ getFormattedValue() : String
EmployeeDAO
connectionManager : DatabaseConnectionMa
+ saveEmployee(Employee emp)
+ deleteEmployee(Employee emp)
Employee
- id: long
- name: String
- department: String
- working : boolean
...
EmployeeReportFormatter
- anEmployee : Employee
- formatType : FormatType
+ getFormattedEmployee()
« instantiates »
DatabaseConnectionManager
+ getManagerInstance() : DatabaseConnection
+ connect()
+ getConnectionObject()
+ disconnect()
« instantiates »
« instantiates »
ClientModule
hireNewEmployee(Employee emp)
terminateEmployee(Employee emp)
printEmployeeReport(Employee emp
Our Application in the Physical World
User

Open/close principle (açık/kapalı prensibi)

- Literatürde Bertrand Meyer tarafından önerildi
- Yazılım modülleri
  - Paketler
  - Sınıflar
  - Metotlar
- Modüller
  - Genişletilmeye açık (open extension)
  - Değiştirilmeye kapalı (close modification)
- Soyutlama ile sağlanabilir
- Faydalar
  - Projenin ölçeklenebilirliğini artırır
  - Sistemi değiştirmek ve yeniliklerle desteklemek kolaylaşır



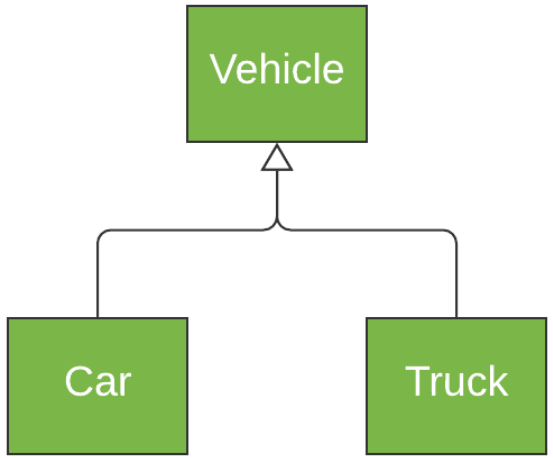

```
public class LoadDemo {
public static void main(String[] args) {
      User user = new User();
      LoanHandler handler = new LoanHandler();
      handler.approveMortgageLoan(user);
      handler.approvePersonalLoan(user);
}
}
```




```java
public class LoadDemo {
public static void main(String[] args) {
   User user = new User();
   LoanHandler handler = new PersonalLoanHandler(new PersonalLoanValidator());
   handler.approveLoan(user);
   LoanHandler handler2 = new MortgageLoanHandler(new MortgageLoanValidator());
   handler2.approveLoan(user);
}
}
```

Liskov substitution principle (liskov yerine geçme prensibi)

- Literatürde Barbara Liskov tarafından önerildi
- Bir programdaki nesneler, alt tiplerin nesneleri ile programın doğruluğunu değiştirmeden yer değiştirilebilir olmalı
- Çok biçimliliğin yanlış kullanılması ile ilgili
- Derleme zamanı hata değil mantıksal hata oluşturmayı engeller
- Alt sınıf her durumda üst sınıfın yerine kullanılabilmeli
  - Downcast yapmak zorunda kalmamalı

