

Temiz Kod (Clean Code)

3. Bölüm Temel Prensipler



Eğitmen:

Akın Kaldıroğlu

Çevik Yazılım Geliştirme ve Java Uzmanı

“

Bir yazılım tasarıımı geliştirmenin iki yolu vardır. Birisi onu o kadar basit yapmaktır ki hiç bir hata olmadığı açık olsun. Diğerini onu o kadar karmaşık yapmaktır ki açıkta görünen hiç bir hata olmasın.

S

There are two ways of constructing a software design.
One way is to make it so simple that there are
obviously no deficiencies. And the other way is to
make it so complicated that there are no obvious
deficiencies.

”

C. A. R. Hoare,



- **SOLID**
 - Single Responsibility Principle
 - Open-Closed Principle
 - Liskov Substitution Principle
 - Design by Contract
 - Interface Segregation Principle
- Dependency Inversion Principle
- Granularity Principle
- **Demeter kanunu**
- **GRASP**



- Bu bölümde sağlıklı nesne-merkezli tasarımın ve programmanın temel prensipleri ele alınacaktır.
- **SOLID, GRASP** prensip aileleri ile **Design by Contract, Defensive Programming** ve **En az Bilgi Prensibi (Least information principle/Law of Demeter)** işlenecektir.
- Bu prensipler sıkılıkla **Temiz Kod (Clean Code)**, tersine durumlar ise **Kötü Koku (Code Smell)** adı altında da ele alınmaktadır.

Temel Prensipler - I



- Kalıplara geçmeden önce, **yüksek birliktelikli** (**highly-cohesive**) ve **düşük bağımlılıklı** (**lowly-coupled**) yapılar oluşturmak için uygulanması gereken prensipler ele alınmalıdır.
- Bu prensipler yüksek birliktelik ve düşük bağımlılık hedefleri ile tasarım kalıpları mekanizmaları arasında prensipler tabakasını oluştururlar.

Techniques - Patterns

Principles

Coupling & Cohesion

Temel Prensipler - II



- Literatürde farklı prensip listeleri ya da aileleri vardır, en bilinen ikisi şunlardır:
 - **SOLID**
 - **GRASP**

SOLID



SOLID Prensipleri - I



- **SOLID** prensipleri şunlardır:
- **Single Responsibility Principle**
- **Open-Closed Principle**
- **Liskov Substitution Principle**
- **Interface Segregation Principle**
- **Dependency Inversion Principle**

SOLID Prensipleri - II



- Ek olarak bir prensip daha vardır:
 - **Granularity Principle**
- Prensipler farklı kişilerce ortaya konmuştur.
- 90'lı yıllarda Robert C. Martin tarafından **SOLID** prensipleri olarak derlenip OOP Journal ya da C++ Report vb. dergilerde yayınlanmıştır.

SOLID Prensipleri - III



- **SRP, The Single Responsibility Principle:** Bir sınıfın değişmesi için asla birden fazla sebep olmamalıdır. (There should never be more than one reason for a class to change.)
- **OCP, The Open Closed Principle:** Yazılım yapıları (sınıflar, modüller, fonksiyonlar vs.) genişletilmeye açık ama değişime kapalı olmalıdır. (Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.)

SOLID Prensipleri - IV



- **LSP, The Liskov Substitution Principle:** Taban sınıflara işaretçi ya da referans kullanan fonksiyonlar türetilen sınıfların nesnelerini de (gerçek tiplerini) bilmeden kullanabilmelidir. (Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.)
- **ISP, The Interface Segregation Principle:** İstemciler kullanmadıkları arayüzlere bağımlı olmaya zorlanmamalıdır. (Clients should not be forced to depend upon interfaces that they do not use.)

SOLID Prensipleri - V



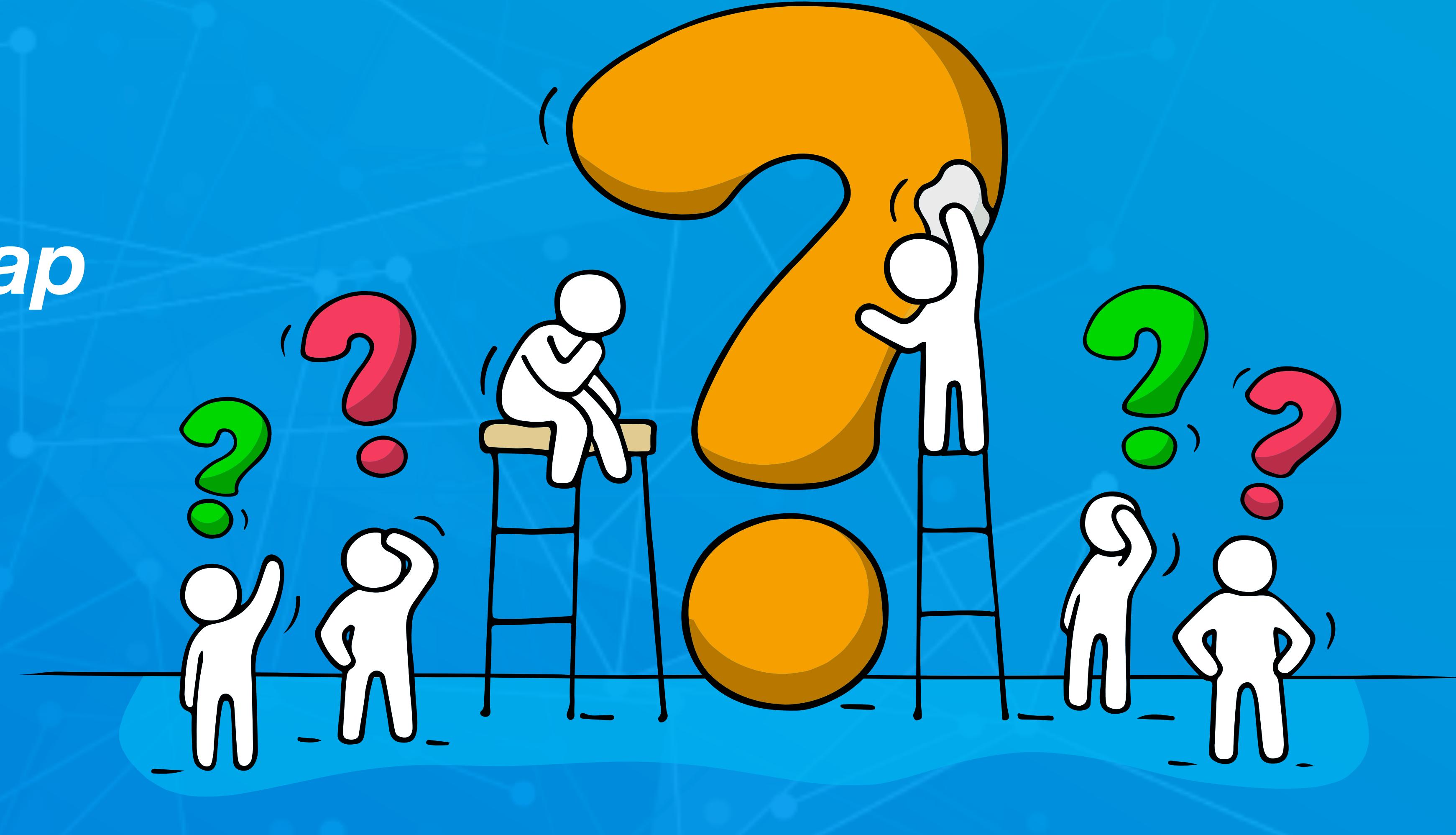
- **DIP, The Dependency Inversion Principle:** Yüksek seviyeli modüller aşağı seviyeli modüllere bağımlı olmamalıdır. İkisi de soyutlamalara bağımlı olmalıdır. Soyutlamalar detaylara bağımlı olmamalı, detaylar soyutlamalara bağımlı olmalıdır. (High level modules should not depend upon low level modules. Both should depend upon abstractions. Abstractions should not depend upon details, details should depend upon abstractions.)

SOLID Prensipleri - VI



- SOLID prensiplerinin her birisi, kendi isimlerini taşıyan ayrı bir makalede açıklanmıştır.
- <http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOOD>
- Ayrıca hepsi özet olarak “**Principles and Patterns**” makalesinde bulunabilir.

Soru ve Cevap Zamani!



Single Responsibility Principle

Single Responsibility Principle - I



- Single Responsibility Principle (SRP) ya da Tek Sorumluluk Prensibi

There Should Never Be More Than One Reason For A Class To Change.

Bir sınıfın değişmesi için asla birden fazla sebep olmamalıdır.

- Bu bağlamda sorumluluktan kasıt, değişim sebebidir (reason to change).

Single Responsibility Principle - II



- **Single Responsibility Principle (SRP)** ya da **Tek Sorumluluk Prensibi**, Bir sınıfın fonksiyonel birlikteliğe sahip olması gerektiğini veciz olarak ifade eder.
- Bir sınıf öyle odaklı olmalıdır ki değişmesi için birden fazla sebep olmamalıdır.
- Bir sınıf sadece ve sadece bir şeyi soyutlamalı ve sadece ona odaklanmalı, onunla ilgili veriye sahip olmalı ve sorumlulukları yerine getirmelidir.
- Dolayısıyla da bir sınıf sadece ve sadece bir soyutlamayla ilgili sebeplerden dolayı değişimlidir.

Separation of Concerns



- **SRP** aslen, Dijkstra'nın **Separation of Concerns** prensibinin bir uygulaması olarak görülebilir.
- Dijkstra, **intelligent thinking**'in, her seferinde konunun bir tarafına (aspect) odaklanarak başarılabilceğini ifade ederken bu terimi kullanmaktadır.
- Bu terim daha sonra programlama kitaplarında kullanılmıştır.

Single Responsibility Principle - III



- **SRP** bu şekilde tanımlandığında anlaşılması ve uygulanması zordur.
- Çünkü “tek sorumluluk”, “değişmek için tek bir sebep” gibi kavramlar muğlaktır, detaylandırılması gereklidir.
- Hele sorumluluğun **değişim sebebi (reason to change)** olarak tanımlanması zaten problemlidir çünkü aslolan sorumluluktur, değişim sebebi ona eklenen bir sonuctur.
- Dolayısıyla bu prensibi sadece sınıf için değil sınıftaki farklı soyutlama seviyeleri için ele almalı ve detaylandırmalıyız.

SRP Seviyeleri - I



- Bu amaçla **SRPyi** sınıfın daha alt ve yukarı düzeylerinde de şöyle tanımlayabiliriz:
 - **Paket (package, name space)**: Birlikte release edilen yapılar aynı pakette olmalıdır.
 - **Sınıf (class)**: Sınıf, sadece bir şeyi soyutlamalı sadece o şeyle ilgili veri ve davranışa sahip olmalıdır.
 - **Metot (method)**: Sınıfın soyutladığı şeyle ilgili, tekrar kullanılabilecek bölünemez tek bir iş yapmalıdır.



- **Blok (block)**: Metot seviyesine çıkamamış dolayısıyla da tekrar kullanımı söz konusu olmayan ama ya hep ya hiç şeklinde çalışan bir grup cümle olmalıdır,
- **Cümle (statement)**: Bir metodun ya da bloğun parçası olarak bir işin tek bir adımını rahat anlaşılır bir şekilde yerine getirmelidir.

SRP Seviyeleri - III



- Bu seviyelerden en tanımlı dolayısıyla da ölçülebilen olanı metottur.
- Diğerlerinin SRP tanımları daha muğlaktır.
- Şimdi, cümleden başlayarak yukarıya doğru giderek SRP'i ele alalım.



Cümle

SRP: Cümle - I



- Cümleyi SRP açısından ele almak bazı zorluklara sahip olsa da temel **SRP** kriterinin **rahat anlaşılma** olduğu söylenebilir.
- Amaç:
- Bir satırda sadece bir cümle (*statement*) olmalı, bir cümle ise sadece bir adımı yerine getirmelidir.
- Bu işlemin ne olabileceği ise daha çok kültüreldir, alışkanlıklara bağlıdır, ortamdan ortama değişebilir.

SRP: Cümle - II



- Bu işlemler şunlar olabilir:
 - Varsa atama operatörü “=” dışında:
 - Bir tane ya da öncelik konusunda anlaşılmaya probleme yol açmayacak birden fazla operatörlü ifade (compound expression),
 - Bir metot çağrısı,
 - Bir kontrol ifadesi,
 - Ya da anlaşılmaya problemi çıkarmayacak şekilde bunların bir kombinasyonu.



- Aşağıdaki satır **SRP** açısından değerlendirildiğinde problemlidir,
- Öncelik karmaşıklığından dolayı anlamada zorluk vardır.

```
double w = a + ++b * c/a * b;
```

- Bu yüzden birden fazla daha rahat anlaşılır satıra bölünmeli, blok hatta kopyalanarak dağılma ihtimali varsa metot olmalıdır.

```
double w = 0.0;
{
    b++;
    w = a + b * b * c / a;
}
```

```
double calculateW(double a, int b, int c){
    b++;
    double w = a + b * b * c / a;
    return w;
}
```

SRP: Cümle - III



- Tek cümlelik karmaşık ifadelerin hiç bir alt parçası başka yerde tekrar kullanılmamalıdır.
- Eğer aynı blokta ya da metotta kullanılacaksa, bu alt parça ayrı bir cümle olmalı ve sonucu yerel bir değişkende saklanıp tekrar tekrar kullanılmalıdır.

Örnek



- Bir cümlede hiç bir zaman tekrar olmamalıdır.
 - Tekrar edecek kısım ayrı bir cümlede olmalı ve yerel bir değişkene atanmalıdır.

```
double w = a + ++b * c/a + b + 2 * c/a;
```

- Bu yüzden sağdaki blok gibi olmalıdır.

```
double w = 0.0;
{
    ++b;
    double d = c / a;
    w = a + b * b * d + 2 * d;
}
```



```
double w = 0.0;
{
    ++b;
    b = b^2;
    double d = c / a;
    w = a + (b + 2) * d;
}
```

Örnek



- Aşağıdaki satır **SRP** açısından değerlendirildiğinde问题li
görünmektedir çünkü **while** döngüsünün şartının anlaşılırlığını
zorlamaktadır.

```
while((remainder = n % prime) == 0){  
    . . .  
}
```

- Alternatif daha anlaşılırdir.

```
remainder = n % prime;  
while(remainder == 0){  
    . . .  
    remainder = n % prime;  
}
```

Örnek



- Ama bu tür satırlar bazen yaygındır ve rahatça anlaşıldığı iddia edilebilir.

```
while((x = in.read()) ≠ 0){  
    . . .  
}
```

- Bazen karmaşık görünen ama çok tanıdık işlemler bir satırda yapılabilir.

```
double distance = Math.sqrt(Math.pow((x1-x2), 2) + Math.pow((y1-y2), 2));
```

SRP: Cümle - IV



- Eğer bir cümleinin ya da bloğun birden fazla yerde bulunması gerekiyorsa, bu cümle ya da blok bir metoda dönüştürülmelidir.
- Hiç bir kod parçası asla tekrar etmemeli ve sistemde sadece ve sadece bir yerde bulunmalıdır.
- **Don't repeat yourself (DRY)** prensibi.



- “w”ı hesaplamak pek çok yerde gereklili bir iş ise bu durumda **copy-paste** ile kodun dağılmasını önlemek için bu iş bir metoda dönüştürülmelidir.

```
public double calculateW(double a, int b, int c){  
    double w = 0.0;  
    ++b;  
    b = b^2;  
    double d = c / a;  
    w = a + (b + 2) * d;  
    return w;  
}
```



- Aşağıdaki cümleyi SRP açısından değerlendirin.

```
cardInfo.setIdNo(cardInfo.getTcCitizen() ?  
    (formUtils.isNullOrEmptyString(verifyOtpWSResponse.getTckn()) ?  
     registrationForm.getTckn() : verifyOtpWSResponse.getTckn()):  
    (formUtils.isNullOrEmptyString(verifyOtpWSResponse.getCustno())) ?  
     registrationForm.getTckn() : verifyOtpWSResponse.getCustno());
```



- Aşağıdaki cümleyi üç açıdan **SRP**'ye terstir:
 - Üç tane ?: operatörü barındırıyor,
 - Pek çok metot çağrısı var,
 - Aynı metot çağrılarını tekrarlıyor.

```
cardInfo.setIdNo(cardInfo.getTcCitizen() ?  
    (formUtils.isNullOrEmptyString(verifyOtpWSResponse.getTckn()) ?  
        registrationForm.getTckn() : verifyOtpWSResponse.getTckn()):  
    (formUtils.isNullOrEmptyString(verifyOtpWSResponse.getCustno())) ?  
        registrationForm.getTckn() : verifyOtpWSResponse.getCustno());
```

Bazen... - I



- Yüksek ifade gücüne sahip yapılar **SRP**'ye tersmiş gibi görünebilirler ama değildir, o türden ifadeler bağlamaşlardır ve kültürel bir norm gibi görülürler:
- Builder kalıbında:

```
return Response.status(Response.Status.OK)
    .entity(message)
    .type(MediaType.TEXT_PLAIN)
    .build();
```

```
Trigger trigger = TriggerBuilder.newTrigger()
    .withIdentity("myTrigger", "myTriggerGroup")
    .startNow()
    .withSchedule(SimpleScheduleBuilder.simpleSchedule().withIntervalInSeconds(1)
    .repeatForever()).build();
```



- Fonksiyonel sitilde stream ile veri işlemede:

```
Arrays.stream(array).filter(x → x % 2 == 0).map(n → n * n).average().ifPresent(System.out::println);
```

```
Arrays.stream(array)
    .filter(x → x % 2 == 0)
    .map(n → n * n)
    .average()
    .ifPresent(System.out::println);
```

- Dekoratör kalıbında:

```
BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(new URL(url).openStream()));
```



- Mock nesnelerini eğitme:

```
when(selam.selamSoyle(anyString())).thenReturn("Selam " + anyString() + " :");
```

- Regular expressions:

```
matcher.group().replaceFirst("href=\"", "")  
    .replaceFirst(">", "")  
    .replaceFirst("\"[\\"s]?target=\"[a-zA-Z_0-9]*", "");
```

SRP: Cümle - V



- Dolayısıyla cümlenin **SRP** açısından temel kriteri anlaşılabilirlik olmakla birlikte yapısı
 - kesinlikle tekrar içermeyecek,
 - öncelik vb. faktörlere bağlı olarak anlaşılmaya problemine sebep olmayacak,
 - kopyalanarak dağıtılmadan sistemde sadece tek bir örneği olacak şekilde,
 - bağlamsal ve kültürelidir.

Ama!



- Ama anlaşılabilirliği azaltan ve yanlış sevkeden, bağlam değişimi içeren, örneğin aynı satırda farklı nesneler üzerindeki çağrıları zincirleyen ifadeler o kadar masum değildir.

```
shoppingCard.getItemList().get(0).recommendedBy().size();
```



ShoppingCard



List<Item>



Item



List<Customer>

- Bu ve benzeri şekiller çok daha anlaşılirdir, muhtemel tekrar metod çağrılarını önler ve gerekirse `null` kontrolleri yapılabilir.

```
Item item = shoppingCard.getItemList().get(0);
int recommendedBySize = item.recommendedBy().size();
```



Metot

Metot Tipleri - I



- **SRP** açısından değerlendirmede, metotların ne türden iş yaptıkları önemlidir.
- Metotlar genelde dört tipe ayrılabilir:
 - **Kurucular (constructors)**: Nesne oluşturma ve ilk haline getirme (initialization).
 - **Set/Get (setters/getters)**: Bilgi alıp verme.
 - **toString () /ToString () , equals () /Equals ()** vb.
`java.lang.Object / System.Object` vb. nesnelerin metotları.
 - İş yapan metotlar.

Kurucular (Constructors) - I



- Kurucular çok bilinen, standart bir iş yaparlar: nesne yaratmak.
- Bu açıdan sorumlulukları daima bir tanedir, son derece odaklı metotlardır.
 - Kurucularda nesne oluşturma dışında başka bir şey yapmamak önemlidir.
 - Kurucularda sıkılıkla `init()` vb. isimlerde başlatma (initialization) yapan metodların çağrıldığı görülür.
 - Bu tür metodlar yerine destekleniyorsa nesne başlatma bloklarını tercih daha rahat bir çözüm olabilir.

Kurucular (Constructors) - II



- Kurucular nesnenin karmaşıklığına göre çok parametre alabilirler.
- Parametrelerin de oluşturulması gereği hallerde bu durum istemciler için bir problemidir.
 - Bu amaçla yaratımsal kalıplar (creational patterns) kullanılmalı ve istemcinin hem nesne yaratmasından hem de gerekli parametreleri oluşturup geçmesinden olabildiğince kaçınmak gereklidir.
 - Öte taraftan fazla sayıda parametre, kötü tasarlanmış bir sınıfın göstergesi olabilir.

Kurucular (Constructors) - III



- Örneğin aşağıdaki kurucular daha az parametre alacak hale getirilebilir ve statik factory metodlarıyla değiştirilebilir.

```
public Email(long id, String userId, String emailAddress, MailType type,  
           MailStatus status, String subject, boolean deliveryGuarranteed,  
           Date creationTime, Date tryUntil){...}
```

```
public Email(long id, String userId, String emailAddress, MailType type,  
           MailStatus status, String subject, boolean deliveryGuarranteed,  
           Date creationTime, Date tryUntil, boolean validationNeeded){...}
```

```
public static Email createNewUrgentDeliveryGuarranteedEmail(long id, String userId,  
                String emailAddress, String subject, Date tryUntil){...}
```

```
public static Email createNewUrgentDeliveryGuarranteedToBeValidatedEmail(long id,  
                String userId, String emailAddress, String subject, Date tryUntil){...}
```

set/get Metotları



- **set/get** metotları da çok odaklı metotlardır, hemen daima tek bir sorumluluğu yerine getirirler.
- Bu metotlarda bazen doğrulama (validation), çevirim (conversion) ve karşı gelen bir üye değişken (member variable) olmadığından hesaplama vs. türden işler yapılsa da hepsi **set/get** metodunun tabiatına uygun işlerdir.

java.lang.Object/System.Object Metotları



- `java.lang.Object/System.Object`'den devralınan metotlar da yüksek birlikteliğe sahip olur çünkü hem sorumlulukları bellidir hem uygun arayüzleri vardır.

Metot Tipleri - II



- İş yapan metotlar da kendi aralarında ikiye ayrılırlar.
- Bazı metotlar bölünemez, atomik bir işi yaparlar; bir sürecin bir adımını, bir hesabı, atomik bir bir iş kuralını vs. yerine getirirler.
- Bu metotların yaptıkları işler daha alt parçalara bölünemez;
 - **İşçi/atomik metotlar (worker/atomic methods)**

Metot Tipleri - III



- Bazı metotlar ise süreci yerine getirirler, dolayısıyla birden fazla atomik metodun çalışmasını koordine ederler.
- **Yöneten/yönetsel/koordinative metotlar (coordinative methods)**
- Bu şekilde tek iş yapan, atomik metodlardan daha üst katmanlarda hala tek iş, süreçsel tabiatta tek iş, yapan metotlara doğru gidilir.