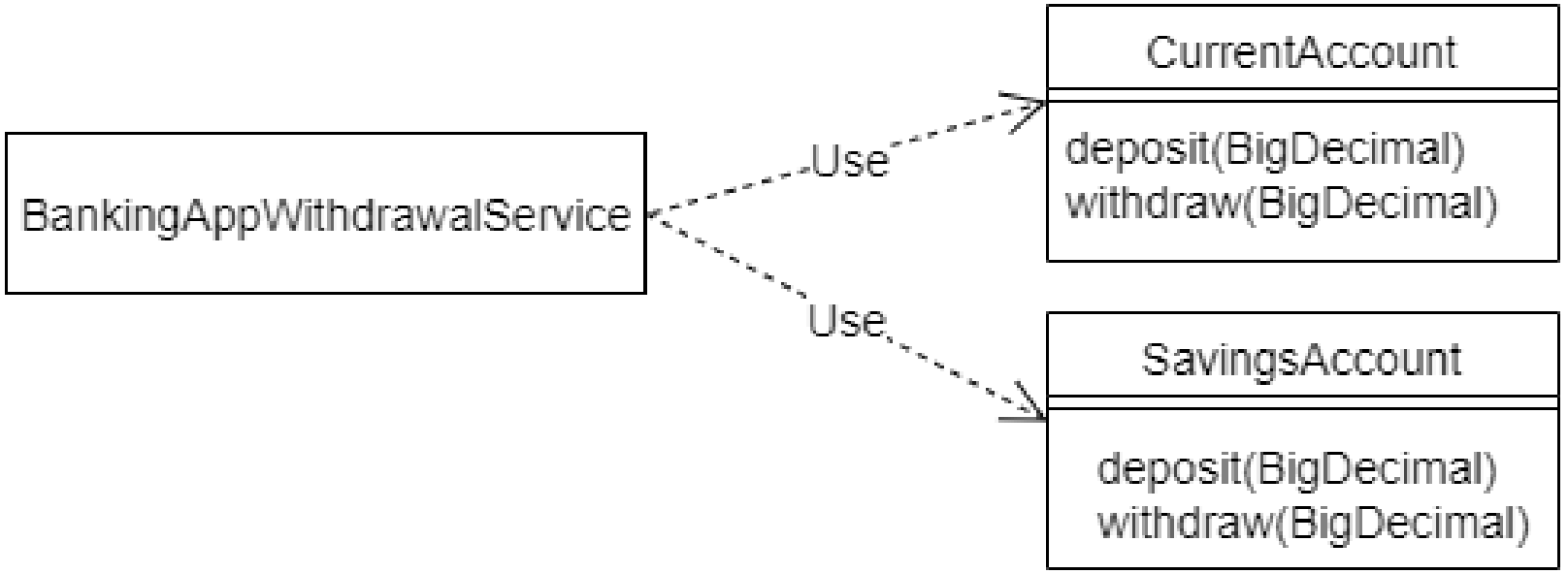


Ancak her sürücü her aracı sürememeli

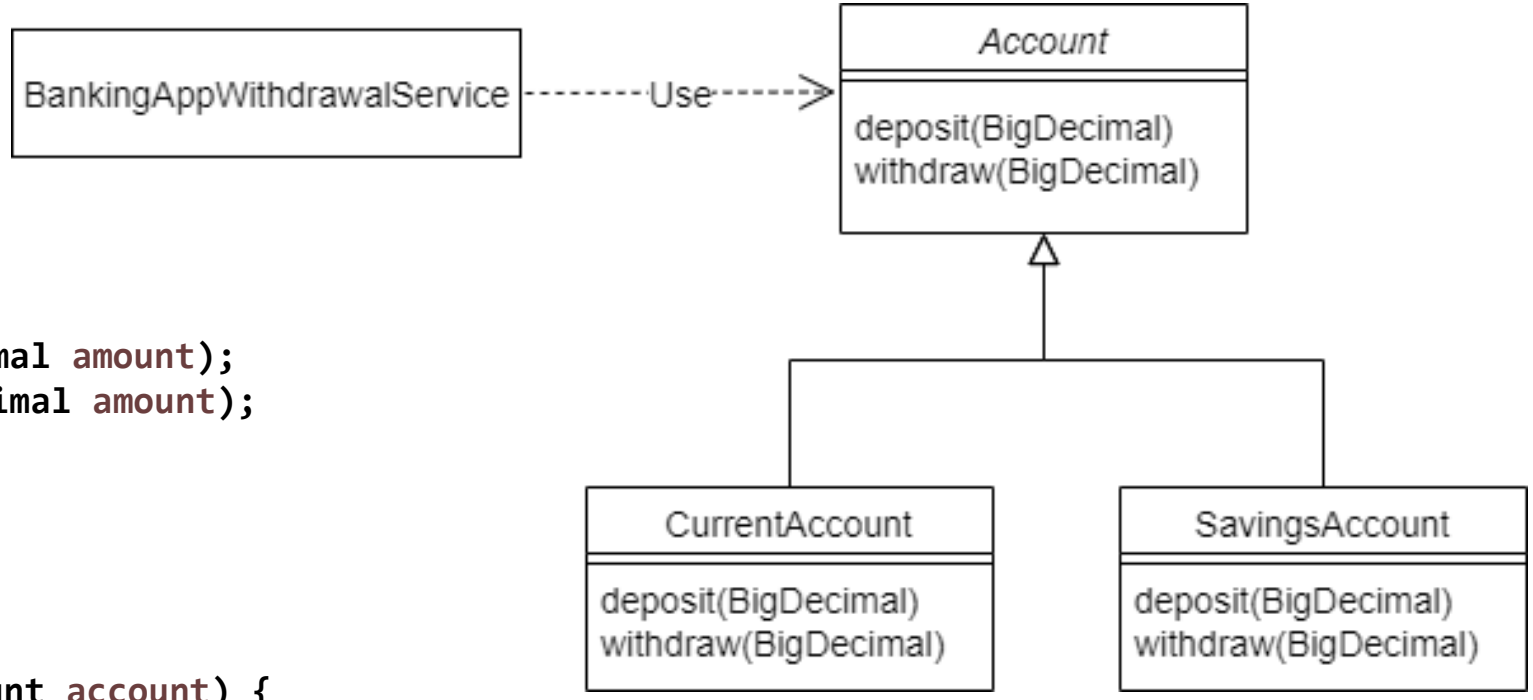
Bir tamirhanede herhangi bir araç tamir edilebilmeli

Sadece uygun sürücü uygun aracı sürebilmeli

NOT : Sınıflar arası ilişkinin bağımlılık (dependency) olduğuna ve yönüne dikkat ediniz.