Account & AccountService



- `changeManage()` metodu ne atomik ne de süreçsel bir metottur, birden fazla atomik işi bir arada yapmaktadır.

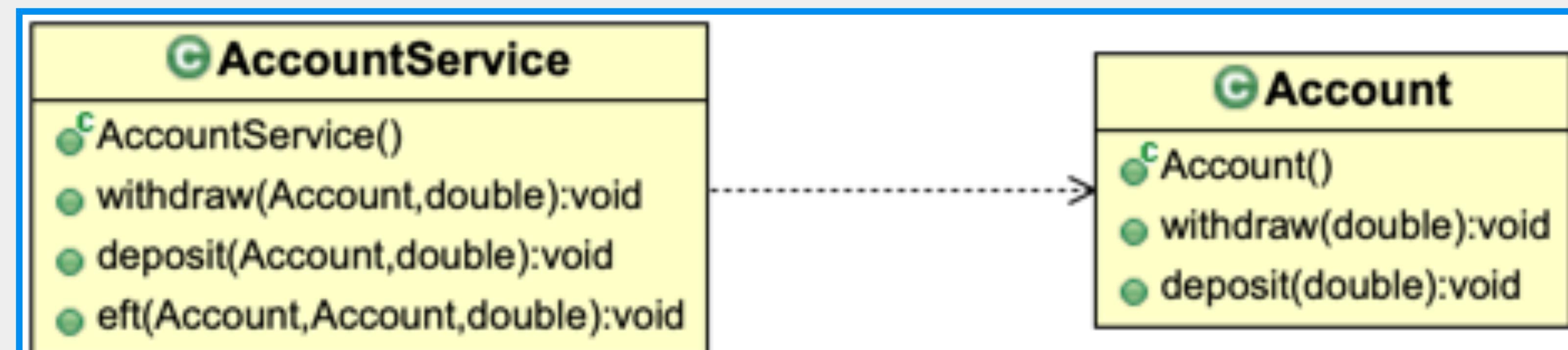
```
public void changeBalance(String action, double amount)
    throws InsufficientBalanceException, NegativeAmountException{
if(amount < 0)
    throw new NegativeAmountException(amount);

if(action.equals("Deposit"))
    balance += amount;
else if(action.equals("Withdraw")){
    if(balance ≥ amount)
        balance -= amount;
    else
        throw new InsufficientBalanceException(action, balance, amount);
}
log.info(action + " : " + amount + " for account id: " + id);
}
```

Account & AccountService



- **AccountService** üzerindeki metodlar yönetsel bir metottur, değer kontrolü (validation) ve **Account** nesnesi ile ilgili atomik işleri bir arada yerine getirmektedir.



ch02.anemic.account2

Account & AccountService



```
public class AccountService {  
    private static final double EFT_CHARGE = 3;  
    private AmountValidator validator = new AmountValidator();  
  
    public void withdraw(Account account, double amount) throws InsufficientBalanceException, NegativeAmountException{  
        validator.validate(amount);  
        account.withdraw(amount);  
    }  
  
    public void deposit(Account account, double amount) throws InsufficientBalanceException, NegativeAmountException{  
        validator.validate(amount);  
        account.deposit(amount);  
    }  
  
    public void eft(Account sourceAccount, Account targetAccount, double amount) throws InsufficientBalanceException, NegativeAmountException{  
        validator.validate(amount);  
        sourceAccount.withdraw(amount);  
        sourceAccount.withdraw(EFT_CHARGE);  
        targetAccount.deposit(amount);  
    }  
}
```

ch02.anemic.account2

```
public class Account {  
    private String iban;  
    private double balance;  
  
    public void withdraw(double amount) throws InsufficientBalanceException{  
        if (balance ≥ amount) {  
            balance -= amount;  
        } else  
            throw new InsufficientBalanceException(balance, amount);  
    }  
  
    public void deposit(double amount) throws InsufficientBalanceException{  
        balance += amount;  
    }  
}
```

Metot Tipleri - IV



- Sistem karmaşıklığı arttıkça, koordinative/yönetsel metodların katmanlaşması gerektiği düşünülebilir.
- Yani koordinative/yönetsel metodlar birbirlerini çağırabilir.
- Bu şekilde koordinative/yönetsel metodlar arasında da bir hiyerarşi oluşur.
- Tabi olarak işleri koordine eden metodlar, uygulama katmanında transaction, güvenlik, kaynak yönetimi vb. uygulama mantığını koordine eden metodlar tarafından kullanılır.

AccountAppService & AccountService



ch02.anemic.account2

```
public class AccountAppService {  
  
    private AccountService as = new AccountService();  
  
    public void withdraw(Account account, double amount) throws InsufficientBalanceException, NegativeAmountException{  
        as.withdraw(amount);  
    }  
  
    public void deposit(Account account, double amount) throws InsufficientBalanceException, NegativeAmountException{  
        as.deposit(amount);  
    }  
  
    public void eft(Account sourceAccount, Account targetAccount, double amount) throws InsufficientBalanceException, NegativeAmountException{  
        as.eft(sourceAccount, targetAccount, amount)  
    }  
}
```

```
public class AccountService {  
  
    ...  
  
    public void withdraw(Account account, double amount) throws InsufficientBalanceException, NegativeAmountException{  
        ...  
    }  
  
    public void deposit(Account account, double amount) throws InsufficientBalanceException, NegativeAmountException{  
        ...  
    }  
  
    public void eft(Account sourceAccount, Account targetAccount, double amount) throws InsufficientBalanceException, NegativeAmountException{  
        ...  
    }  
}
```

SRP: Metot - I



- **SRP** gereği olarak bir metotta sadece tek bir sorumluluk, tek bir iş yerine getirilmelidir.
- Bu tek bir iş
 - atomik bir iş kuralı, dallı (`if else if` vb.) iş kurallarında sadece bir dal,
 - tek bir kısıt,
 - atomik bir algoritma, bir hesaplama ya da
 - karmaşık bir sürecin tek bir adımıdır.

SRP: Metot - II



- Metot, tekrar kullanımının en temel ögesidir.
 - **Tekrar kullanılma ihtimali olan her blok ya da cümle vb. kod parçası, ne kadar kısa olursa olsun, ayrı bir metot olmalıdır.**
 - Aksi taktirde aynı kod parçası **copy-paste** ile farklı yerlere dağıılır.
 - Tekrar kullanımı gereken hatta ihtimali olan kod parçaları refactoring sürecinde **cut-paste** ile metot haline getirilmelidir.
 - Bu şekilde miras (inheritance) ile o kod parçalarının genişletilmesi de söz konusu olur.

Private Metotlar



- Tekrar kullanımı gerekmeyen bir kod parçasının metot haline getirilmesi, ancak **private** metotlar için söz konusudur.
- Bu durumda asıl metodun büyümesi önlenmiş ve daha rahat okunabilir halde kalması sağlanmıştır.
- Ayrıca ileride tekrar kullanım söz konusu olduğunda **private** bir metot hızlıca ulaşılabilir hale getirilebilir.

SRP: Metot - III



- **SRP**'ye uygun metodun şekilsel olarak üç kriteri vardır, metotlar ortalama olarak,
 - 10 - 15 satırı geçmemeli,
 - olabildiğince az parametre ($<= 3$) almalı,
 - parametre sayısı arttıkça bağımlılığın veri hatta içerik bağımlılığına dönüşme ihtimalini göz önüne alın,
 - olabildiğince az karar vermeli (cyclomatic complexity),
 - olabildiğince az sıra dışı durum (exception) fırlatmalıdır.

SRP: Metot - III



- **SRP**'ye uygun metodun şekilsel olarak üç kriteri vardır, metotlar ortalama olarak,
 - 10 - 15 satırı geçmemeli,
 - olabildiğince az parametre ($<= 3$) almalı,
 - parametre sayısı arttıkça bağımlılığın veri hatta içerik bağımlılığına dönüşme ihtimalini göz önüne alın,
 - olabildiğince az karar vermeli (cyclomatic complexity),
 - olabildiğince az sıra dışı durum (exception) fırlatmalıdır.



- Metot, yüksek birlikteli, odaklı olması için kısa yazılmaz, odaklı yazıldığından zaten kısa olur.
- Odaklı metot, atomiktir, daha alt parçaları bölünemez.
- Verilen kriterlerin ortalamada geçerli olduğuna dikkat edin!
 - Metot bazen 1 satır olur bazen 10-15 satır olur ama muhtemelen 250 ya da 800 satır olmaz!
 - Ama muhtemelen parametre, karar ve sıra dışı durum sayısı daha sıkı, esnetilmesi zor olan kriterlerdir.

Az Parametreli Metot



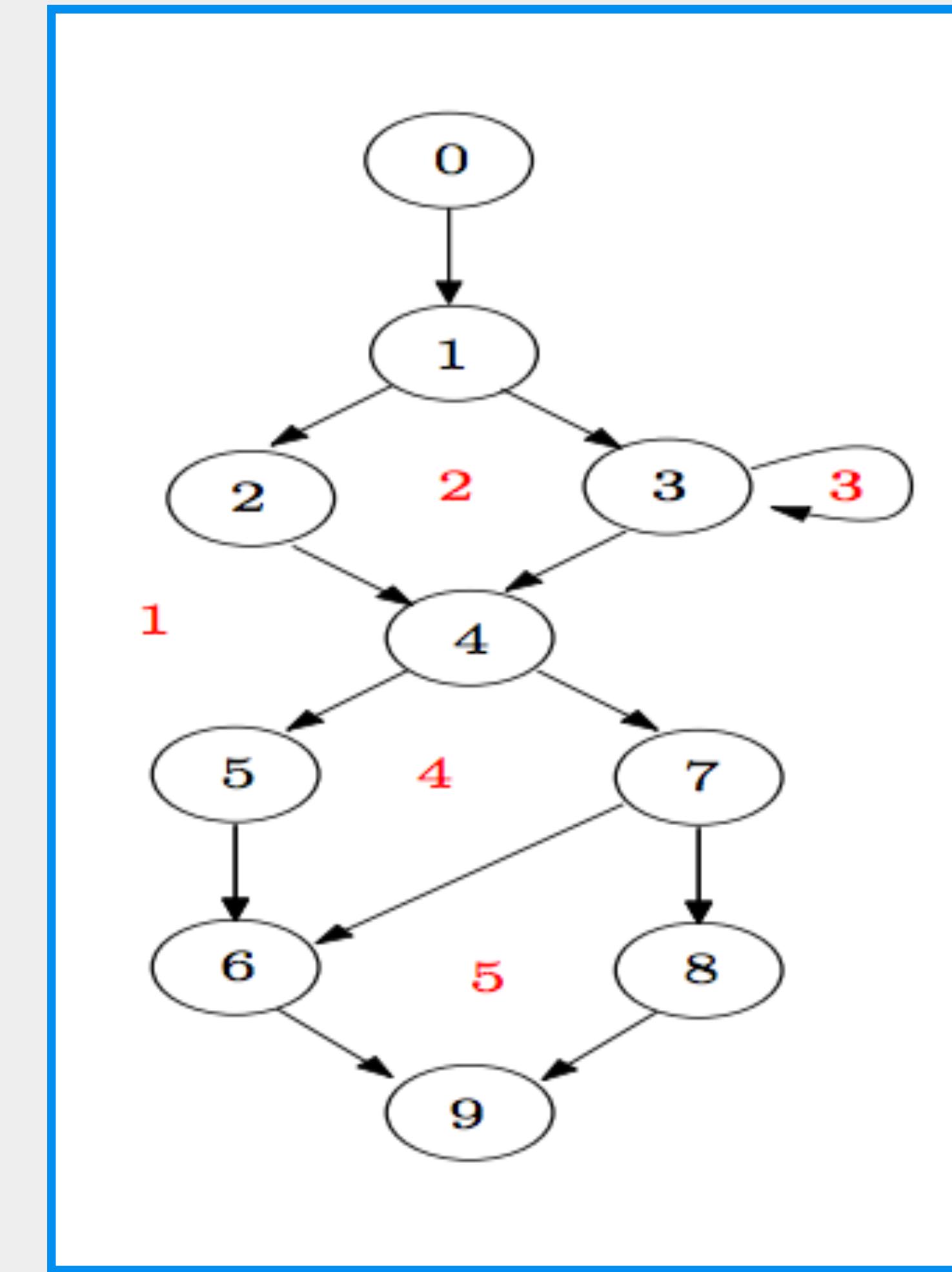
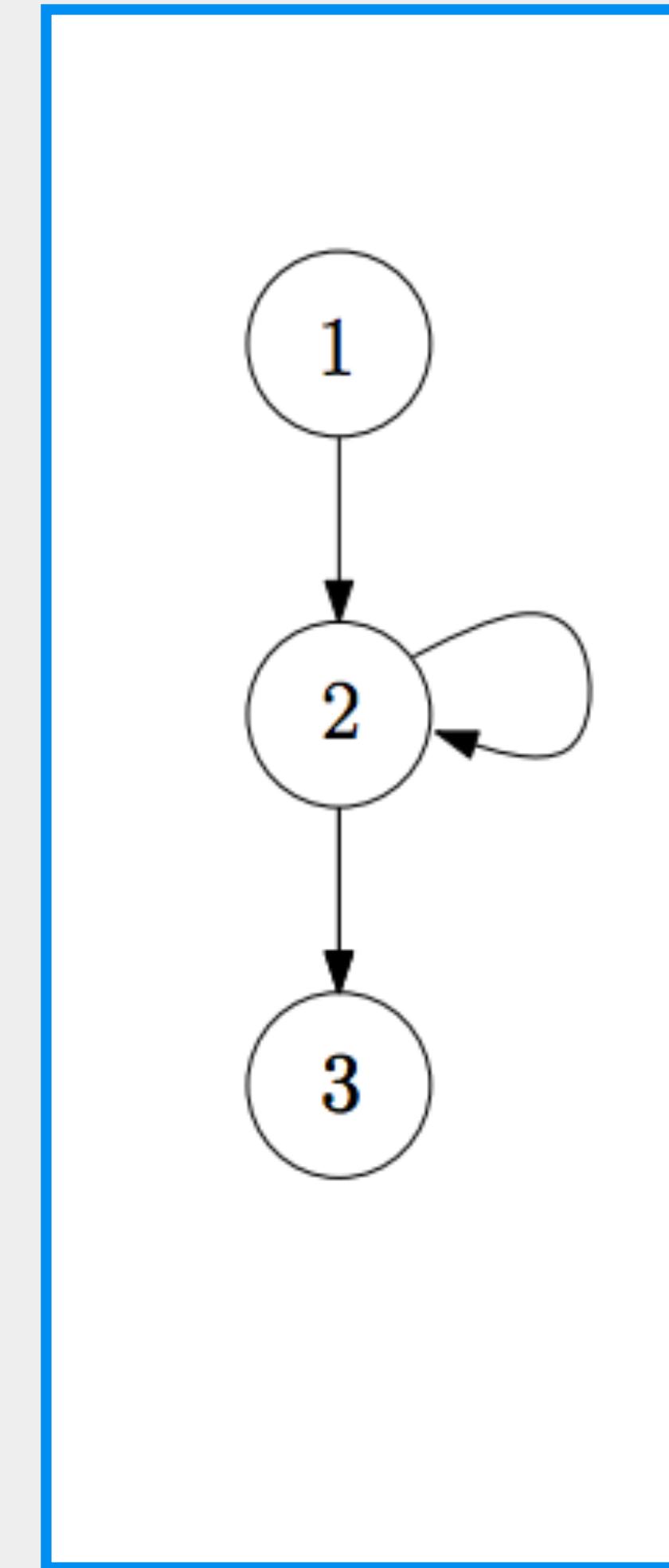
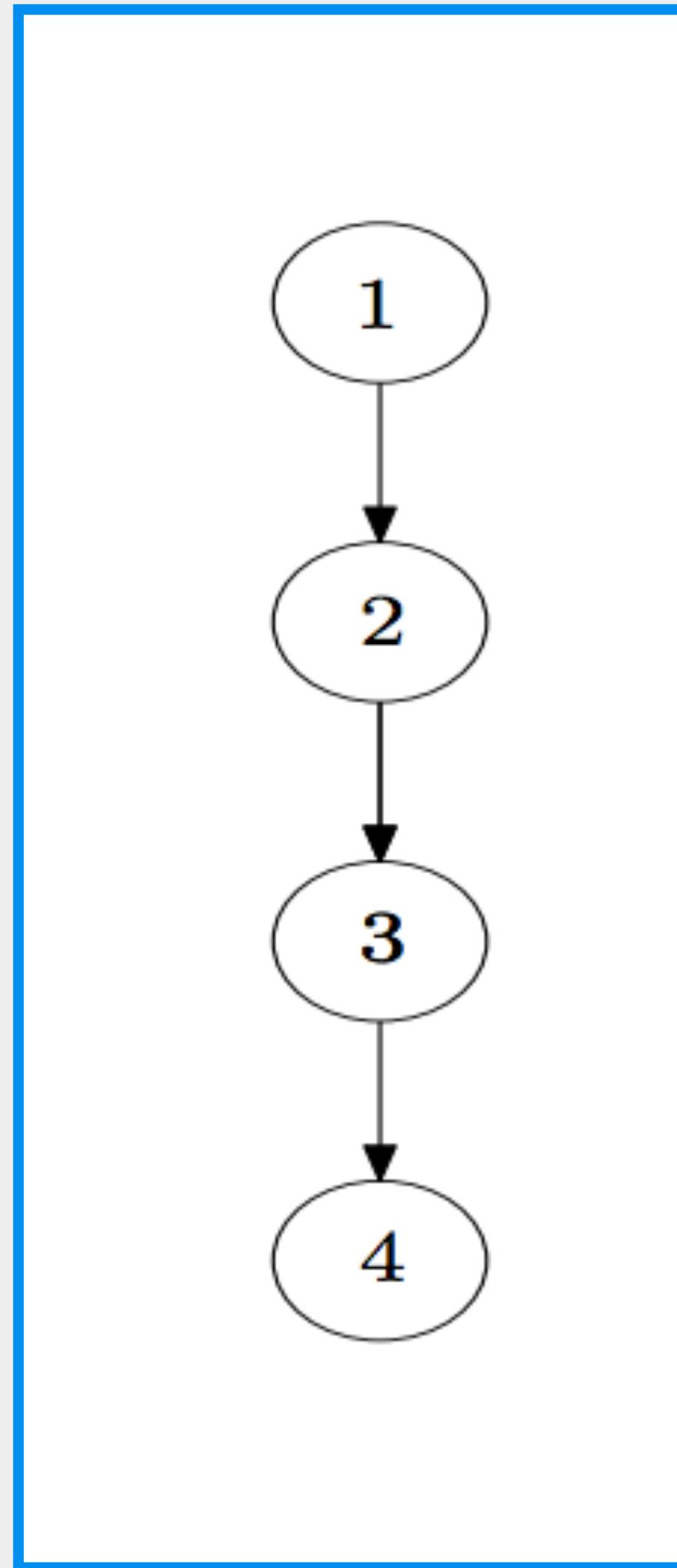
- Parametre sayısı arttıkça metodun odağı düşme eğilimine girer çünkü muhtemelen daha çok iş yapar.
- Eğer parametreler ilkel (primitive) tiplerdense, metodun worker metot olma ihtimali artar.
- Yönetsel metotlar ilkel yerine referans parametre alma eğilimindedirler.
- Eğer parametreler referans tip ise, ikiyi geçmemelidir, aksi taktirde odak sorunu ortaya çıkabilir.

Karar Karmasıklığı - I



- **Karar karmaşıklığı (cyclomatic compexity)** McCabe tarafından 1976'da geliştirilmiş bir ölçütür.
- Bir koddaki bağımsız, lineer akışların sayısına denir.
- Lineerliği bozan kapalı alanlardır ki bunlar da kararlar ile oluşur.
- Odaklı metot olabildiğince az karar verir.
- Çünkü çok karar, birlikteliği düşürür, bağımlılığı arttırmır.

Karar Karmasıklığı - II



Sıra Dışı Durum Fırlatan Metot



- Bir metot çok iş yapıyorsa çok sıra dışı durum (exception) fırlatır!
- Bunu en temel istisnası kendisi oluşturmadığı halde, sıra dışı durum fırlatan metodun çağrı zincirinde (call chain/stack) olup sıra dışı durum nesnesini bir üst bağlama yükseltmektir (raising).
- Java gibi bazı dillerde metodun fırlattığı sıra dışı durumları arayüzünde belirtmesi gereklidir.
- Bunlar dilden dile değişmekle birlikte metodun olabildiğince az sıra dışı durum oluşturup fırlatması **SRP**'ye uygunluğu açısından önemlidir.

Metotları SRP'ye Uygun Hale Getirme



- **SRP** olmayan metotlar bölünmelidir:
 - Yönetsel kısımlar ile iş yapan kısımlar ayrı metotlara bölünmeli, belki yönetsel metotlar farklı bir sınıfa aktarılmalıdır,
 - Birden fazla atomik işi bir araya getirmiş metotlar bölünerek tamamen atomik hale getirilmelidir.
 - Atomik oldukları halde birden fazla bloğa sahip olan metotların blokları **private** metotlar olarak ayrılabilir,
 - **private** metotlarının ileride ulaşılabilir hale getirilmesi çok daha kolay olur!

Uygulama

- **assignIdNo** metodunu **SRP** açısından değerlendirin.
- Bölünmesine gerek var mı?

```
public void assignIdNo(CardInfo cardInfo, Form registrationForm,
    WSResponse verifyOtpWSResponse){

    // Burada CardInfo'nun Idsi atanır. Id olarak eğer musteri TC vatandası ise
    // WS'den ya da fromdan gelen tckn atanır. Eğer müşteri TC vatandası
    // değilse Id olarak ya WS'den ya da formdan gelen customer no atanır.

    String idNo = null;
    Boolean isTcCitizen = cardInfo.getTcCitizen();
    String tcknFromForm = registrationForm.getTckn();
    String tcknFromWS = verifyOtpWSResponse.getTckn();
    boolean nullOrEmptyTcknFromWS = formUtils.isNullOrEmptyString(tcknFromWS);
    String custNoFromWS = verifyOtpWSResponse.getCustNo();
    boolean nullOrEmptyCustNoFromWS = formUtils.isNullOrEmptyString(custNoFromWS);

    if(isTcCitizen){
        if(nullOrEmptyTcknFromWS)
            idNo = tcknFromForm;
        else
            idNo = tcknFromWS;
    }
    else{
        if(nullOrEmptyCustNoFromWS)
            idNo = tcknFromForm;
        else
            idNo = custNoFromWS
    }
    cardInfo.setId(idNo);
}
```



Uygulama

Uygulama



- `login()` metodunu **SRP**ye uygun olarak nasıl gerçekleştirilebileceğini düşünün.
- Verilen gerçekleştirmeyi değerlendirin.

```
public void login(String tckn, String password) throws  
    NoSuchCustomerException, CustomerAlreadyLoggedException,  
    WrongCustomerCredentialsException,  
    MaxNumberOfFailedLoggingAttemptExceededExpection,  
    CustomerLockedException, ImproperCustomerCredentialsException,  
    NoProperPasswordException;
```

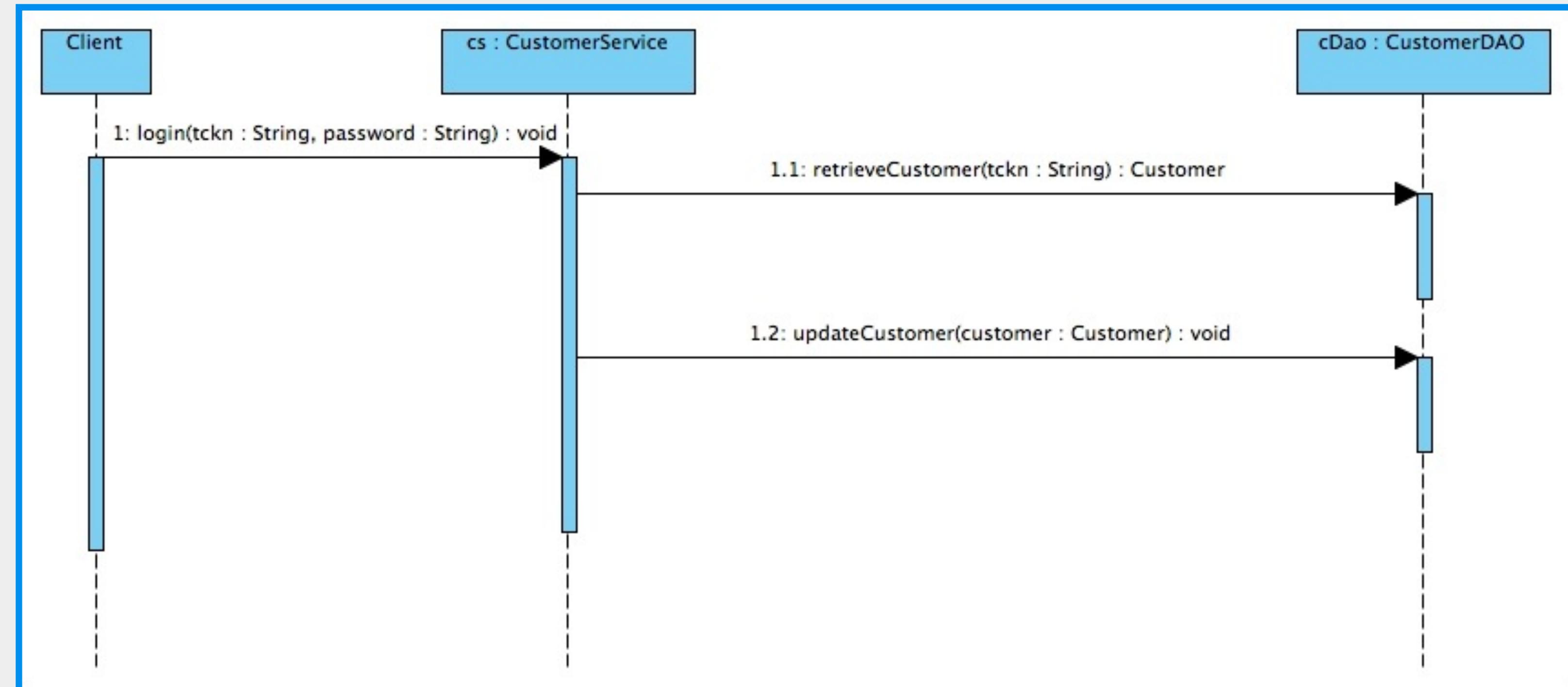
ch03.srp.customerService.problem.CustomerService

- login metodu prosedürel birlikteğe sahiptir.
- Birden fazla işi sırayla yapıyor.

```
public void login(String tckn, String password) throws NoSuchCustomerException,
CustomerLockedException, CustomerAlreadyLoggedException, WrongCustomerCredentialsException,
MaxNumberOffFailedLoggingAttemptExceededException, ImproperCustomerCredentialsException,
NoProperPasswordException {

    validateTckn(tckn);
    validatePassword(password);
    Customer customer = customerDao.retrieveCustomer(tckn);

    if (customer.getPassword().equals(password) & !customer.isLocked() & !customer.isLoggedIn()) {
        customer.setLoggedIn(true);
        logger.info(customer.getFirstName() + " " + customer.getLastName() + " logs in.");
        customerDao.updateCustomer(customer);
        currentCustomer = customer;
    } else if (customer.isLoggedIn()) {
        throw new CustomerAlreadyLoggedException("Customer is already logged in. Please first log out.");
    } else if (customer.isLocked()) {
        throw new CustomerLockedException("Customer is locked. Please consult your admin.");
    } else if (!customer.getPassword().equals(password)) {
        loginAttemptCount++;
        int limit = Integer.parseInt(ATMProperties.getProperty("customer.maxFailedLoginAttempt"));
        if (loginAttemptCount == limit) {
            customer.setLocked(true);
            customerDao.updateCustomer(customer);
            loginAttemptCount = 0;
            throw new MaxNumberOffFailedLoggingAttemptExceededException(
                "Max number of login attempt reached: " + loginAttemptCount);
        }
    } else
        throw new WrongCustomerCredentialsException("TCKN/password is wrong.");
}
}
```



Uygulama



- `login()` metodunda şu işlerin hepsi metotta halledilmektedir:
 1. `Customer`'ın `CustomerDao` ile getirtilmesi,
 2. `Customer`'ın passwordü ile geçen passwordün aynı olduğu ve aynı zamanda da kilitli ve hali hazırda login olmadığıının belirlenmesi halinde sisteme login olması ve veri tabanında durumunun güncellenmesi,
 3. `Customer`'ın hali hazırda login olmuş ise sıra dışı durum fırlatılması,
 4. `Customer`'ın hali hazırda kilitli ise sıra dışı durum fırlatılması,
 5. `Customer`'ın passwordü yanlış girilmişse sıra dışı durum fırlatılması ve `badLoginAttemptCount`' un arttırılması. `badLoginAttemptCount` limite ulaşmışsa sıra dışı durumun fırlatılması.



```
public void login(String tckn, String password) throws NoSuchCustomerException,  
    CustomerLockedException, CustomerAlreadyLoggedException,  
    WrongCustomerCredentialsException,  
    MaxNumberOfFailedLoggingAttemptExceededException {  
  
    Customer customer = customerDao.retrieveCustomer(tckn);  
  
    if (customer.getPassword().equals(password) &  
        !customer.isLocked() & !customer.isLoggedIn()) {  
        // Database is updated when a customer logs in.  
        customer.setLoggedIn(true);  
        if (customerDao.updateCustomer(customer))  
            currentCustomer = customer;  
        loginAttemptCount = 0;  
    }  
  
    else if (customer.isLoggedIn()) {  
        throw new CustomerAlreadyLoggedException("Customer is already logged in.  
            Please first log out.");  
    }  
  
    else if (customer.isLocked()) {  
        throw new CustomerLockedException("Customer is locked.  
            Please consult your admin.");  
    }  
  
    else if (!customer.getPassword().equals(password)) {  
        badLoginAttemptCount++;  
        if (badLoginAttemptCount == Integer.parseInt(ATMProperties.getProperty(  
                "customer.maxFailedLoginAttempt"))) {  
            customer.setLocked(true);  
            customerDao.updateCustomer(customer);  
            throw new MaxNumberOfFailedLoggingAttemptExceededException("Max number of  
                login attempt reached: "  
                + badLoginAttemptCount);  
        }  
        throw new WrongCustomerCredentialsException("TCKN/password is wrong.");  
    }  
}
```

1

2

3

4

5



```
public void checkIfCustomerAlreadyLoggedIn(Customer customer) throws CustomerAlreadyLoggedInException {  
    if (customer.isLoggedIn()) {  
        throw new CustomerAlreadyLoggedInException("Customer is already logged in.  
            Please first log out.");  
    }  
}
```

3

```
public void checkIfCustomerLocked(Customer customer) throws CustomerLockedException {  
    if (customer.isLocked()) {  
        throw new CustomerLockedException("Customer is locked. Please consult your admin.");  
    }  
}
```

4

```
public void checkCustomerPassword(Customer customer, String password) throws  
    MaxNumberOffFailedLoggingAttemptExceededexception, WrongCustomerCredentialsException {  
    if (!customer.getPassword().equals(password)) {  
        loginAttemptCount++;  
        checkLoginAttempCount(customer);  
        throw new WrongCustomerCredentialsException("Wrong password!");  
    }  
}
```

5

```
public void checkLoginAttempCount(Customer customer) throws  
    MaxNumberOffFailedLoggingAttemptExceededexception{  
    if (loginAttemptCount = Integer.parseInt(ATMProperties.getProperty(  
                    "customer.maxFailedLoginAttempt"))){  
        lockCustomer(customer);  
    }  
}
```

5

```
public void lockCustomer(Customer customer) throws  
MaxNumberOffFailedLoggingAttemptExceededexception{  
    customer.setLocked(true);  
    throw new MaxNumberOffFailedLoggingAttemptExceededexception("Max number of login  
        attempt reached: "  
        + loginAttemptCount);  
}
```

5

- Atomik iş yapan metotlar



```
public void login(String tckn, String password) throws  
    NoSuchCustomerException, CustomerAlreadyLoggedException,  
    WrongCustomerCredentialsException,  
    MaxNumberOfFailedLoggingAttemptExceededException,  
    CustomerLockedException{  
    validateTckn(tckn);  
    validatePassword(password);  
    Customer customer = customerDao.retrieveCustomer(tckn);  
    loginCustomer(customer, password);  
}
```

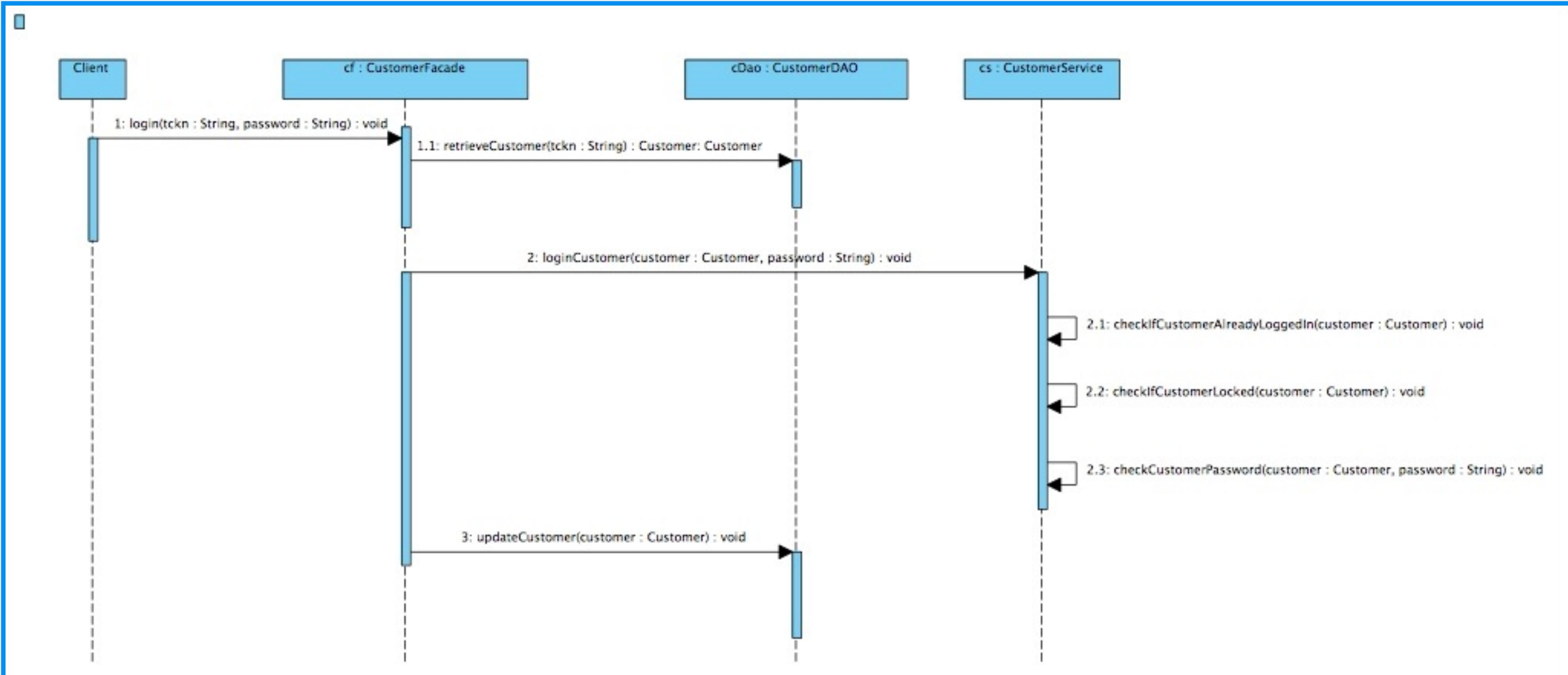
1

- Yönetsel metotlar.

```
private void loginCustomer(Customer customer, String password)  
throws  
    CustomerAlreadyLoggedException,  
    WrongCustomerCredentialsException,  
    MaxNumberOfFailedLoggingAttemptExceededException,  
    CustomerLockedException{  
  
    boolean login = false;  
    checkIfCustomerAlreadyLoggedIn(customer);  
    checkIfCustomerLocked(customer);  
    checkCustomerPassword(customer, password);  
    customerDao.updateCustomer(customer);  
}
```

2

- Atomik işler ile yönetsel işler ayrılmış durumda.



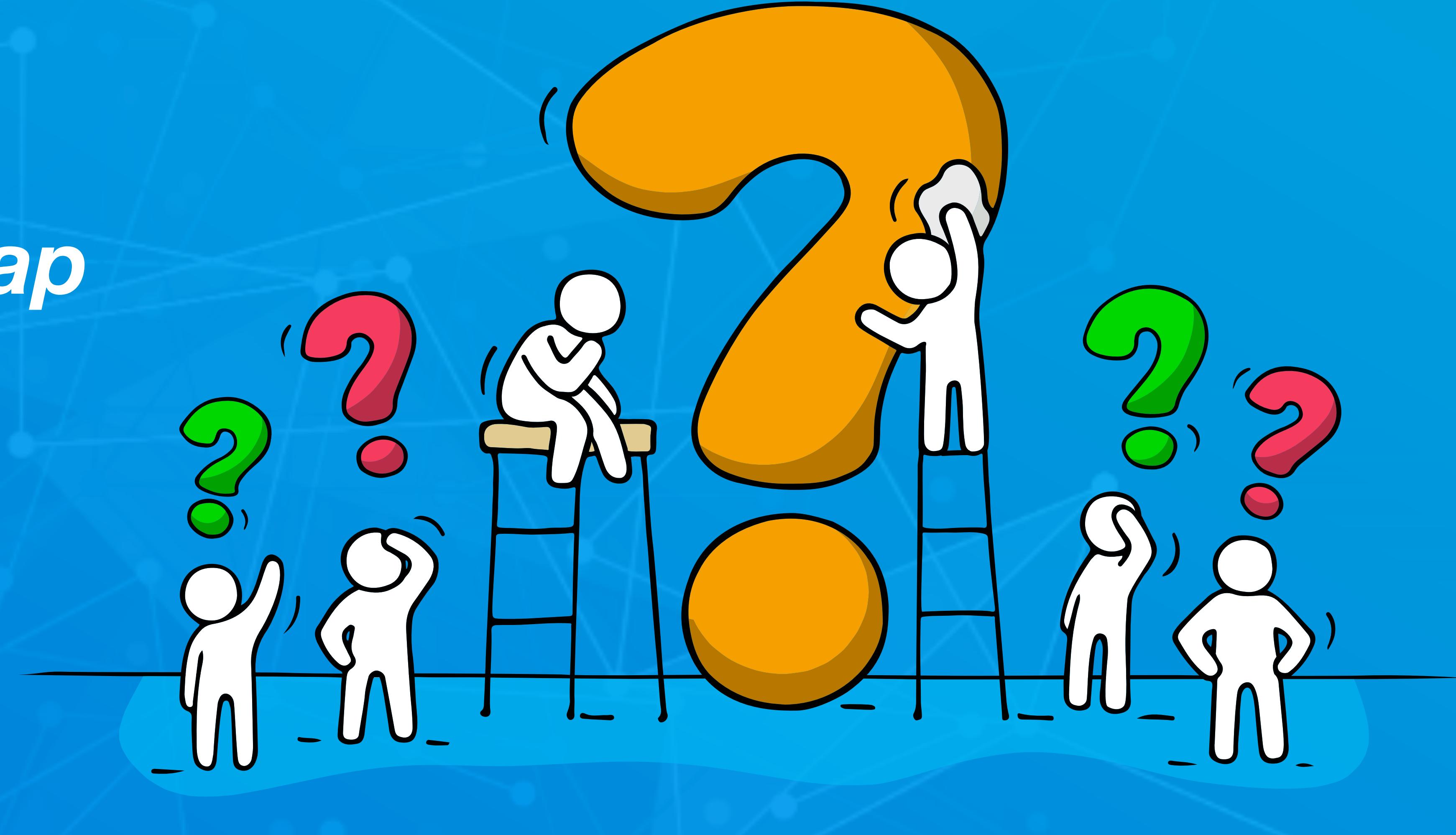


- Gelinin noktada, ilk halde bir tek metodun içinde yer alan bloklar, tekrar kullanılabilecek şekilde metotlara dönüşmüş durumdadır.
- Böylece işçi metotlar sadece bir şey yapar hale gelmiştir.
- Orijinal `login (String username, String password)` metodu ise yönetsel metottur, diğer metotları çağrıarak `login` sürecini yönetmektedir.



- `login(String username, String password)` metodu en üstteki, **Façade** nesnesindeki yönetici metottur,
- Çağırıldığı metotlardan `login(Customer customer, String password)` metodu ise **Servis** nesnesinde daha aşağı seviyedeki bir yönetsel metottur.
- Ama hepsi iş mantığını içeren iş metotlarıdır.

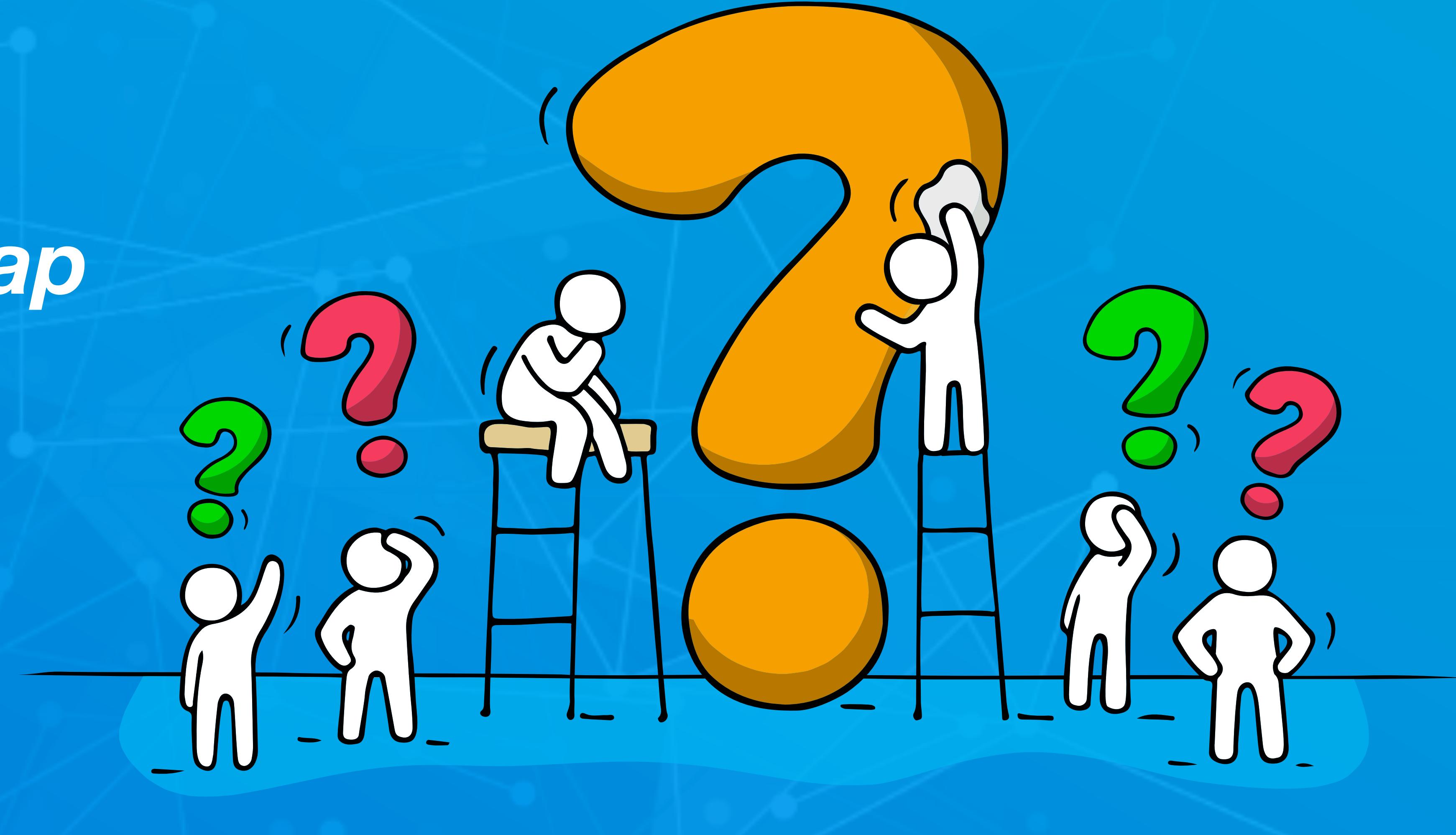
Soru ve Cevap Zamani!





Sinif

Soru ve Cevap Zamani!





- Sınıflar sadece tek bir kavramla ilgili sorumlulukları bir araya getirmelidir.
- Sınıfların karmaşıklığı sahip olduğu sorumluluklara bağlıdır.
 - Çok sorumluluk, çok metot, karmaşık arayüz, dolayısıyla da karmaşık sınıf demektir.
 - Durumunun (state) karmaşıklığı da buna bağlıdır.
 - Çok nesne değişkeni, karmaşık durum, karmaşık sınıf demektir.