- Hangi tip hesabın kullanılacağı bilinmesi gerekir
- CurrentAccount ve SavingsAccount birbiri yerine kullanılabilir değil
- BankingAppWithdrawalService uygulamasında asıl işlemler (algoritma) dışında sınıf belirlemek içinde koşul ifadeleri kullanmak gerekecek
- Her yeni sınıfta bu koşulları da güncellemek lazım – Açık/kapalı prensibi ihlali




```java
public abstract class Account {
    protected abstract void deposit(BigDecimal amount);
    protected abstract void withdraw(BigDecimal amount);
}

public class BankingAppWithdrawalService {
    private Account account;

    public BankingAppWithdrawalService(Account account) {
        this.account = account;
    }

    public void withdraw(BigDecimal amount) {
        account.withdraw(amount);
    }
}
```

- Açık/kapalı prensibine uydu.
  - Yeni hesap için bu ihtiyaç için değişime gerek yok gibi

Yeni İhtiyaç

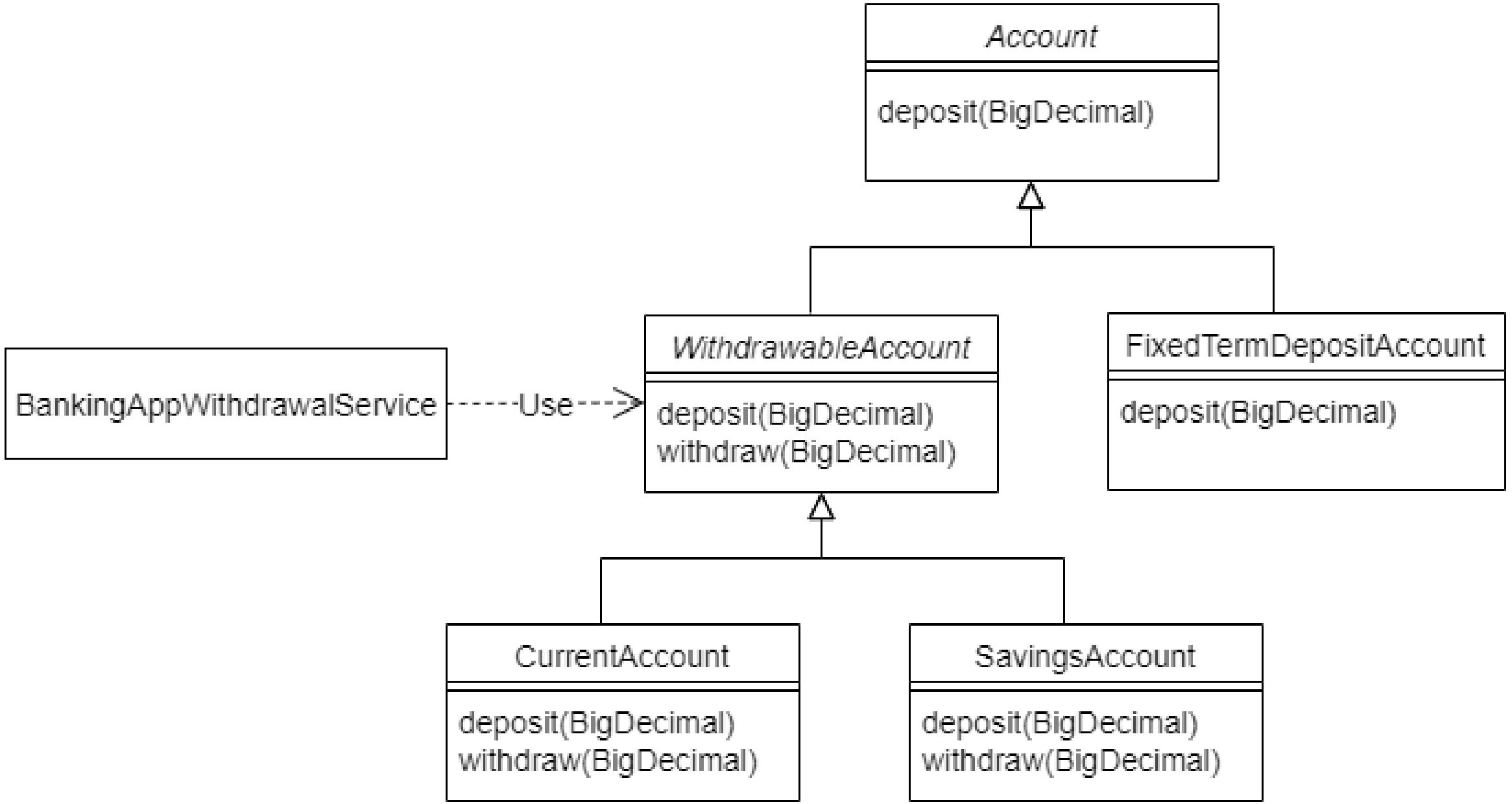
- Eklenecek hesap tipinde para çekme özelliği olmayacak
- Para bir sene sonra otomatik olarak vadesiz hesaba (currentAccaunt) aktarılacak

```java
public class FixedTermDepositAccount extends Account {
    @Override
    protected void deposit(BigDecimal amount) {
        // Deposit into this account
    }

    @Override
    protected void withdraw(BigDecimal amount) {
        throw new UnsupportedOperationException("Withdrawals are not supported by FixedTermDepositAccount!!");
    }
}
```

```java
Account myFixedTermDepositAccount = new FixedTermDepositAccount();
myFixedTermDepositAccount.deposit(new BigDecimal(1000.00));

BankingAppWithdrawalService withdrawalService = new BankingAppWithdrawalService(myFixedTermDepositAccount);
withdrawalService.withdraw(new BigDecimal(100.00));
```