Sınıfları SRP'ye Uygun Hale Getirme - I



- **SRP** olmayan sınıflar bölünmelidir:
 - Daha önce ele alınan birliktelik karşı kalıpları (cohesion anti-pattern), mixed-instance, mixed-domain ve mixed-role durumlarını ortadan kaldıracak şekilde sınıflar bölünmelidir.
 - Atomik işçi metotlar ile yönetsel metotlar muhtemelen farklı sınıflarda olmalıdır çünkü bu sınıfların rolleri farklıdır, mixed-role problemine sahip olabilirler.

Sınıfları SRP'ye Uygun Hale Getirme - II



- İlk etapta aynı soyutlamayla ilgili sorumluluklar bir araya getirildiği halde, ilerleyen zamanlarda sorumlulukların sayısının artması, yapıyı büyük ve karmaşık hale getirebilir.
- Bu durumda zamanla refactoring ile soyutlamanın alt soyutlamalara bölünerek yapının makul karmaşıklıkta kalması sağlanmalıdır.
- Dolayısıyla yeni sınıflar ve metodlar sadece yeni ihtiyaçlardan dolayı değil, var olanların bölünmesinden dolayı da oluşturulmalıdır.

Sınıfları SRP'ye Uygun Hale Getirme - III



- Şekil açısından bakıldığında sınıfın metodlarının sayısı 15-20'yi geçiyorsa, muhtemelen o sınıf, birden fazla sınıfa bölünmelidir.
- Çoğu zaman sınıfların, metod sayıları bu sayılarla gelmeden bölünmeleri gereklidir, çünkü zaten farklı şeyler yapıyorlardır.
- **Interface Segregation** gibi diğer prensipler ve tasarım kalıpları, bir sınıfı bölmeye yardımcı olur.

Birden Fazla Üst Tipi Olan Sınıflar - I



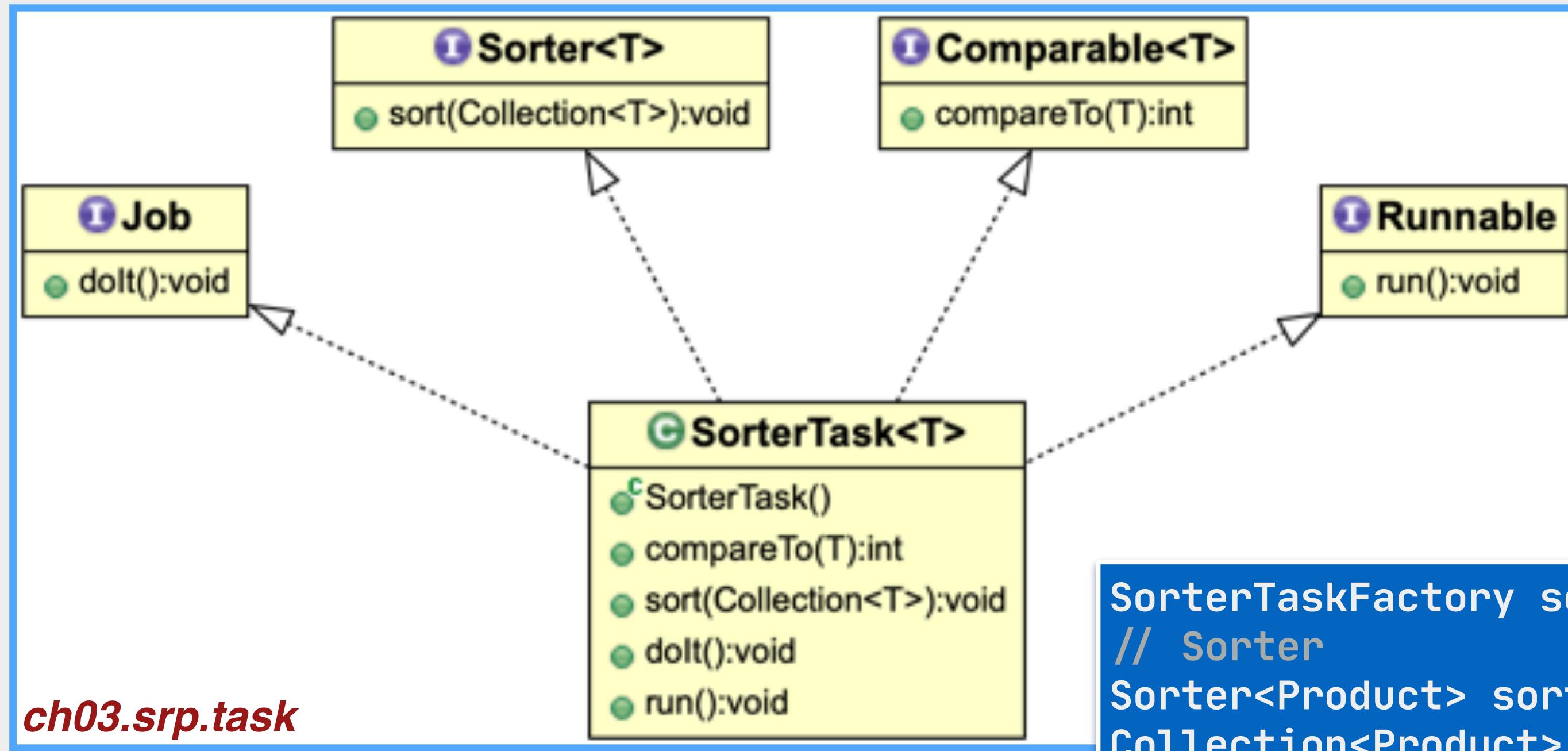
- Bazen sınıfların birden fazla arayüzü yerine getirerek, pek çok role sahip oldukları görülür.
- Özellikle yetkinlik kazandırmak amacıyla bir API'nin parçası olan arayüzleri yerine getiren ama aynı zamanda bir rolü olan nesnelerde bu durum yaygındır.
- Böyle sınıflar **bileşik/küme sınıf (composite/aggregate)** olarak adlandırılır.
 - Bu türden sınıfların arayüzlerinin şişman (fat) ya da kirli (polluted) olduğundan bahsedilerek bunun bir kötü koku (code smell) yayıldığı da iddia edilebilir.

Birden Fazla Üst Tipi Olan Sınıflar - II



- Burada amaç, farklı yetkinlikleri ayrı yapılar olarak soyutlayıp sistemde tekrar kullanımı (reusability) artırrarak daha büyük bir fayda elde etmektir.
- Öte yandan ISP'de de ele alacağımız gibi istemciler bu türden nesneleri olabildiğince üst tipleri cinsinden göreceklerinden istemciler açısından gereksiz olan davranışları saklanacak ve nesne hala ince bir arayüzle temsil edilecektir.

Birden Fazla Üst Tipi Olan Sınıflar - II

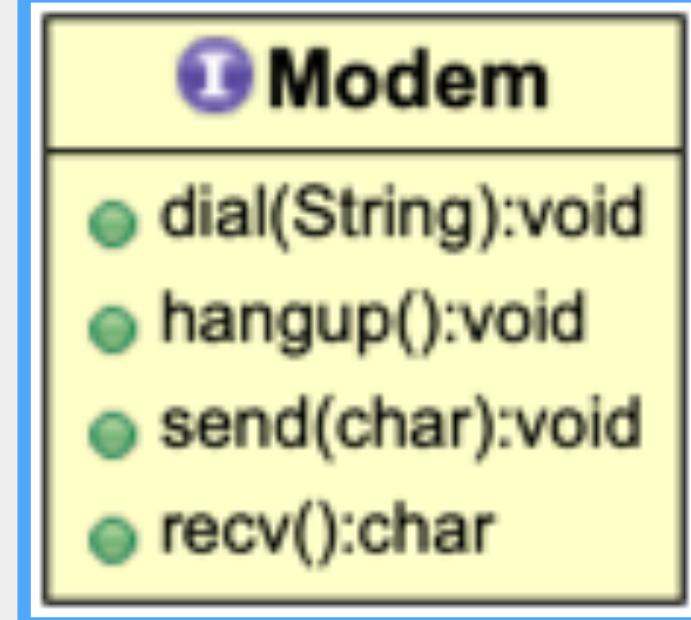


```
SorterTaskFactory sorterTaskFactory = new SorterTaskFactory();
// Sorter
Sorter<Product> sorter = sorterTaskFactory.create();
Collection<Product> list = null;
// ...
sorter.sort(list);
// Thread
Thread sorterThread = (Thread) sorter;
sorterThread.run();
// Comparable
Comparable comparableSorter2 = (Comparable)sorterTaskFactory.create();
Comparable comparableSorter1 = (Comparable) sorter;
comparableSorter1.compareTo(comparableSorter2);
```

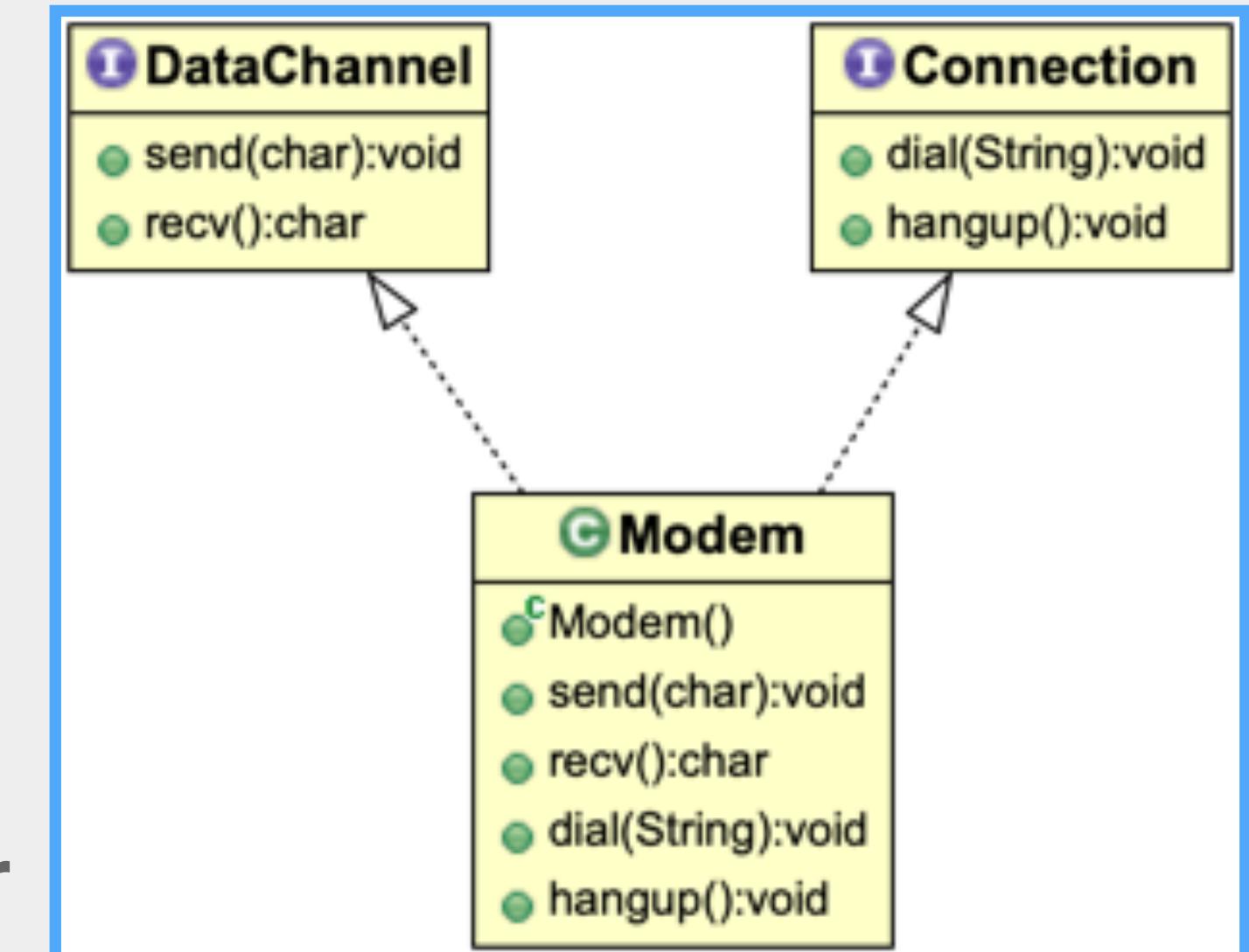
Modem



- Modem arayüzünün iki sorumluluğu vardır.
 - Dolayısıyla Modem arayüzü bölünmeli ve ancak birden fazla arayüzün gerçeklestirmesi olarak ifade edilmelidir.
- Bu durumda birden fazla rolü bir araya getirmesinden dolayı Modem'in birlikteliğinin düştüğünden bahsedilebilir.
- Ama unutmayın ki Modem de kime bir yapı olarak bir başka roldür!



ch03.srp.paper.comm1.Modem

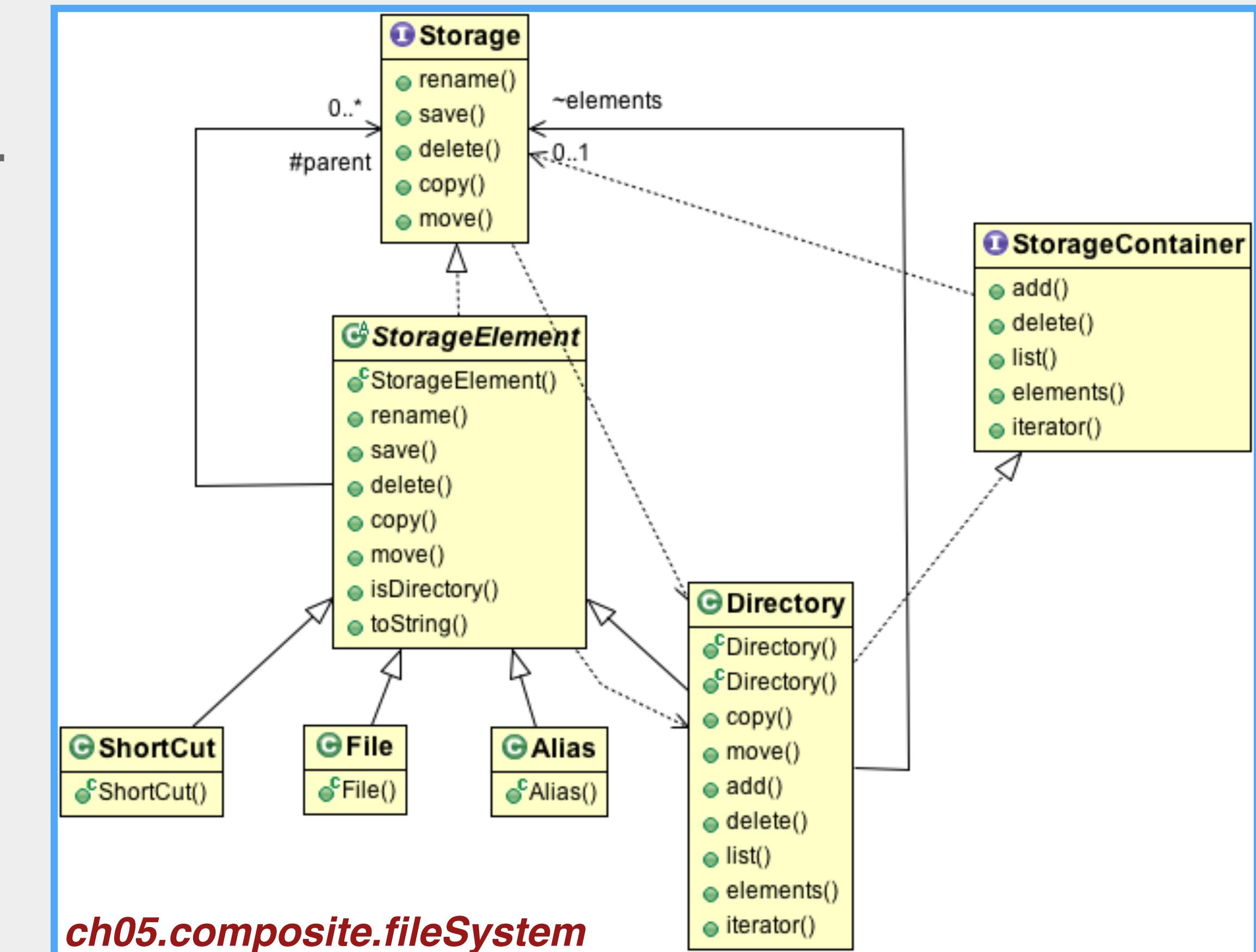


ch03.srp.paper.comm2

Directory



- **Directory** hem dizini temsil eden bir torba rolüne sahiptir hem de **File** gibi bir saklama, **StorageElement** yapısıdır.
- **Composite** kalıbı bu türden nesneleri kullanır.



Birden Fazla Üst Tipi Olan Sınıflar - II



- Bu durumlarda dikkat edilecek nokta, üst tip sayısının 3-5 gibi makul seviyede kalması ve yetkinlikleri bir araya toplayan (aggregate) sınıfın anlamlı bir role sahip olmasıdır.
- Benzer şekilde sınıfındaki toplam metot sayısının çok yükselmemesi de önemlidir.



- Daha önce de bahsettiğimiz gibi, ince arayüzlerden bir ya da daha fazlasını devralan sınıfların arayüzlerinin şişman ya da kirli olduğu düşünülebilir.
- Burada iki nokta vardır:
 - Tabiatı icabı birden fazla arayüzü bir araya getirmek kaçınılmazdır ve sınıf açısından tutarlı bir rol oluşturmaya yarar.
 - Çok sayıda ince arayüzün sağladığı fayda çok daha yüksektir.



Uygulama

BankAccount - II



- BankAccount soyutlamasının kendisiyle ilgili sorumluluklar, tekrar kullanılabilecek diğerlerinden ayırmalıdır.
- Diğer sorumluluklar başka yerlerde soyutlanmalıdır.

```
public interface BankAccount{  
    public Date getCreationDate()  
    public double getBalance();  
    public double getInterestRate();  
    public Customer getOwner();  
    public void deposit(double amount);  
    public void withdraw(double amount);  
    public List<Transaction> getHistory(Date from, Date to){}  
    public void print(){  
    public void save(){  
}
```

BankAccount - III



- **getHistory()**, karmaşıklığına göre farklı bir yere alınabilir.
- **print()** ve **save()** pek çok entity sınıfını ilgilendirmektedir.
- **save()**, bir kalıcılık (persistence) metodu olduğundan ilgili diğer metodlarla birlikte kalıcılık yapısında düşünülmelidir.

```
public interface BankAccount{  
    public Date getCreationDate(){}
    public double getBalance(){}
    public double getInterestRate(){}
    public Customer getOwner(){}
    public void deposit(double amount){}
    public void withdraw(double amount){}
    public List<Transaction> getHistory(Date from, Date to){}
    public void print(){}
    public void save(){}
}
```

BankAccount - IV

```
public class Account implements Printable, Persistable, Transactional{  
    public Date getCreationDate(){}
    public double getBalance(){}
    public double getInterestRate(){}
    public Customer getOwner(){}
    public void deposit(double amount){}
    public void withdraw(double amount){}
}
```

```
public interface Printer{  
    public void print(Printable printable);  
}
```

```
public interface Dao{  
    public void save(Persistable persistable);  
    public Persistable retrieve(int id);  
    public void update(Persistable persistable);  
}
```

```
public interface TransactionService{  
    public List<Transaction> getHistory(  
        Transactional transactional, Date from, Date to);  
}
```



Soyutlama Seviyeleri - I



- **SRP**'ye uygun dolayısıyla yüksek birlikte kalmakta problem yaşandığında yapılması gereken soyutlama seviyesini arttırap, problemi bir üst seviyeye taşıımaktır.
- Bir cümlede yapılmaya çalışılanı bir blokta ya da metotta yapmak,
- Bir metotta yapılmaya çalışılanı bir kaç metotlu bir sınıfta yapmak,
- Bir sınıfta yapılmaya çalışılanı bir kaç sınıf, arayüz vs. tiplerden oluşan bir modülde (**package**, **namespace**) yapmak.

Soyutlama Seviyeleri - II



- Bu şekilde daha geniş hareket alanında **SRP**'ye uygun yapılar üretilebilir.
- Tasarım kalıpları, bu şekilde soyutlama seviyesini değiştirerek problem çözmenin örnekleriyle doludur.
- Örneğin **Command** kalıbında, metot sınıf olarak modellenerek hem daha geniş hareket alanı kazanılır hem de polymorphism sayesinde **Strategy** kalıbındaki gibi nesnesin tipinin saklanmasıyla metot da saklanmış olur.

İsimlendirme ve Soyutlama



- Düzgün isimlendirme iyi soyutlamaların, kötü isimler, kötü soyutlamalar, birlikteliği düşük yapıların göstergesidir.
- Sık isim değişikliğinin sebebi iyi anlaşılmamış, soyutlanmamış yapılardır.
- Kullanılan kalıp ya da framework veya gelenekte yerleşmiş olma dışında isimlerin olabildiğince odaklı olması gereklidir.
- Bu sebeple detaylı, muhtemelen birkaç kelimeden oluşan isimler tek kelimelik genel simlerden çok daha faydalıdır.

changeBalance	→ withdraw, deposit
process	→ processOrder, processDailyBills,
execute	→ executeLogin, executeBranchDisposalRequest
Util	→ StringUtil, FileUtils, JsonUtil
AccountService	→ AccountMonetaryService, AccountHistoryService



Uygulama

Uygulama

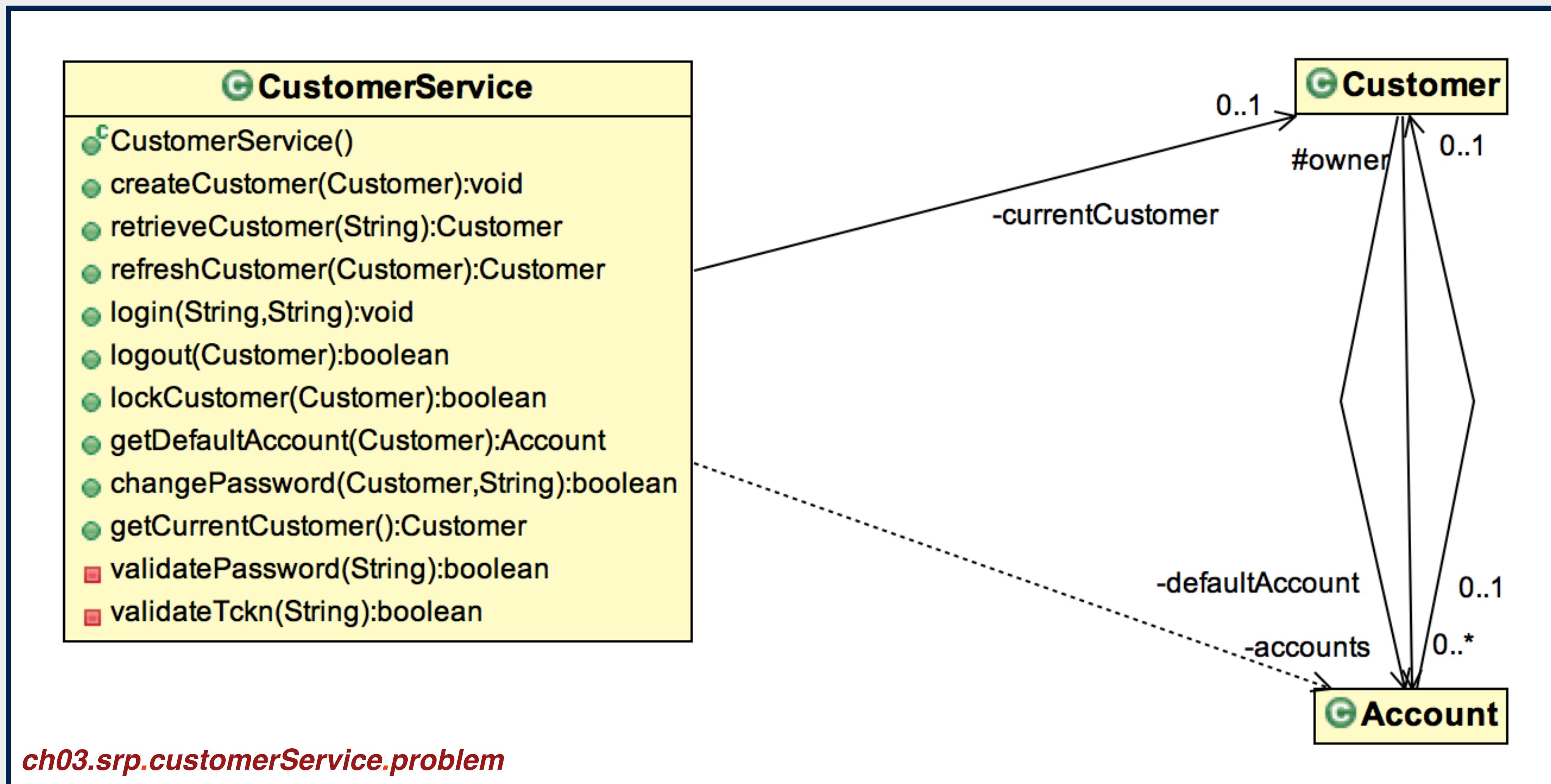


- `org.javaturk.dp.ch03.srp.customerService.problem.CustomerService` sınıfını **SRP** açısından değerlendirin.
- Varsa problemlerini listeleyin
- Ve problemlere çözümler sunun.
- Çözümünüzü yeni bir sınıf diyagramıyla çizin.

Uygulama



- Metotların hiçbirisi doğrudan veri tabanına ulaşmamaktadır, DAO nesnelerini düşünmeyin.
- `lock()` metodu 3 defa arka arkaya yanlış password giren **Customer**'ı kilitlemektedir.
- **Customer** ile **Account** arasında bir iki yönlü digeri tek yönlü iki ilişki vardır.
 - **Customer**'ın **Account**'u ve **Account**'u sahibi vardır.
 - **Customer** varsayılan/default **Account**'u bilir.



ch03.srp.customerService.problem

Open-Closed Principle

Open-Closed Principle



Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Yazılım yapıları (sınıflar, modüller, fonksiyonlar, vs.) genişletmeye açık ama değişime kapalı olmalıdır.



- Yazılım sistemleri, yeni özellik, yeni kullanıcı, yeni iş kuralları vs. sebebiyle sürekli değişir.
- Bu değişiklikler nasıl yerine getirilmelidir?
- Var olan yapıların değişmesiyle mi?
 - Hayır!



- Yazılım sisteme yapılacak değişiklikler, var olan yapıların değişmesiyle değil, genişletilmesiyle yerine getirilmelidir.
- Yazılım yapıları değişime kapalı olmalıdır, değiştirilmemelidir.
- Yeni ihtiyaçlar var olan yazılım yapılarının genişletilmesiyle karşılanmasıdır.

Değişime Kapalı



- Yazılım yapılarının değişime kapalı olmaları, kaynak kodu seviyesinde değişiklik yapılmaması gerektiğini ifade eder.
- Var olan arayüzler, sınıflar ve metodlar, herhangi bir değişikliğe uğramadan yaşamaya devam etmelidir.

Genişletilmeye Açık - I



- Genişletilmeye açık olmak ile farklı şeyler kastedilebilir:
- Yapının en baştan, değişimi göz önüne alacak şekilde, değişimdeki kısımların, değişimeyecek kısımlardan yalıtılarak kurgulanması, dolayısıyla yeniliklerin var olan yapıları değiştirmeden yapılması,
- Kaçınılmaz bağımlılılıkların olabildiğince soyutlamalar üzerinden yapılması, gerçekleştirmelere bağımlılık oluşturulmaması.

Genişletilmeye Açık - II



- Sonucunda da yeni geliştirmeleri için var olan tiplerin farklı amaçlara hizmet edecek şekilde alt tiplerinin oluşturulması ve sahip olduğu davranışların **ezme (override)** ile farklılaştırılması söz konusudur.



- **OCP**'nin stratejik hedefi tekrar kullanımı (resusability) artırmak ve bakımlanabilirliktir (maintainability)
- Böylece hem yazılım ürününün bakım maliyeti azalacak hem de yeni ürünler tekrar kullanım ile daha hızlı hayatı geçebilecektir.

Kaçınılacaklar

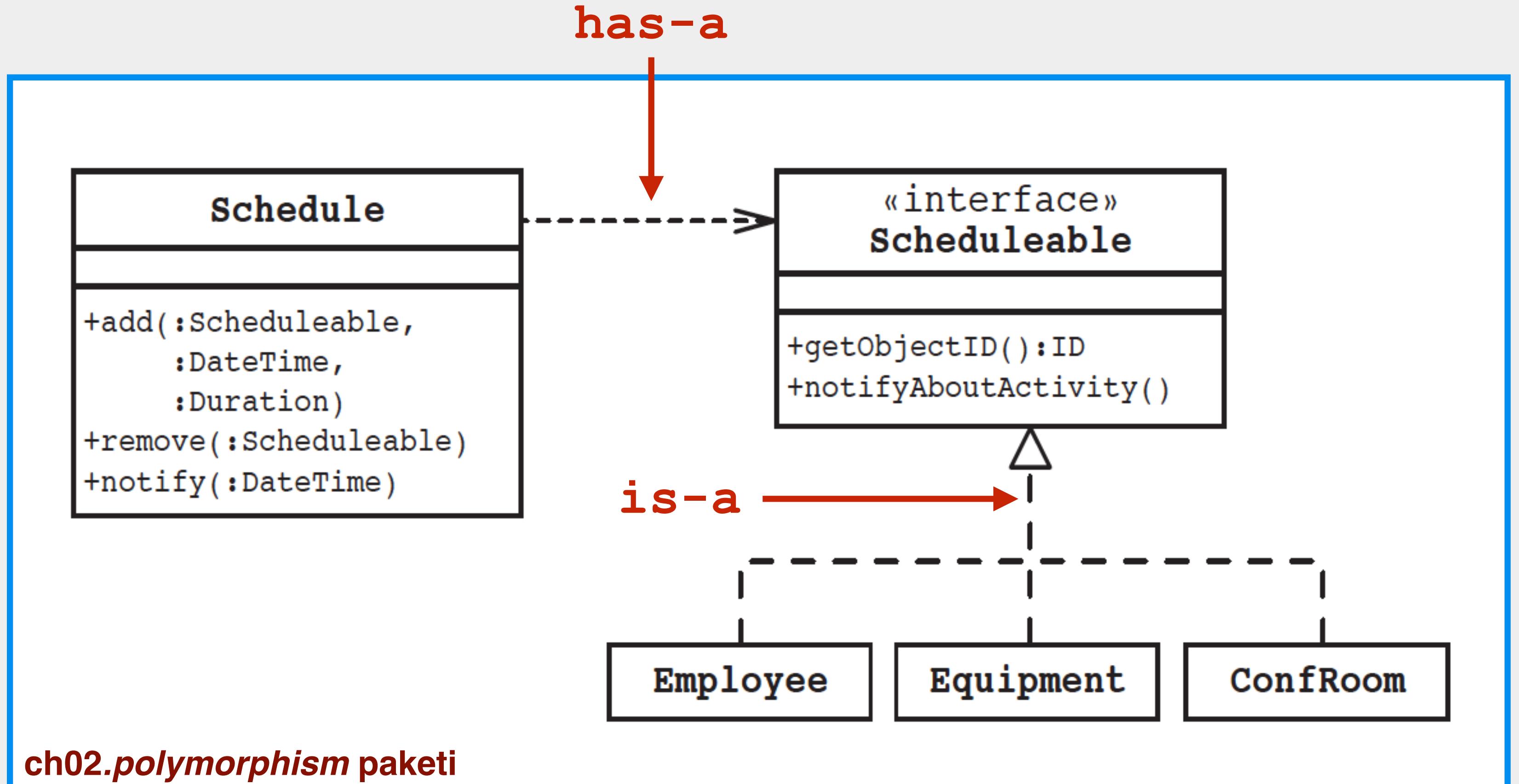


- Dolayısıyla şunlardan kaçınılmalıdır:
 - Düşük birliktelikli yapılar,
 - İç yapılara bağımlılıklar,
 - Somut tiplere bağımlılıklar,
 - Global değişkenler,
 - RTTI (Run-time Type Identification)

Soyut Bağımlılıklar



- Burada hem **is-a** (miras) hem de **has-a** bağımlılıkları soyuttur.
- Bu durumda yeni yapılar genişletmeyle oluşturulabilir.



```
Schedulable s = new Employee();  
s = new Equipment();  
s = new ConfRoom();
```

```
Schedule sc = new Schedule();  
sc.add(new Employee(), ..., ...);  
sc.add(new Equipment(), ..., ...);  
sc.add(new ConfRoom(), ..., ...);
```



Uygulama



- Account sınıfındaki `changeBalance()` metodunu OCP açısından değerlendirip OCP'ye uygun hale getirelim.

```
public void changeBalance(String action, double amount)
    throws InsufficientBalanceException, NegativeAmountException{
    if(amount < 0)
        throw new NegativeAmountException(amount);

    if(action.equals("Deposit"))
        balance += amount;
    else if(action.equals("Withdraw")){
        if(balance ≥ amount)
            balance -= amount;
        else
            throw new InsufficientBalanceException(action, balance, amount);
    }
    log.info(action + " : " + amount + " for account id: " + id);
}
```

ch02.coupling.account.account1



- `changeBalance()` metodu, kendisine geçilen `String` tipinde `action` verisinden dolayı değişime kapalı değildir.
- Muhtemelen `action` bir başka sınıfın iç işleyişinin bir detayıdır.
- Dolayısıyla sadece veri olarak bu metoda geçilmesi, içerik bağımlılığıdır (content coupling) ve birlikte değişme problemi oluşturur.
- `changeBalance()` **SRP** olmadığı gibi **OCP** de değildir.



- Bunu yerine **AccountAction** tipinde nesne geçerek değişiklik ihtiyacı genişletmeyle karşılanabilir hale getirilebilir.
- Veriye olan bağımlılık nesneye (veri + davranış) döner ve soyut hale gelir.
- **changeBalance ()** metodu içindeki iş mantığı da farklı **AccountAction** nesnelerine dağıtıılır ve yüksek birliktelikli olarak halledilir.
- Bu şekilde **changeBalance ()** metodunun birlikteliği artar ve **OCP**'ye uygun hale gelir.

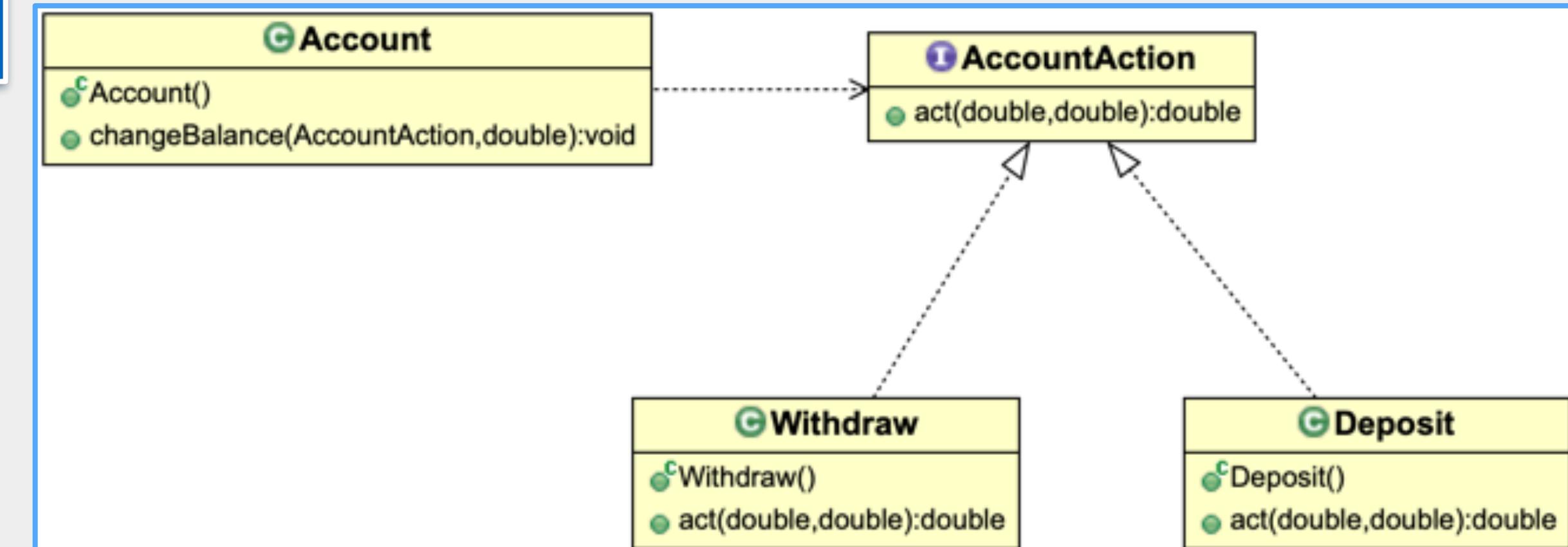
```
public void changeBalance(Action action, double amount)
throws InsufficientBalanceException, NegativeAmountException{
if(amount < 0)
    throw new NegativeAmountException(amount);
action.act(amount);

}
log.info(action + " : " + amount + " for account id: " + id);
}
```

```
public interface AccountAction {
    public double act(double balance, double amount)
        throws InsufficientBalanceException;
}
```

```
public class Withdraw implements AccountAction {

    @Override
    public double act(double balance, double amount)
        throws InsufficientBalanceException {
        if (balance ≥ amount)
            balance -= amount;
        else
            throw new InsufficientBalanceException(balance, amount);
        return balance;
    }
}
```



ch02.coupling.account.account2



- **OCP**'nin ifade ettiği, var olan yapıların herhangi bir değişikliğe uğramadan yaşamaya devam etmesinin idealize bir durum olduğu açıktır.
- Çünkü var olan hiç bir kod parçasını değiştirmeden geliştirmeye devam etmek imkansızdır.
- İş ihtiyaçlarından dolayı olmasa bile refactoring her zaman gereklidir!
- Bu sebeple **OCP** gerçekleştirilmesi imkansız bir prensiptir.
- Bu prensip bir hedef olarak düşünülmeli, geliştirme sırasında var olan yapıların olabildiğince değiştirilmemesi hedeflenmelidir.

Sabit Kalan ve Değişen - I



- Bu durumda sistemin ve parçalarının ne yönde evrileceğini öngörmek stratejik olarak çok değerlidir.
- Strateji, sık değişeceğin kısımların kolay değişeceğin şekilde tasarlanmasıdır.
- Fiziksel mühendislik ürünlerinde her şey değişebilir çünkü bozulma ve aşınma söz konusudur.
- Ama sıkı değişeceğin parçaların daha kolay değişmesi sağlanır, daha nadir değişeceğin, uzun ömürlü parçalar ise nispeten zor değişir.

Sabit Kalan ve Değişen - II



- Yazılımda değişen ile sabit kalanı ayırt etmek fiziksel mühendislik ürünlerindeki kadar öngörülebilir değildir.
- Değişen-sabit ayırımı her şeyden önce iyi bir ürün bilgisi, piyasa ve kullanıcı eğilimleri ve davranışları, rekabet analizi vb. etmenlere bağlıdır.
- Bu konular temelde iş mimarisini (business architecture) belirler.
- Dolayısıyla iş mimarisi değişimi yönetmede en stratejik disiplindir.

Sabit Kalan ve Değişen - III



- Dolayısıyla **OCP** sadece uygulama geliştirmeden sorumlu mimar/tasarımcı/geliştiricinin elinde değildir.
- Mimar/tasarımcı/geliştirici, öncelikle tecrübeyle diğer paydaşları değişim üzerinde düşünmeye zorlar,
- Ve ortaya konan değişim noktalarını göz önüne alarak takım olarak stratejik seçimlerde bulunmaya yardımcı olur.
 - Hangi değişimlerin ihtimali ve maliyeti daha yüksektir?
 - Bu çalışmadan çıkanlara göre yazılımı tasarlar ve geliştirir.

Sabit Kalan ve Değişen - IV



- Değişimin hangi yönlerde gerçekleşeceğini, hangi parçaların değişmeden kalma eğiliminde olduğunu vs. belirlemek her zaman kolay değildir.
- Burada iki uç tutumdan bahsedilebilir:
 - Aşırı ön tasarım (up-front design) ile vakit kaybetmek ve yeterince bilgi olmadığı halde değişim ile ilgili kararlar vermek,
 - Bu durumun sonucu **YAGNI**'ye (**You're Not Gonna Need It**) düşcar olmaktadır.
 - Herşeyin nasıl olsa değişeceğini düşünerek, kervanın yolda düzülür deyip, çala-kalem kodlamaya girişmek.

Sabit Kalan ve Değişen - V



- İlk tutum boş harcanan zamandır, çünkü proje ilerledikçe ve daha fazla bilgi sahibi oldukça yeterince bilgi sahibi olmadan alınan kararların geçersizliği ortaya çıkacak ve ön tasarımda gereksiz detaya girilerek vakit kaybedildiği anlaşılacaktır.
- İkinci tutumda ise proje ilerledikçe geliştirilmiş kodun değiştirilmesi büyük bir yük yaratacak, hatta mimari öneme sahip noktalarda iyileştirme yapmak çok maliyetli ya da imkansız olacaktır.
- Daha projenin başlarından itibaren ciddi bir bakım maliyeti ortaya çıkacaktır.

Sabit Kalan ve Değişen - VI



- Bu iki nokta arasında dengeli bir tutuma sahip olmak, tecrübe gerektirir.
- Ayrıca yazılım ürününün/projesinin tabiatı, ne kadar oturmuş bir sektörde, bilinen bir ürün grubuna dahil olduğu ya da ne kadar yenilikçi, türünün ilk örneği cinsinden olduğu da önemlidir.
- Örneğin ikinci durumda ARGE türünden, sabit kalacak ile değişeni keşfetmeye ve anlamaya yönelik çalışmalar daha fazla olacaktır.
- Bu durumda bir süre sonra hangi kısımlara çok değişiklik yapıldığının belirlenerek oralarının daha esnek hale getirilmesi gibi stratejiler uygulanabilir.



- Bir kod parçasının **OCP**'ye (muhtemelen **SRP** gibi diğer prensiplere de) uyma noktasında sıkıntılı olduğuna dolayısıyla refactoring yoluyla iyileştirilmesi gereğine ne zaman karar verirsiniz?
 - İlk değiştirmemde,
 - En az 2 kere değiştirdiğimde,
 - Genelde bir modülü toptan değiştirmeyi gerektiren büyülükte işlerde,
 - Küçük-büyük demeden projede zaman ayırır refactoring yaparım.