Bu nesnenin withdraw metodu içermemesi gereken **FixedTermDepositAccount** sınıfından nesne ile türetilmesini engelleyecek bir tasarım kullanılmamış.

- Yeni **FixedTermDepositAccount** sınıfının barındırmaması gereken witdraw metodunu içerdiği görülmekte
- Gereksinimler gereği **FixedTermDepositAccount** sınıfı üst sınıf (**Account**) yerine kullanılması uygun değil.
  - İlave gereksinimler alınması gerekiyor.
  - Bu durum çok biçimliliğe zarar veriyor


Account
deposit(BigDecimal)
WithdrawableAccount
deposit(BigDecimal)
withdraw(BigDecimal)
FixedTermDepositAccount
deposit(BigDecimal)
BankingAppWithdrawalService
Use
CurrentAccount
deposit(BigDecimal)
withdraw(BigDecimal)
SavingsAccount
deposit(BigDecimal)
withdraw(BigDecimal)

Alt sınıf her durumda üst sınıfın yerine kullanılabilmeli

1. The signature rule
   a. Metot parametreleri
      - Bu kural, override olmuş alt tür metodu değişken türlerinin, üst tür metodu değişken türleriyle aynı veya onlardan daha geniş olabileceğini belirtir.
      - Java da override kuralları bunu sağıyor
   b. Geri dönüş değer türü
      - Override olmuş alt tür metodunun dönüş türü, üst tür yönteminin dönüş türüne eşit ya da daha dar olabilir.

```
public abstract class Foo {
    public abstract Number generateNumber();
}
public class Bar extends Foo {
    public Integer generateNumber() {
        return new Integer(10);
    }
}
```

Dönüş türü Object olabilseydi metot sonucu alakasız bir sınıfa mesela Truck sınıfı nesnesine atanabilirdi.

Alt sınıf her durumda üst sınıfın yerine kullanılabilmeli

2. The properties rule

a. Sınıf değişkenleri arasındaki ilişki kuralı

- değişkenler arası bir kural varsa nesnenin tüm geçerli durumları için sağlanmalıdır.

```
public abstract class Car {
    protected int limit;
    // ilişki: speed < limit : şart her yerde sağlanmalı;
    protected int speed;
    protected abstract void accelerate();
}
public class HybridCar extends Car {
    // yeni ilişki: charge >= 0;
    private int charge;
    // üstsınıf kuralı: speed < limit: burada da sağlanmalı
    protected void accelerate() {
        // HybridCar hızlanırken speed < limit kuralına uymalı
    }
}
```

b. Geçmiş Kısıtlaması

- Alt sınıf yöntemleri (kalıtsal veya yeni), temel sınıfın izin vermediği durum değişikliklerine izin vermemelidir.

```
public abstract class Car {

    // sadece en başta değer atanabilir
    // daha sonra değer sadece artırılabilir.
    // değer resetlenemez.
    protected int kilometre;

    public Car(int kilometre) {
        this.kilometre = kilometre;
    }
}
public class ToyCar extends Car {
    public void reset() {
    kilometre = 0;
    }
}
```

ToyCar Car sınıfı ile değiştirilebilir değildir. Çünkü kilometre değişkeni sıfırlanmış-kural çiğnenmiş

Alt sınıf her durumda üst sınıfın yerine kullanılabilmeli

3. The Methods rule

a. Ön koşullar

- Override olan bir metot çalışmadan önce bir koşul varsa bu koşul zayıflatılabilir ancak güçlendirilemez

```java
public class Foo {
    // precondition: 0 < num <= 5
    public void doStuff(int num) {
        if (num <= 0 || num > 5) {
            throw new IllegalArgumentException("Input out of range 1-5");
        }
    }
}
```