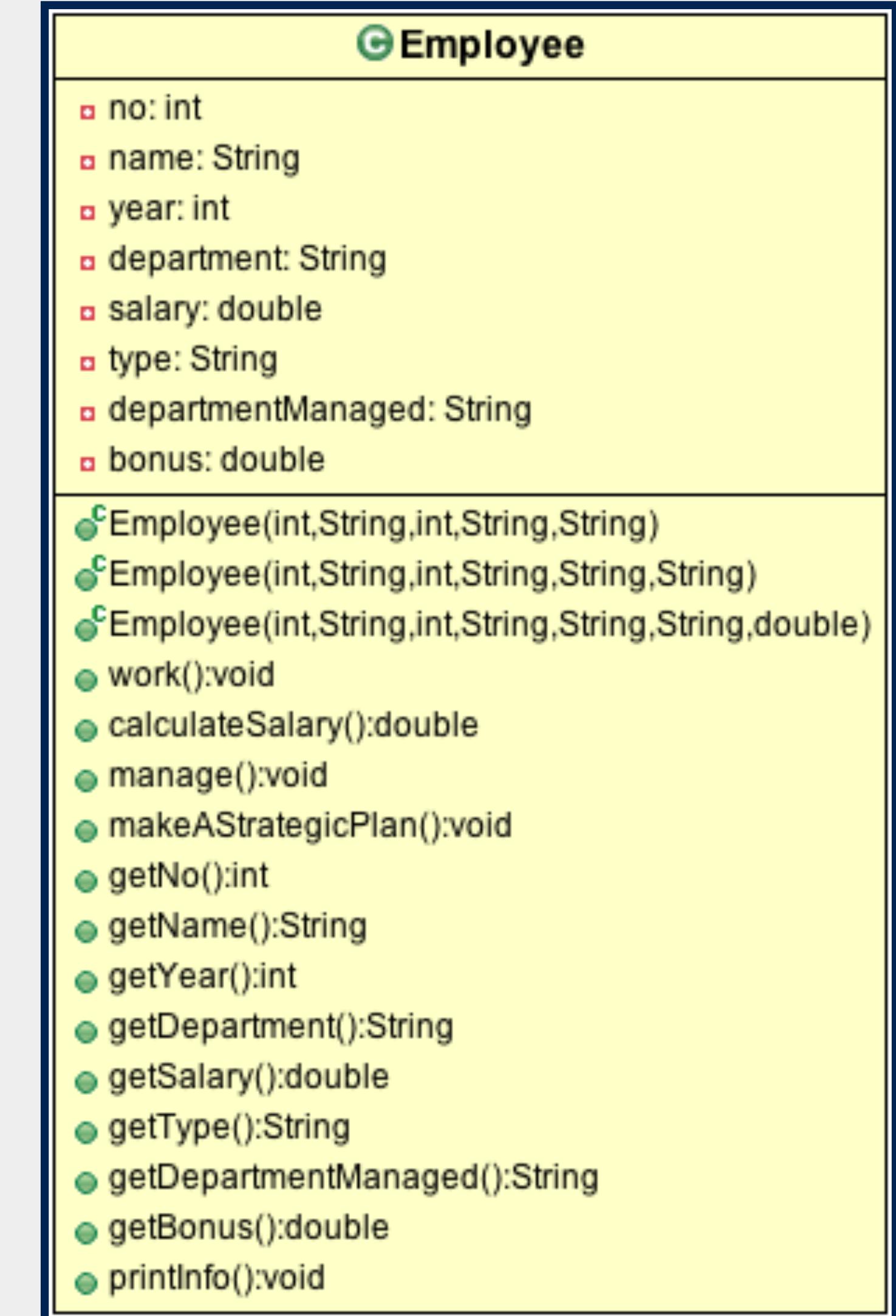


Uygulama

Uygulama



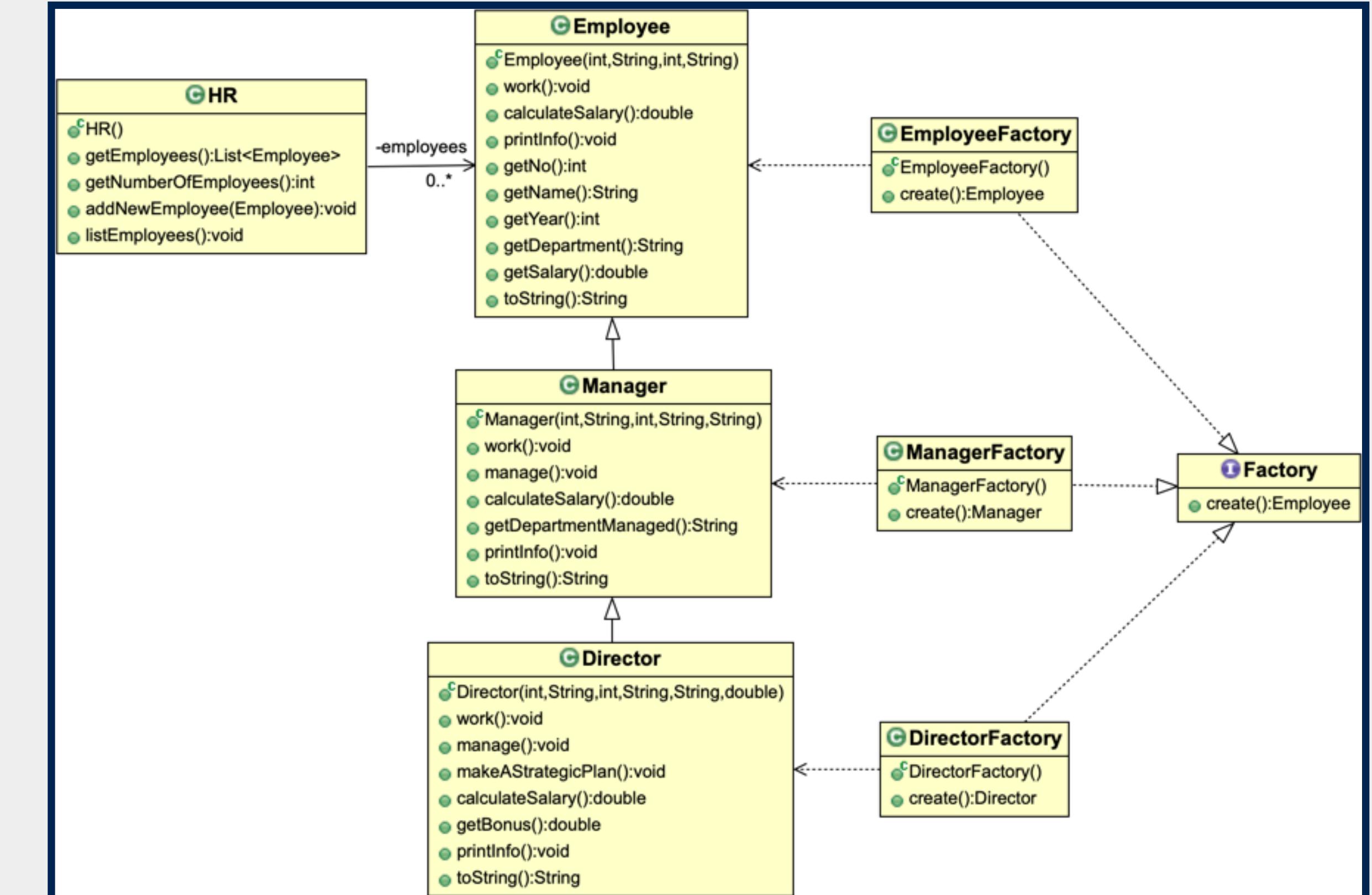
- Yandaki **Employee** sınıfını ve üzerindeki metotları **OCP** açısından değerlendirin.
- Nasıl bir çözüm önerilebilir, tartışın.



Uygulama

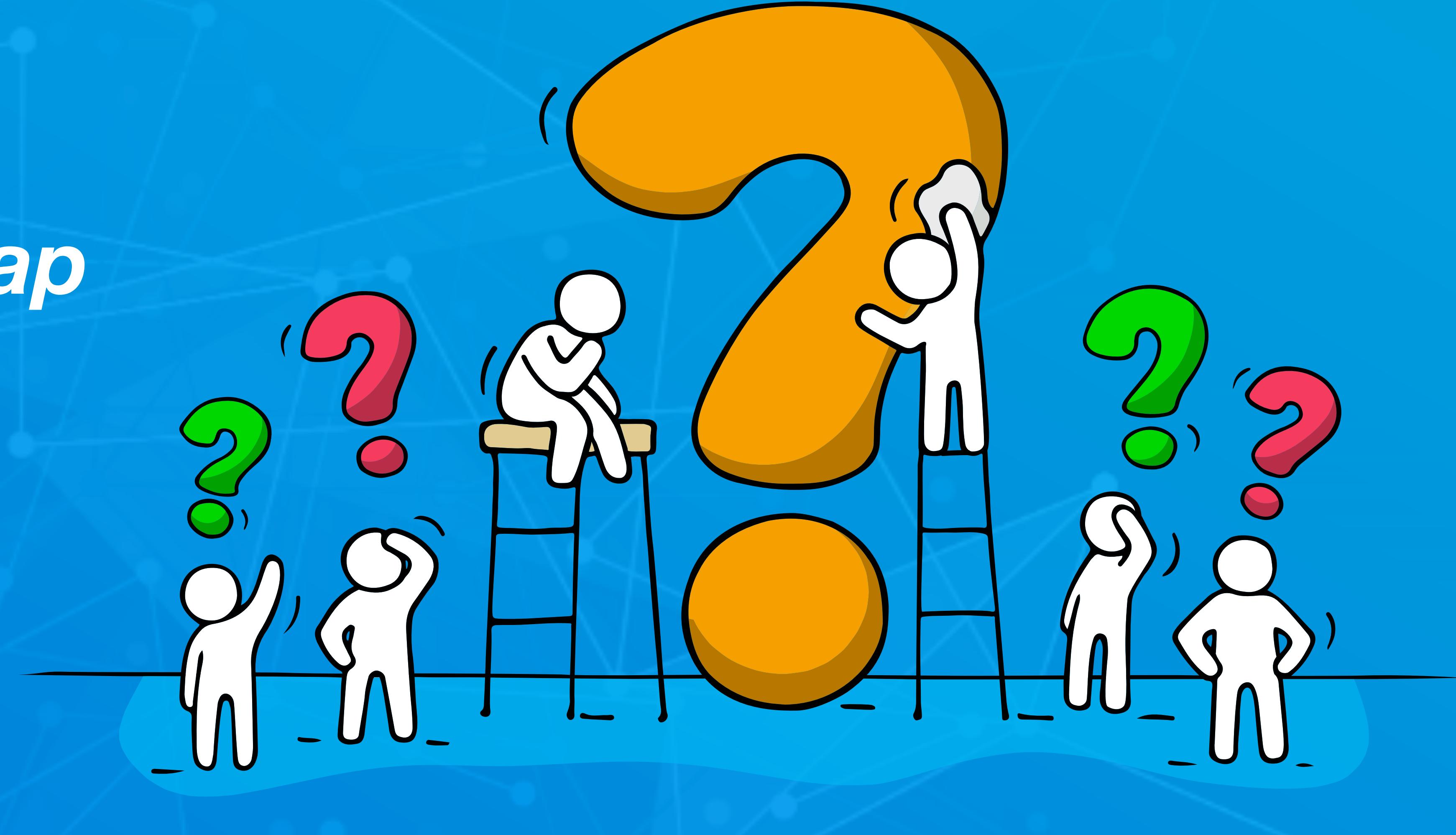


- Employee sınıfının değişmeden farklı Employee tiplerinin, genişletilmeyle (extension) oluşturulması, OCP'nin uygulanmasına bir örnektir.



ch03.ocp.factory.solution

Soru ve Cevap Zamani!



Liskov Substitution Principle

Liskov Substitution Principle



- **Liskov Substitution Principle (LSP)** ilk defa 1987 yılında, MIT'de profesör olan Barbara Liskov tarafından ifade edilmiştir.
- Daha sonra R. C. Martin tarafından C++ Journal dergisinde makale olarak yayınlanmıştır.
- Akademik olarak 1999 yılında yayınlanan “**Behavioral Subtyping Using Invariants and Constraints**” başlıklı makalede ifade edilmiştir.



Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

Taban sınıfa işaretçi ya da referans kullanan fonksiyonlar, türetilmiş sınıfların nesnelerini de bilmeden kullanabilmelidirler.

- Buradan bilinmesi gerekmeyen bilgiden kasıt, türetilen nesnelerin gerçek tipleri ile onlara has bilinmesi gereken durumlardır.



- Orijinal makaleye Liskov şöyle başlar:

We present a way of defining the subtype relation that ensures that subtype objects preserve behavioral properties of their supertypes.

- Esas soru da şudur:

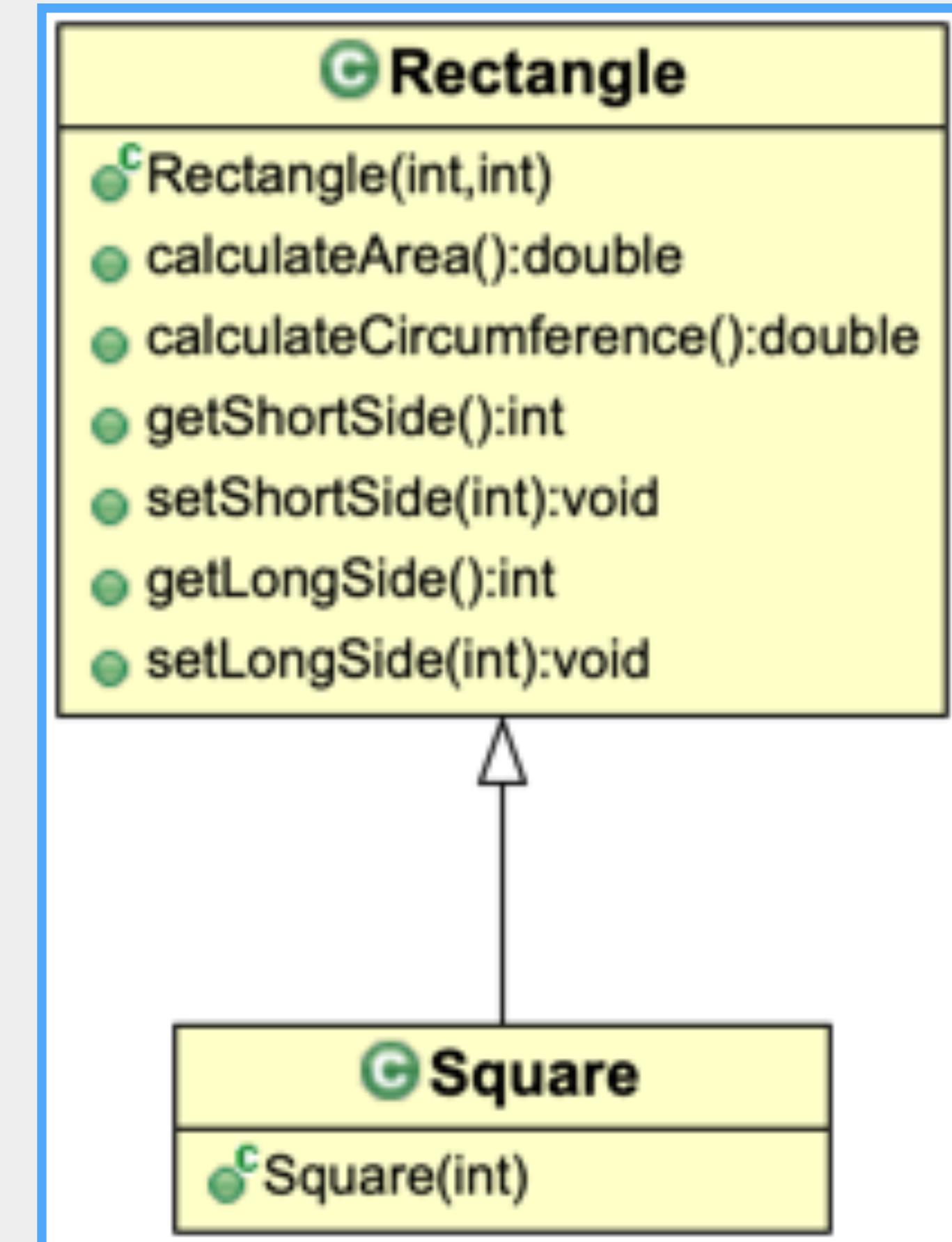
What does it mean for one type to be a subtype of another? We argue that this is a semantic question having to do with the behavior of the objects of the two types: the objects of the subtype ought to behave the same as those of the supertype as far as anyone or any program using supertype objects can tell.



- Dolayısıyla **LSP** ilişkisi, sadece subtyping ya da miras ilişkisini basitce söz dizimi (syntactic) açısından ele alan bir prensip değil, aslen anlamsal (semantic) bir prensiptir.
- Dolayısıyla alt tipler, üst tiplerin koymuş olduğu ve istemcilerin bildiği ve alışkin olduğu davranışsal çerçeveyi değiştirmemeliler, istemcileri şaşırtmamalılar.



- Martin, makalesinde **LSP**'ye ters örnekler verir ve kaçınılmazı gereken durumları listeler:
 - Yerine geçmede, anlamsal uyumu göz önünde bulundurarak, problem çıkan durumlardan kaçınmak.
 - Örneğin, **Kare**'nin **Dikdörtgen**'den miras alması:
 - **Kare**'nin ayrı genişlik ve uzunluk özelliklerinin olmamasından (ikisi bir birine eşittir) dolayı bu ilişki **LSP**'ye uymaz.

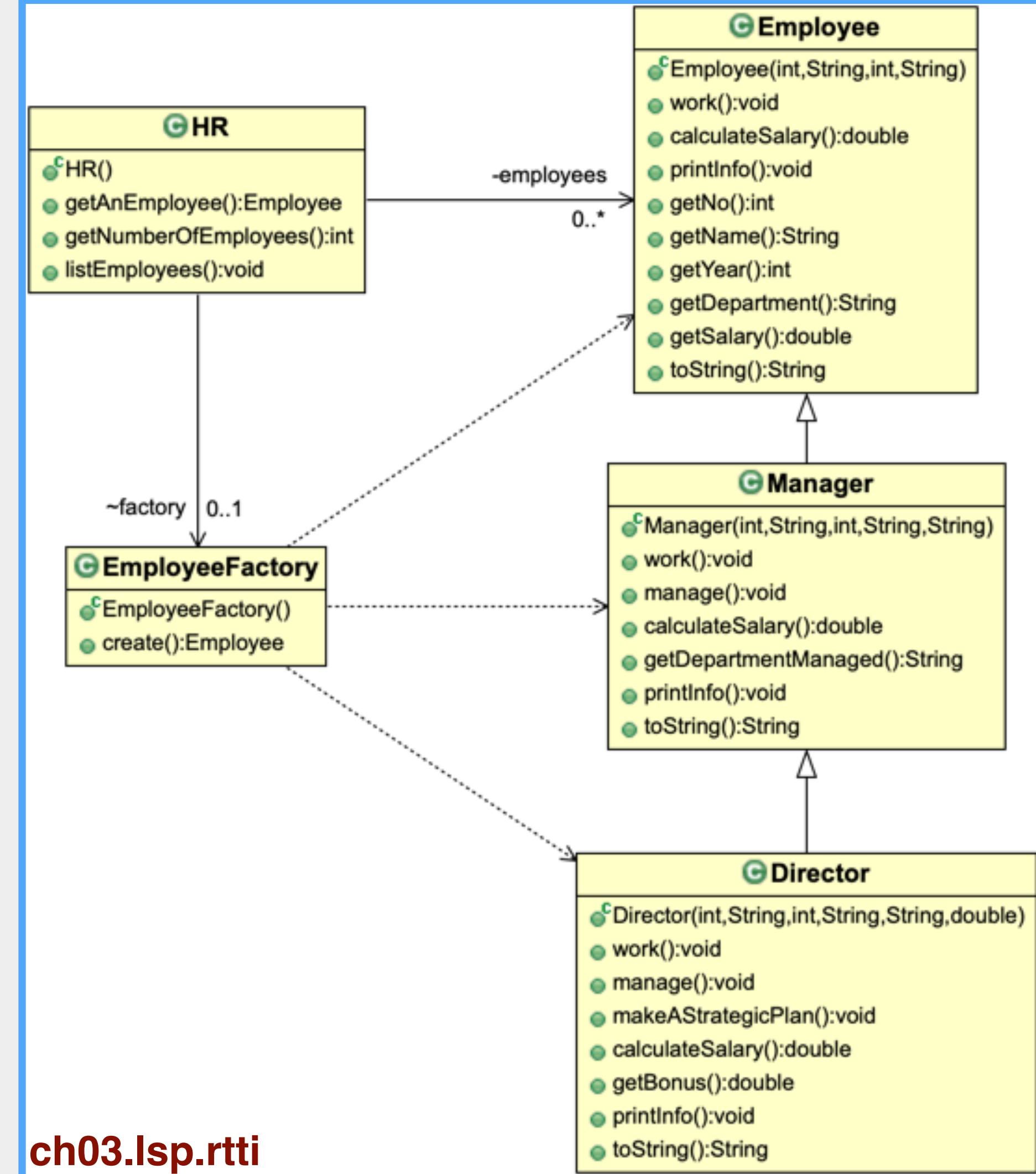


ch03.lsp.square



- Run-time Type Identification'dan (RTTI) kaçınmak,

```
Employee e = hr.getAnEmployee();
if (e instanceof Director) {
    Director d = (Director) e;
    ...
} else if (e instanceof Manager) {
    Manager m = (Manager) e;
    ...
}
else{...}
```



ch03.lsp.rtti



Design by Contract

Design by Contract - I



- **Design by Contract (DbC)**, Bertrand Meyer tarafından 1986 yılında, Eiffel dilini tasarlarken ortaya konulmuştur.
- Yazılım bileşenleri için formal (formal), kesin (precise) ve doğrulanabilir (verifiable) arayüzlerin tasarlanması gerektiğini ifade eder.
- Bu amaçla arayüzler için ön ve son şartlar (pre- and post-conditions) da söz konusudur.
- Liskov'un davranışsal özellik (behavioral property) dediği şey Meyer'in ön ve son şartıdır.

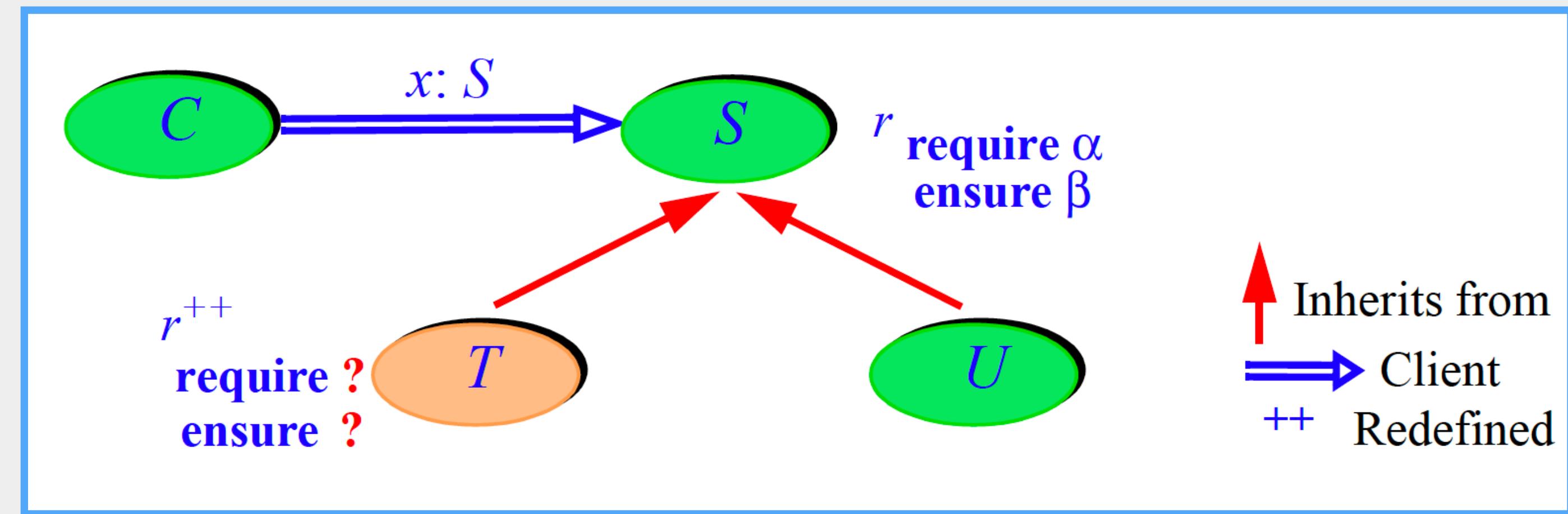


- Ön şartlar, metodun istemcisi tarafından çağrılması için gereklidir,
- Son şartlar ise metodun çalışmasını bitirdiğinde oluşacak nesnenin durumu, döndürülen değer vs. ile ilgilidir.
- **Alt tipler ön şartları (require) zayıflatırabilir ve son şartları (ensure) sıkılaştırabilir.**
- Yani alt tipler üst tiplerin kabul etmediğini kabul edebilir ve daha iyi, özel (specific) sonuçlar verebilir.

Design by Contract - III



- Miras aslen bir **genelleştirme-özelleştirme** (**generalization - specialization**) ilişkisidir.
- Üstte daha genel altında daha özel tipler bulunmalıdır.
- Hiyerarşî tersine çevrilmemeli ve şaşırtmamalıdır.