Limiti güçlendirip 0< num <3 yapma!

```java
public class Bar extends Foo {
    // precondition: 0 < num <= 10
    public void doStuff(int num) {
        if (num <= 0 || num > 10) {
            throw new IllegalArgumentException("Input out of range 1-10");
        }
    }
}s
```

b. Son koşullar

- Override olan bir metod çalıştıktan sonra üretilecek değer üst sınıfın kurallarına uymalı

```java
public abstract class Car {
    protected int speed;
    // son koşul: hız azalmalı
    protected abstract void brake();
}
```

HybridCar'daki override olan fren yöntemi, şarjın da artmasını sağlayarak son durumu güçlendirir. Sonuç olarak, Araba sınıfındaki fren yönteminin son koşuluna dayanan herhangi bir müşteri kodu, Car sınıfı için HybridCar'ı kullandığında ettiğinde hiçbir fark görmez.

```java
public class HybridCar extends Car {
    @Override
    // son koşul: hız azalmalı
    // son koşul: Şarj artırtmalı
    protected void brake() {
    }
}
```

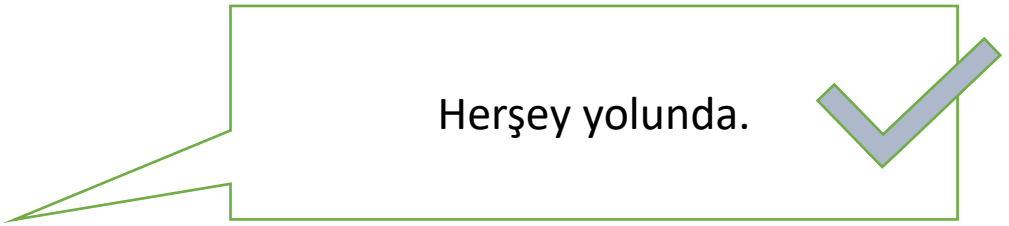
Liskov yerine geçme prensibinin delindiğini nasıl anlarız?

- Bir alt tür, gerçekleştiremediği bir davranış için istisna atıyorsa
- Bir alt tür, gerçekleştiremeyeceği bir davranış için uygulama sağlamıyorsa
- İstemci kodunun instanceof veya downcasting kullanması gerekiyorsa,
- Bir alt tür yöntemi her zaman aynı değeri döndürüyorsa

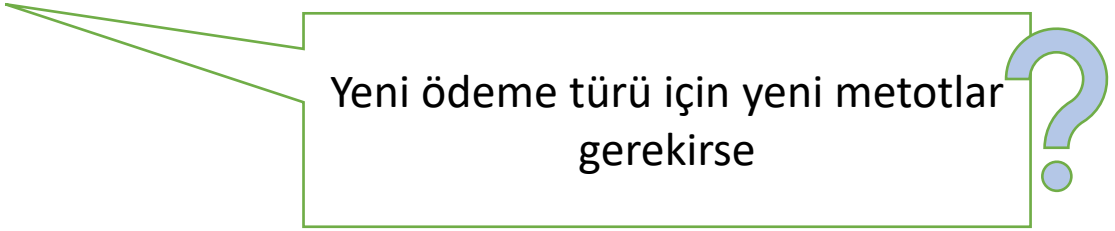
Interface segregation principle (arayüz ayırma prensibi)

- Sınıflar kullanmayacakları metotların implemantasyonuna zorlanmamalı
- Soyut sınıf içerikleri şişkin olmamalı
  - Farklı amaçlar için çok fazla metot içermemeliler
  - Alt sınıfların bütün metotları gerçekleştirmesi gerekecektir.
- Bir uygulamanın tasarım aşamasında daha fazla zaman ve çaba gerektirip ve kod karmaşıklığını artırsa da, sonunda esnek bir kod elde ederiz.

```java
public interface Payment {
    void initiatePayments();
    Object status();
    List<Object> getPayments();
}
```




```java
public class BankPayment implements Payment {
    @Override
    public void initiatePayments() {
    }
    @Override
    public Object status() {
        return null;
    }
    @Override
    public List<Object> getPayments() {
        return Collections.emptyList();
    }
}
```



```java
public interface Payment {
    void initiatePayments();
    Object status();
    List<Object> getPayments();
    //Loan related methods
    void intiateLoanSettlement();
    void initiateRePayment();
}
```