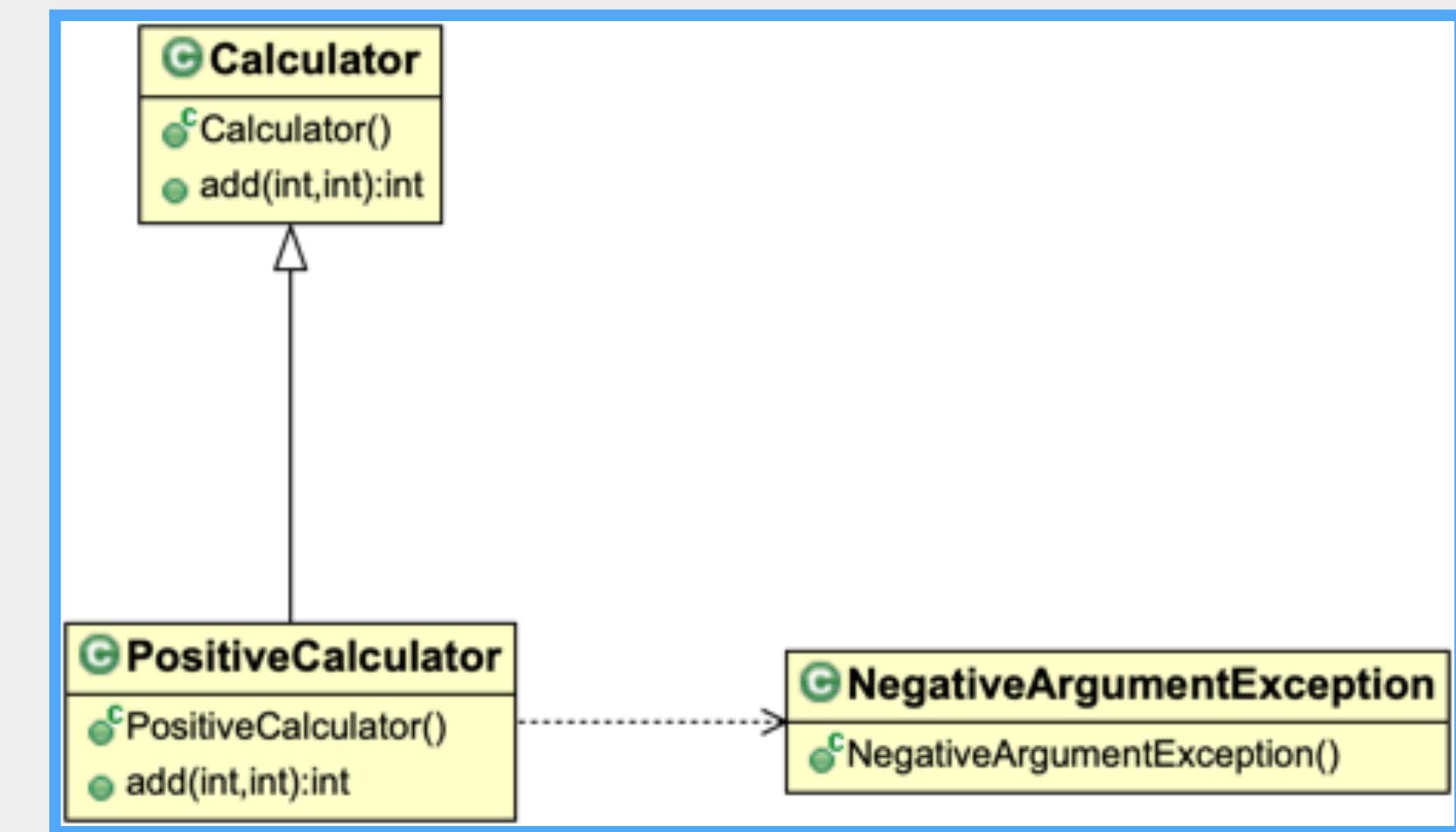


- Dolayısıyla üst tipi bilen istemciler alt tipler tarafından şaşırtılmamalıdır ama aldıkları hizmet beklediklerinden daha özel olabilir.
- Örneğin taban (base) tipteki metodun fırlatmadığı bir sıra dışı durumu alt tipin ezdiği (override) metotta fırlatmak.
 - Bu ön şartın sıkılaştırılmasıdır ve kabul edilemez.
 - Java ve C# gibi modern dillerde bu durumlar önlenmiştir.

Calculator & PositiveCalculator - I



- **Calculator** iki tam sayıyı toplama hizmeti vermektedir.
- **PositiveCalculator** ise iki pozitif tam sayıyı toplama hizmeti vermektedir ama negatif argüman geçilmesi durumunda **NegativeArgumentException** fırlatmak istemektedir.



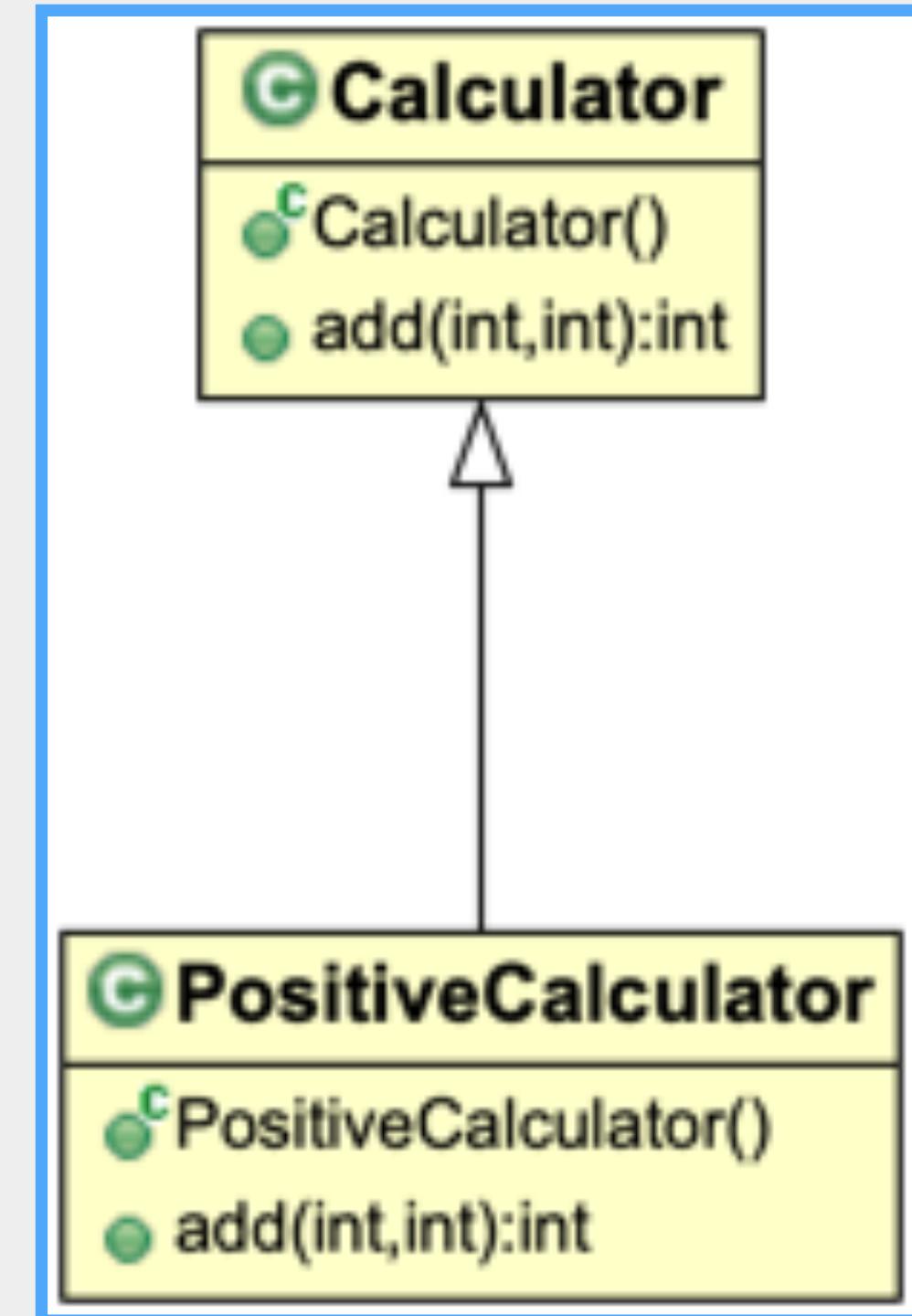
ch03.Isp.designByContract.calculator.ex

- Bu ise **Design by Contract**'ın tavsiye ettiğinin tersine, üst tipteki **add ()** davranışında olmayan bir ön şart eklemektir.

Calculator & PositiveCalculator - II



- Üst tipin davranışlarıyla ilgili olarak öne sürmediği bir şartı alt tipin öne sürmesi ya da var planı daha da sıkılaştırması sadece fırlatılan sıra dışı durumlarda söz konusu olmaz.
- PositiveCalculator** iki pozitif tam sayıyı toplama hizmeti vermek için önce bir kontrol yapar sonra uygunsa işlemi gerçekleştirir, aksi takdirde -1 döndürür.
- Bu da **Design by Contract**'ın tavsiye ettiğinin tersine, üst tipteki davranışta olmayan bir ön şart eklemektir.
 - Bu da **Calculator** ile çalışmasını bilen istemciyi şaşırtır ve onu RTTI ile gerçek tipleri kullanmaya zorlar.

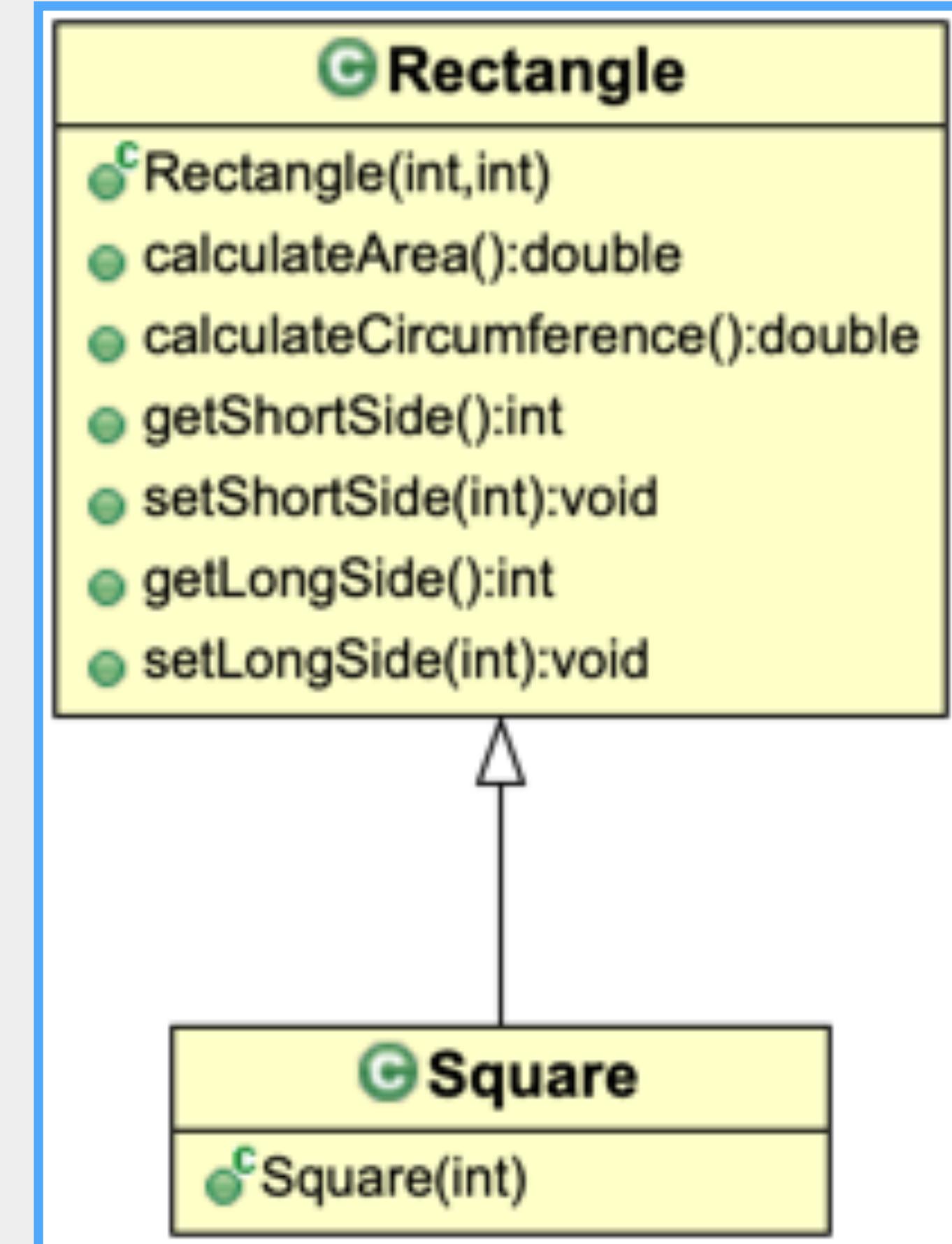


ch03.Isp.calculator.bad

Square ve Rectangle - I



- Daha önce tavsiye edilmeyen **Square** - **Rectangle** ilişkisinde **Rectangle**'ın öne sürmediği bir durum, kısa ve uzun kenarın birbirine eşit olması **Square** tarafından ön şart olarak öne sürülmektedir.
- **Square** nesnesini **Rectangle** olarak gören istemci, kısa ve uzun kenarlara ayrı ayrı değerler vererek, onların farklı olduğunu farzedecektir.

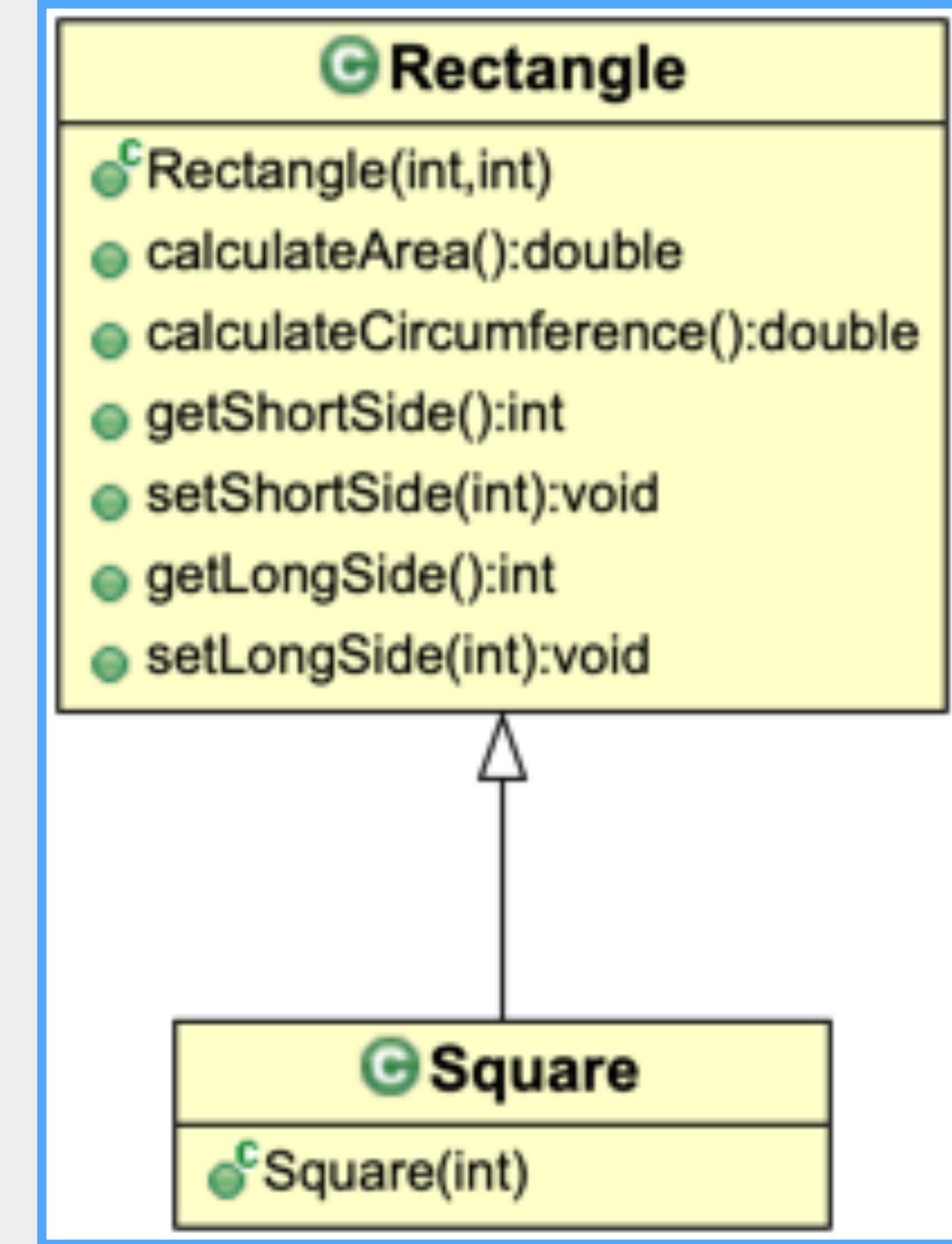


ch03.Isp.square

Square ve Rectangle - II



- Ama **Square** sadece en son geçilen kenar bilgisini tutmaktadır.
- Bu durumda üst arayüz hakkında yapılan kabul, alt arayüz tarafından geçersiz kılınmaktadır.
- Bu durumda hiyerarşî problemi yoktur ama **Design by Contract**'ın tavsiye ettiğinin tersine **Square** nesnesi, davranışları için üst tipinde olmayan bir ön şart öne sürmektedir.

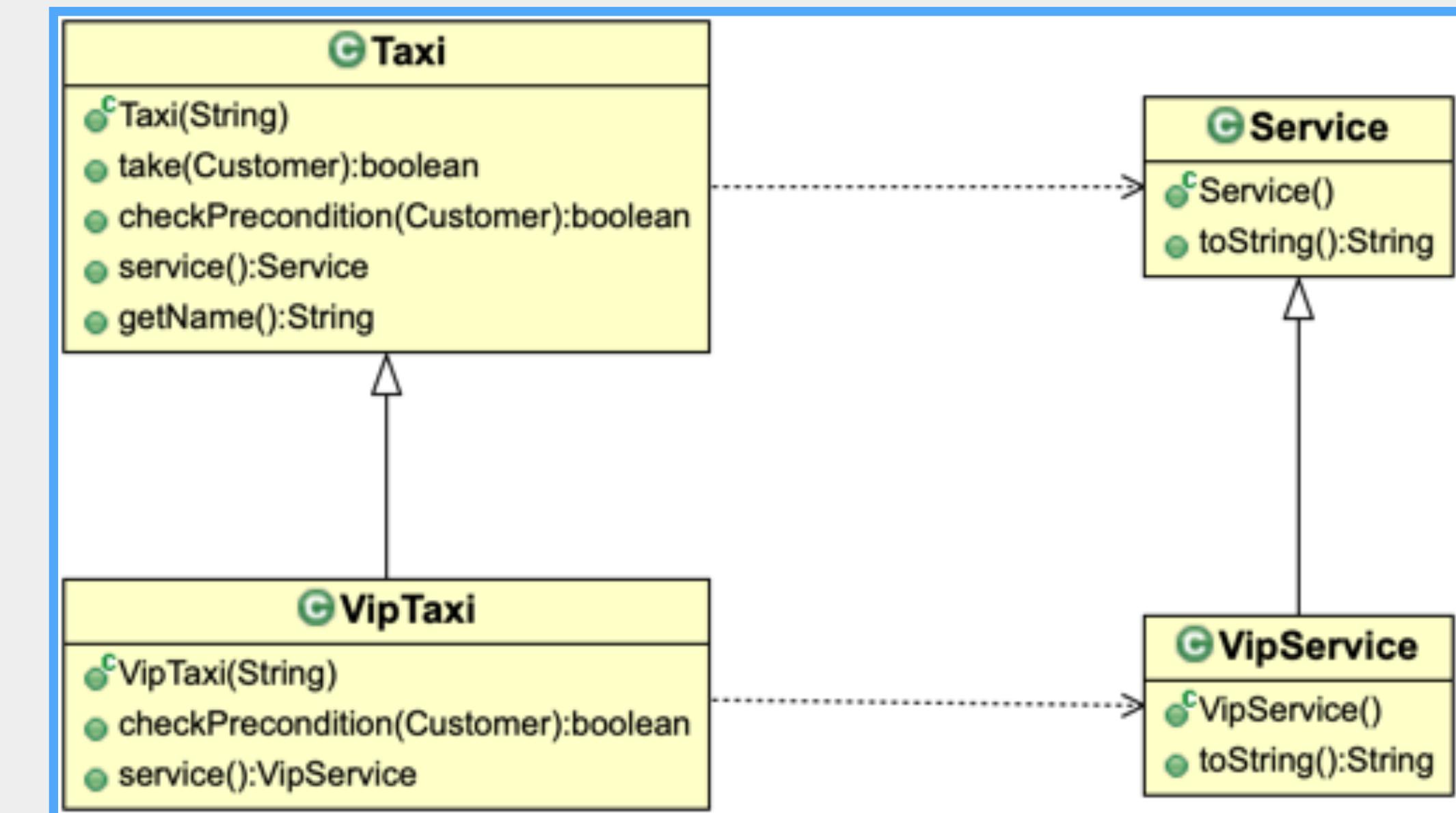


ch03.Isp.square

Taxi & VipTaxi



- Alt tipler, override ettikleri davranışlarda, üst tipin sağladığından daha özel dönüş tipleri sağlayabilir.
- VipTaxi Taxi'nin sağladığı **take()** hizmeti için gerekli ön şartı kaldırırken **service()** sonucunda **Service** yerine daha özel bir **VipService** cevabı döndürmektedir.



ch03.Isp.good



```
public class Taxi{  
  
    public boolean take(Customer customer) {  
        // Require  
        if(checkPrecondition(customer)) {  
            this.customer = customer;  
            return true;  
        }  
        else  
            return false;  
    }  
  
    public boolean checkPrecondition(Customer customer){  
        if(customer.getDistance() < 1000)  
            return true;  
        else  
            return false;  
    }  
  
    public Service service() {  
        return new Service();  
    }  
}
```

```
public class Service {  
    protected String description="Destination arrived!";  
  
    @Override  
    public String toString() {  
        return "Service [description=" + description + "]";  
    }  
}
```

```
public class VipTaxi extends Taxi {  
  
    public VipTaxi(String string) {  
        super(string);  
    }  
  
    public boolean checkPrecondition(Customer customer){  
        return true;  
    }  
  
    @Override  
    public VipService service() {  
        return new VipService();  
    }  
}
```

ch03.lsp.good

```
public class VipService extends Service{  
    private String vipDescription = "Wifi provided! ";  
  
    @Override  
    public String toString() {  
        return "Service [description=" + vipDescription +  
               description + "]";  
    }  
}
```