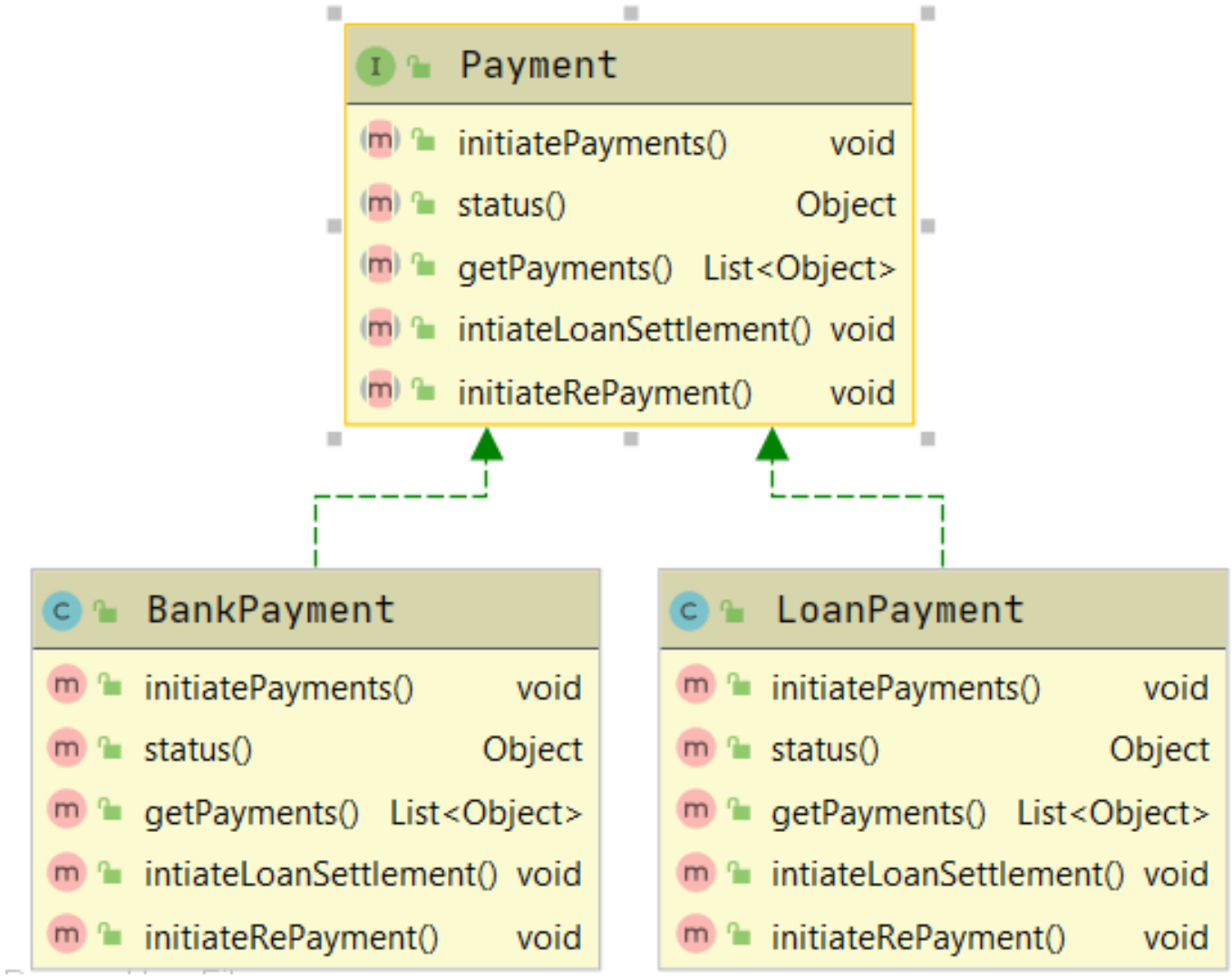
- Zamanla LoanPayment hizmeti eklemeye ihtiyaç oldu.
- Bu hizmet aynı zamanda bir Ödeme türü ancak birkaç metoda daha ihtiyacı var.

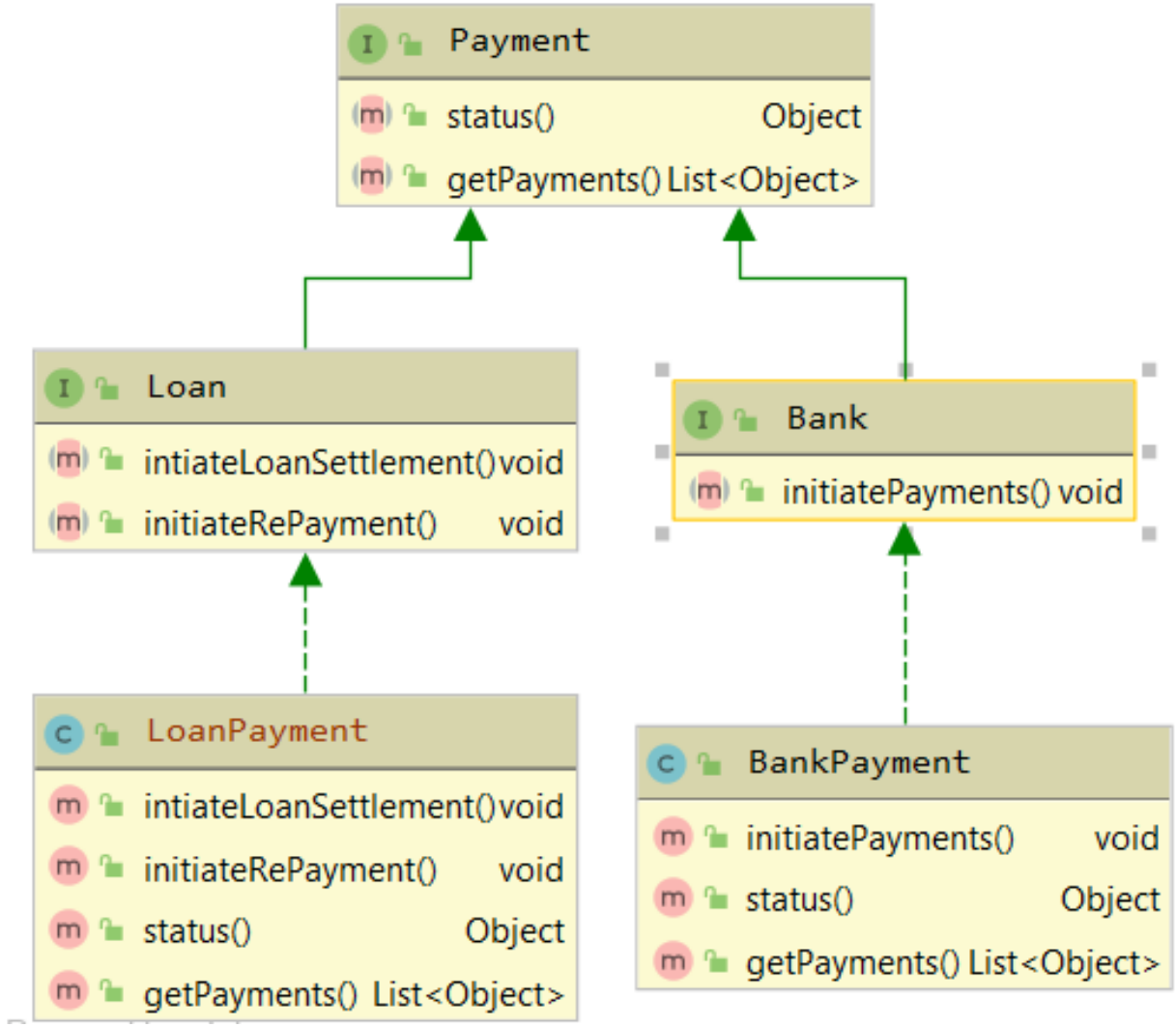
- Ödeme arayüzü değiştiğinden ve daha fazla yöntem eklendiğinden, tüm uygulama sınıflarının artık yeni yöntemleri uygulaması gerekiyor.
- Burada, LoanPayment uygulama sınıfının, herhangi bir gerçek ihtiyaç duymadan initialPayments()'ı uygulaması gerekir.
- Ve böylece, ilke ihlal edilir.

```java
public class BankPayment implements Payment {
    @Override
    public void initiatePayments() {
    }
    @Override
    public Object status() {
        return null;
    }
    @Override
    public List<Object> getPayments() {
        return Collections.emptyList();
    }
    @Override
    public void intiateLoanSettlement() {
        throw new UnsupportedOperationException("This is not a loan payment");
    }
    @Override
    public void initiateRePayment() {
        throw new UnsupportedOperationException("This is not a loan payment");
    }
}
```

- Benzer durum BankPayment sınıfında da var

```java
public class LoanPayment implements Payment {
    public void initiatePayments() {
    throw new UnsupportedOperationException("This is not
a bank payment");
     }
    @Override
    public Object status() {
        return null;
    }
    @Override
    public List<Object> getPayments() {
        return null;
    }
    @Override
    public void intiateLoanSettlement() {
    }
    @Override
    public void initiateRePayment() {
    }
}
```


Payment
initiatePayments() void
status() Object
getPayments() List<Object>
intiateLoanSettlement() void
initiateRePayment() void
BankPayment
initiatePayments() void
status() Object
getPayments() List<Object>
intiateLoanSettlement() void
initiateRePayment() void
LoanPayment
initiatePayments() void
status() Object
getPayments() List<Object>
intiateLoanSettlement() void
initiateRePayment() void


Payment
status() Object
getPayments() List<Object>
Loan
intiateLoanSettlement() void
initiateRePayment() void
Bank
initiatePayments() void
LoanPayment
intiateLoanSettlement() void
initiateRePayment() void
status() Object
getPayments() List<Object>
BankPayment
initiatePayments() void
status() Object
getPayments() List<Object>

```java
public interface Payment {
    Object status();
    List<Object> getPayments();
}
```

```java
public interface Loan extends
Payment {
    void intiateLoanSettlement();
    void initiateRePayment();
}
```

```java
public interface Bank extends
Payment {
    void initiatePayments();
}
```

```java
public class LoanPayment implements Loan {
    @Override
    public void intiateLoanSettlement() {
    }
    @Override
    public void initiateRePayment() {
    }
    @Override
    public Object status() {
        return null;
    }
    @Override
    public List<Object> getPayments() {
        return null;
    }
}
```

```java
public class BankPayment implements Bank {
    @Override
    public void initiatePayments() {
    }
    @Override
    public Object status() {
        return null;
    }
    @Override
    public List<Object> getPayments() {
        return null;
    }
}
```

Dependency inversion principle (bağımlılığı ters çevirme prensibi)

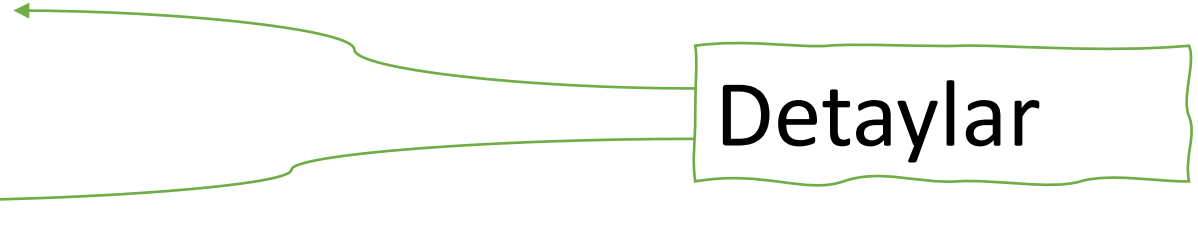
- Üst seviye modüller alt seviye modüllere bağlı olmamalı. İkisi de soyutlamaya bağlı olmalı
- Soyutlama detaya bağlı olmamalı, detay soyutlamaya bağlı olmalı
- Ne sağlar
  - İyi yapılandırılmış,
  - Düşük couplinge sahip
  - Yeniden kullanılabilir

  Kod yazmamızı

Dependency inversion principle (bağımlılığı ters çevirme prensibi)

Farklı hava durumu web servislerinden alınan verilerin ortalamasını alarak hava durumu tahmini yapan bir sınıf yazmak istediğimizi düşünelim

- **`BbcWeatherApi`**
  - Fahrenhayt olarak değer veriyor
- **`AccuweatherApi`**
  - Derece olarak değer veriyor



```java
public class BbcWeatherApi {
    public double getTemperatureFahrenheit() {
        Random rand = new Random();
        return rand.nextDouble()*180;
    }
}
```

```java
public class AccuweatherApi {
    public int getTemperatureCelcius() {
        Random rand = new Random();
        return rand.nextInt(100)-40;
    }
}
```

```java
public class WeatherAggregator {
    private AccuweatherApi accuweather = new AccuweatherApi();
    private BbcWeatherApi bbcWeather = new BbcWeatherApi();

    public double getTemperature() {
        return (accuweather.getTemperatureCelcius()
        + toCelcius(
        bbcWeather.getTemperatureFahrenheit())) / 2;
    }

    private double toCelcius(double temperatureFahrenheit) {
        return (temperatureFahrenheit - 32) / 1.8f;
    }
}
```

AÇIKLAMA:

- WeatherAggregator hava durumunun nasıl alındığını soyutlamış.
  - Her şey yolunda gibi duruyor.
- Üst seviye modül son kullanıcıya en yakın modüldür.
  - Burada WeatherAggregator
  - Bu sınıf alt sınıflara bağımlı (overt coupling-izin dahilinde bağımlılık)
  - Alt sınıflardaki değişim bu sınıfı da etkileyecektir.
- Altsınıflar farklı birimlerde ölçüm değeri veriyor (Derece-Fahrenheit)
  - Detaya bağımlılık sözkonusu

```java
public interface WeatherSource {
    double getTemperatureCelcius();
}
```

```java
public class AccuweatherApi implements WeatherSource
{
    @Override
    public double getTemperatureCelcius() {
        return 30;
    }
}
```

```java
public class BbcWeatherApi implements WeatherSource {
    @Override
    public double getTemperatureCelcius() {
        return toCelcius(getTemperatureFahrenheit());
    }

    private double getTemperatureFahrenheit() {
        return 0;
    }

    private double toCelcius(double temperatureFahrenheit)
    {
        return (temperatureFahrenheit - 32) / 1.8f;
    }
}
```

```java
public class WeatherAggregator {
    private WeatherSource[] weatherSources;

    public WeatherAggregator(WeatherSource[] weatherSources) {
        this.weatherSources = weatherSources;
    }

    public double getTemperature() {
        return Arrays.stream(weatherSources)
            .mapToDouble(WeatherSource::getTemperatureCelcius)
            .average()
            .getAsDouble();
    }
}
```

- Üst seviye modül (`WeatherAggregator`) alt seviye modüllere (`AccuweatherApi`, `BbcWeatherApi`) bağlı değil. İkisi de soyutlamaya (`WeatherSource` interface sine) bağlı
- Soyutlama (`WeatherSource`) detaya (uygulayıcı sınıfın celcius olarak değeri nasıl ürettiğine) bağlı değil, detay soyutlamaya bağlı (implement eden sınıf ölçümünü kendi içinde soyutlayarak celciusa dönüştürmeli)