Defensive Programming



- **Defensive Programming** de sadece miras yapısıyla ilgili değil ama genel programlama ile ilgili korumacı teknikleri içeren bir yaklaşımdır:
- <https://wiki.c2.com/?DefensiveProgramming>
- DbC ve DP hakkında detaylı bilgi için B. Meyer'in şu kitaplarına bakılabilir:
 - **Object-Oriented Software Construction**
 - **Touch of Class: Learning to Program Well with Objects and Contracts**



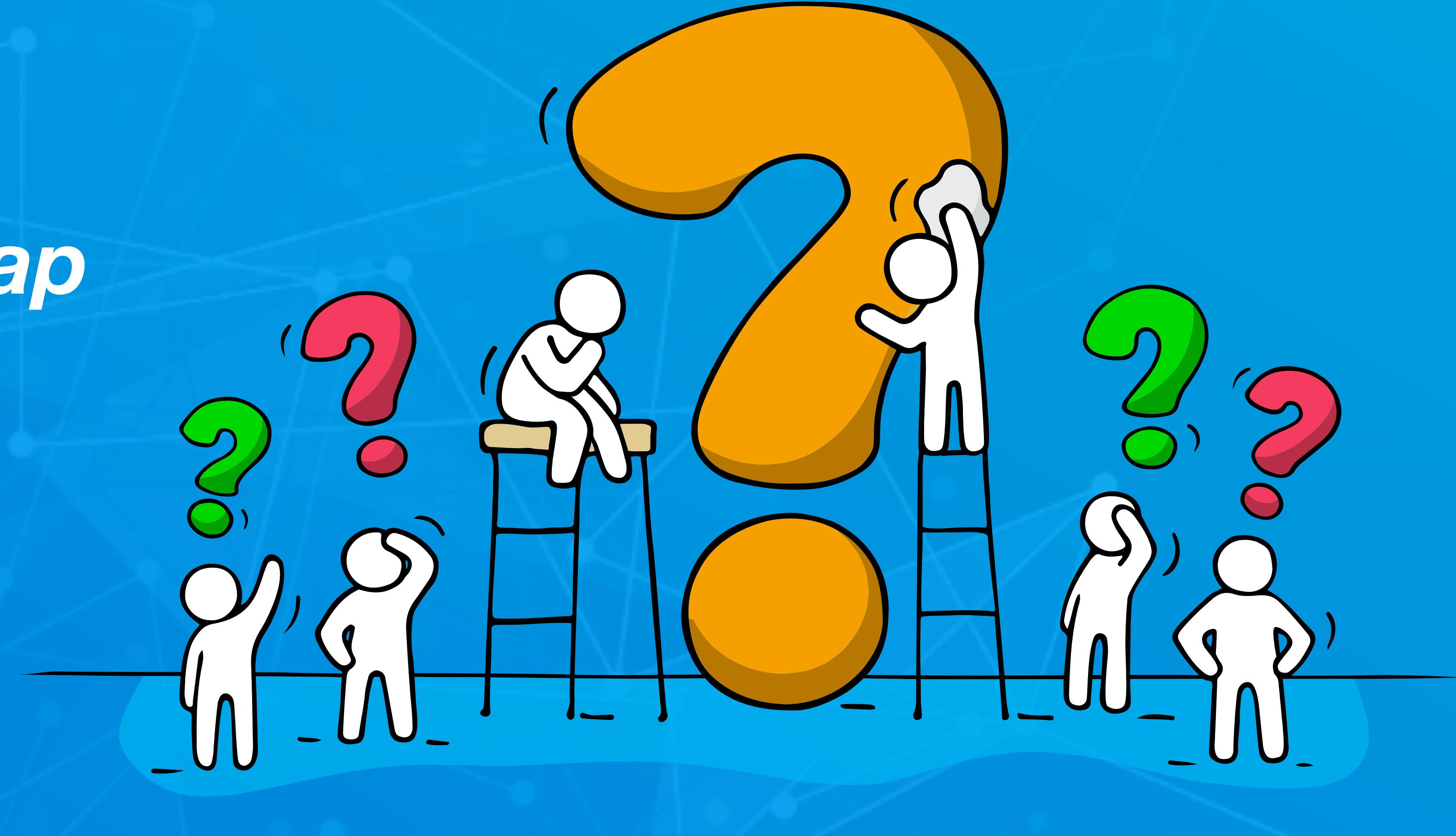
- **LSP** tasarlanan yapının o yapıyı kullanacak istemciler açısından doğrulanması gerektiğini ifade eder.
- Kurulan modeller ilk bakışta anlamlı ve tutarlı olsa bile aslolan istemcilerin o modeli nasıl kullanacağıdır.
- İstemcilerin üst arayüzler hakkında yaptıkları kabuller ve sahip oldukları bekłentiler, alt arayüzler tarafından da karşılanması gereklidir.

Uygulama



- Bir alt tipin, üst tipten devraldığı davranışı daha özel bir tip döndürecek şekilde override etmesi mümkün iken, davranışın parametrelerini daha alt tiplerle değiştirmesi mümkün değildir. Neden?
- Bunu **LSP/Design by Contract** ile açıklayınız.

Soru ve Cevap Zamani!



Interface Segregation Principle

Interface Segregation Principle



- **ISP, The Interface Segregation Principle ya da Arayüzleri Ayırma Prensibi**
- R. C. Martin tarafından Xerox'da çalışırken formule edilmiş ve C++ Report'un 1996 Haziran sayısında yayınlanmıştır.



Clients should not be forced to depend upon interfaces that they do not use.

İstemciler kullanmadıkları arayzlere bağımlı olmaya zorlanmamalıdır.

- **SRP**'nin, yüksek birliktelik amacıyla arayzlere uygulanmış halidir.
- **ISP**, şişman (fat) ya da kirli (polluted) arayzlerden kurtulmak, daha ince, odağı yüksek arayzlere sahip olmak gerektiğini ifade eder.
- Bir arayüzün farklı istemcilere hizmet veren metodlardan oluşan hizmet grupları varsa bu arayüz istemcilerine özel ince arayzlere bölünmelidir.



- **ISP** karşıtı durumun tipik iki göstergesi vardır:
 - Farklı istemcilerin aynı arayüzdeki farklı metotları çağırması,
 - Arayüzün alt tiplerinin bazı metotlara gerçekleştirmeye zorlanması.
- İlk durumda istemciler daha yüksek bir karmaşıklıkla karşılaşır,
- İkinci durumda ise sıra dışı durum fırlatarak ya da farklı yöntemlerle devralmak istediği metotlara bir gerçekleştirme veremeyen alt tipler, **LSP/Design by Contract**'e ters düşer ve istemcilerini RTTI'ye zorlar.

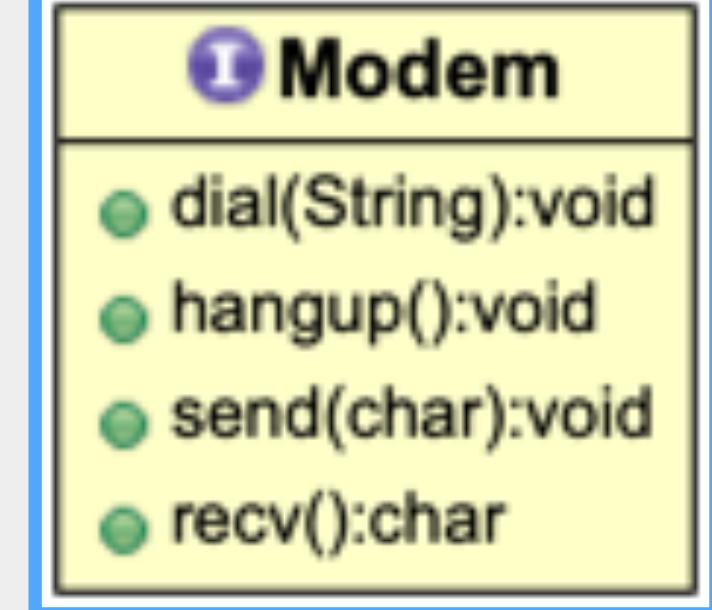


- Bu durumda bu arayüz bölünmelidir öyle ki istemciler sadece ilgilendikleri metotları görmeliler, alt sınıflar da sadece ihtiyacı olan metotları devralıp gerçekleştirmeye vermeliler.
- Böylece daha ince ve yüksek birliktelikli arayüzler ortaya çıkacaktır.

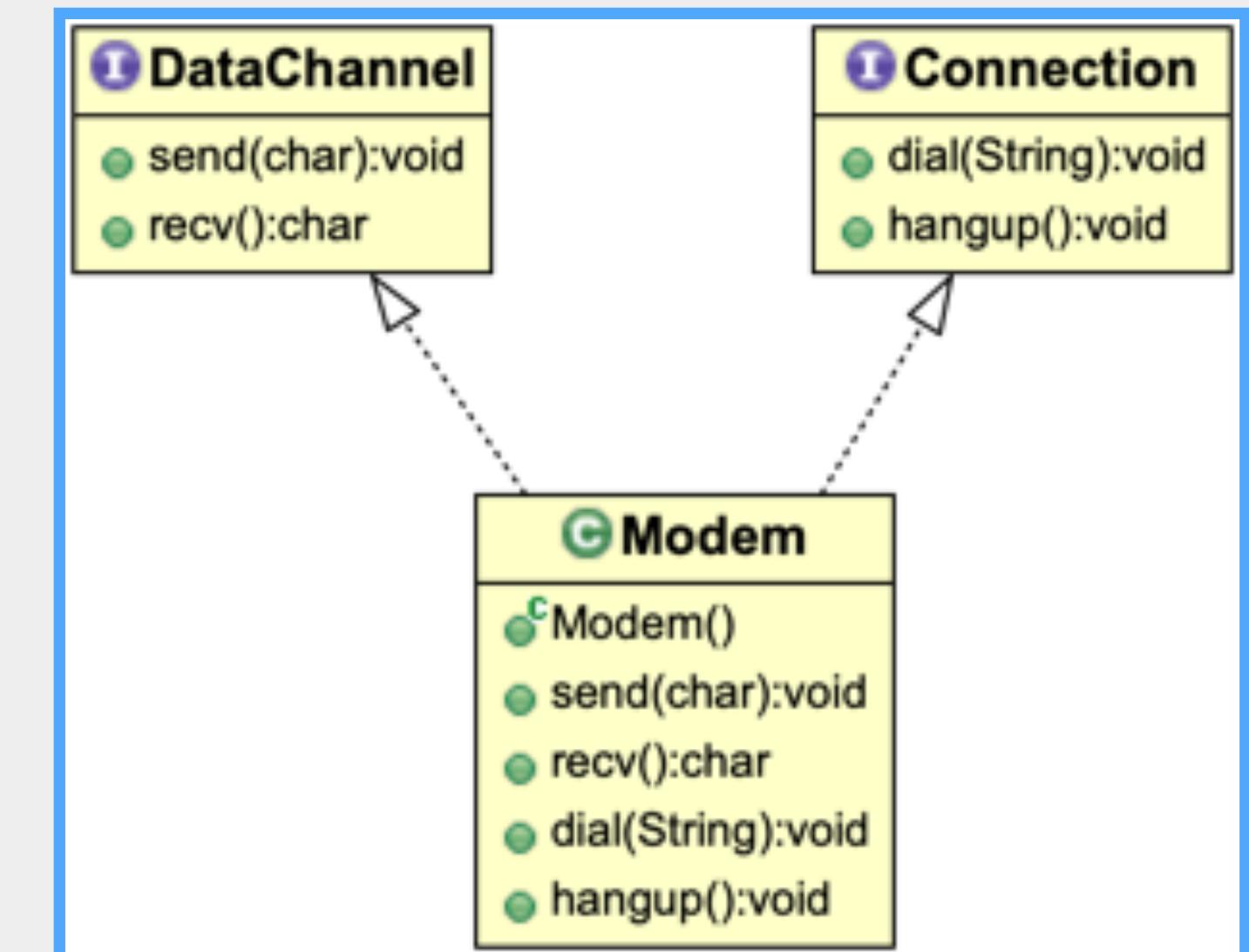
Modem



- Modem arayüzünün iki sorumluluğu vardır.
 - Dolayısıyla Modem arayüzü bölünmeli ve ancak birden fazla arayüzün gerçekleştirmesi olarak ifade edilmelidir.



ch03.srp.paper.comm1



ch03.srp.paper.comm2

Logger - I



- Log arayüzü, logların hem dosyaya hem de veri tabanına yapılacağı farzederek oluşturulmuştur.
- Bu durumda sadece veri tabanına ya da dosyaya loglama yapmak isteyen alt sınıflar bazı metodlara gerçekleştirmeye veremeyeceklerdir.
- Benzer şekilde bu arayüz ile sadece veri tabanına ya da dosyaya loglama yapmak isteyen istemciler ilgisiz metodları da görmek zorunda kalacaklardır.

I Logger

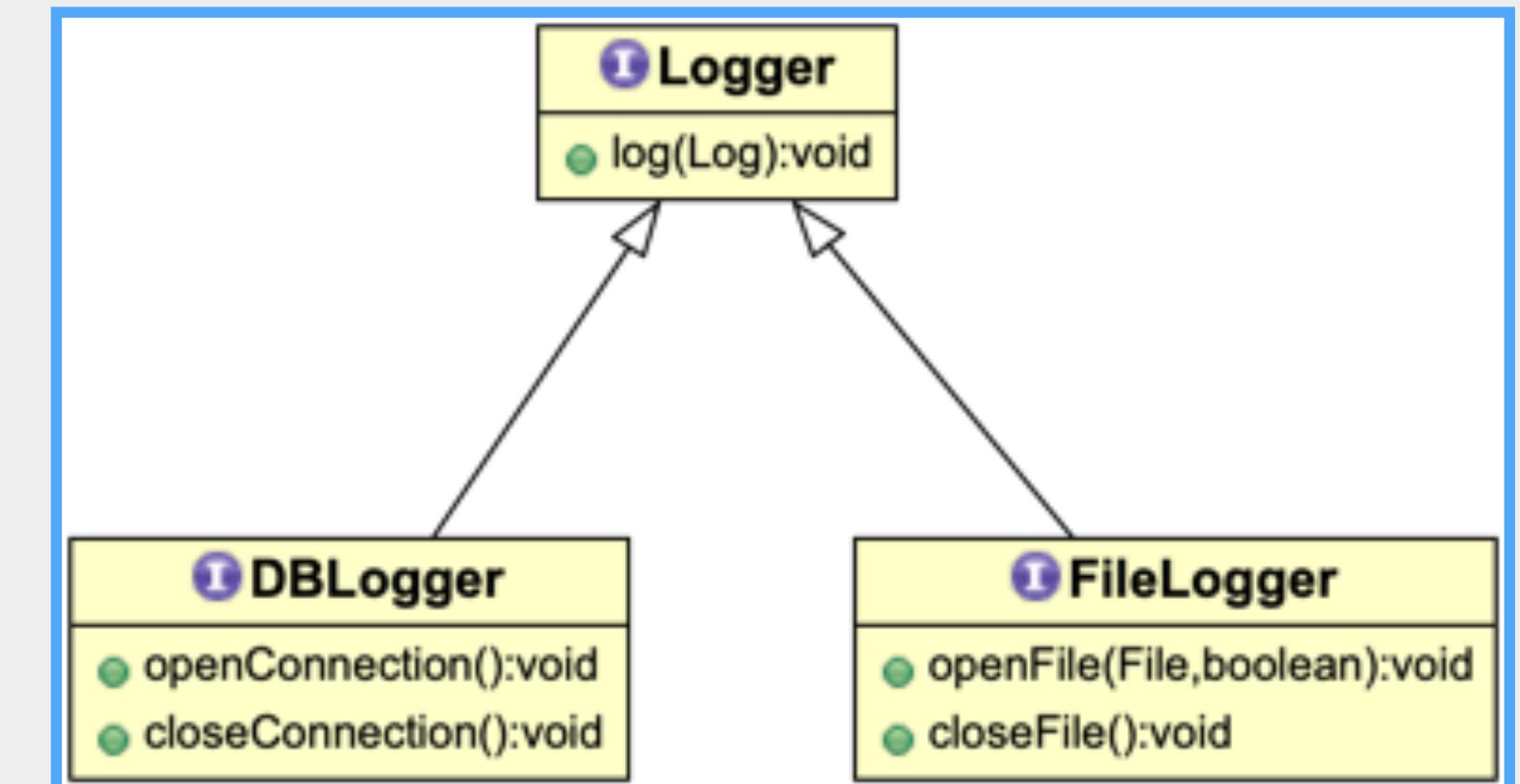
- log(Log):void
- openConnection():void
- closeConnection():void
- openFile(File,boolean):void
- closeFile():void

ch03.isp.log.bad

Logger - II



- Çözüm Log arayüzü birlikteliği yüksek iki arayüze bölmektir.



ch03.isp.log.good



- Daha önce de bahsettiğimiz gibi, ince arayüzlerden bir ya da daha fazlasını devralan sınıfların arayüzlerinin şişman ya da kirli olduğu düşünülebilir.
- Burada iki nokta vardır:
 - Bazen sınıfın tabiatı icabı birden fazla arayüzü bir araya getirmek kaçınılmazdır ve bu durum sınıf açısından tutarlı bir rol oluşturmaya yarar.
 - Çok sayıda ince arayüzün sağladığı fayda çok daha yüksektir.



- Öte taraftan istemciler bu türden nesneleri olabildiğince üst tipleri cinsinden göreceklerinden istemciler açısından gereksiz olan davranışlar saklanacak ve nesne hala ince bir arayüzle temsil edilecektir.

İş Nesnelerinin Arayüzü



- Veri veya veri tabanındaki sağlıksız veri modelinden yola çıkılarak oluşturulan iş nesnelerinde (entity) anemic domain modelinde bahsedilen problem yoksa, fazla sayıda alana bağlı olarak şişman arayüz problemi ortaya çıkabilir.
- Böyle durumlarda iş nesnelerinin, üzerindeki çok fazla iş mantığına bağlı olarak karmaşık arayüzleri olabilir.
- Bu türden iş nesneleri de hem **SRP** hem de **OCP**'den dolayı bölgünmesi gereklidir.

Uygulama

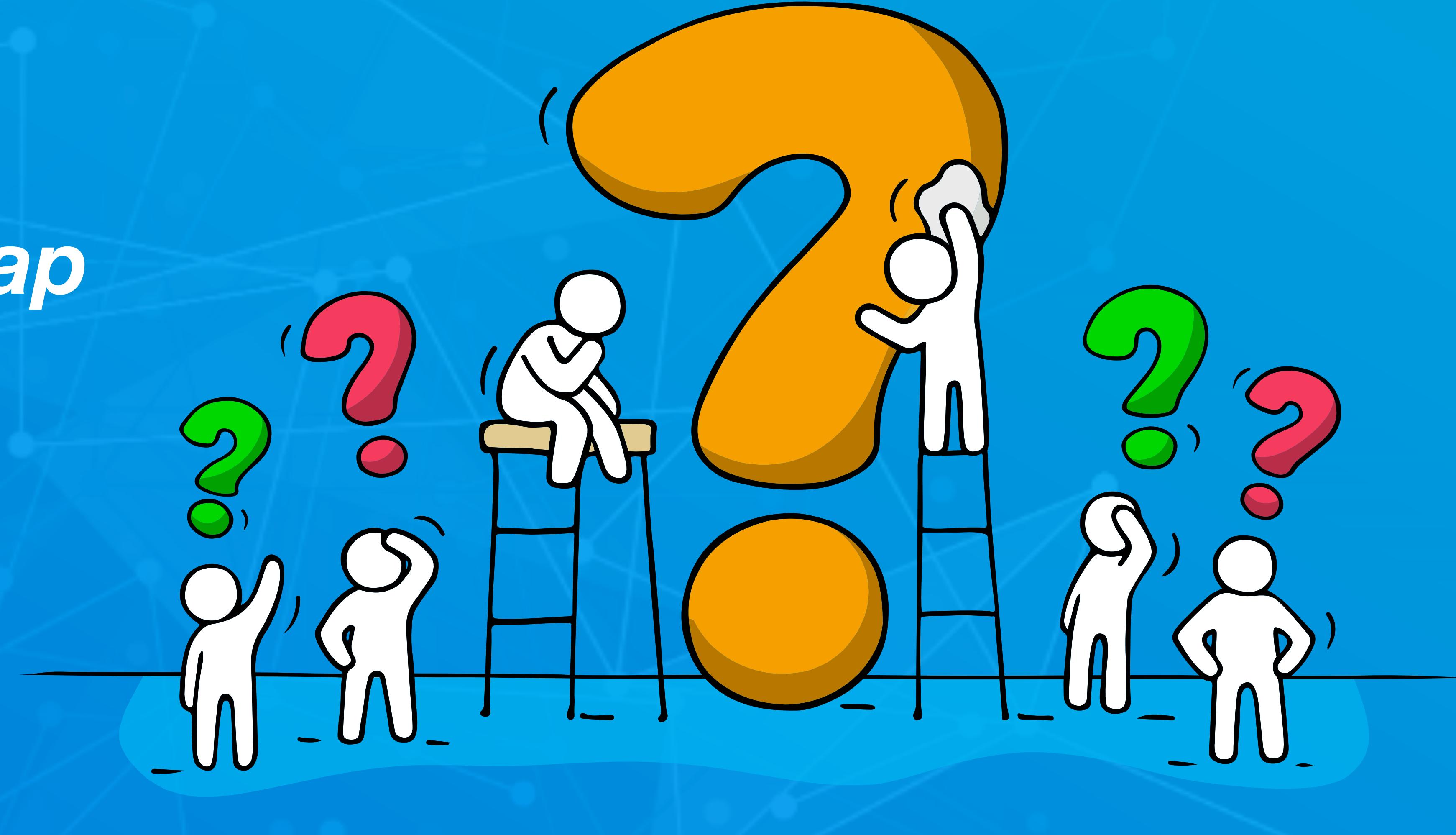


- Java'nın `java.util.Collection` ve C#'ın `System.Collections.ICollection`
- Bu iki arayüzü **ISP** açısından nasıl değerlendiririsiniz?

I Collection<E>
<ul style="list-style-type: none">● <code>size():int</code>● <code>isEmpty():boolean</code>● <code>contains(Object):boolean</code>● <code>iterator():Iterator<E></code>● <code>toArray():Object[]</code>● <code>toArray(T[]):T[]</code>● <code>toArray(IntFunction<T[]>):T[]</code>● <code>add(E):boolean</code>● <code>remove(Object):boolean</code>● <code>containsAll(Collection<?>):boolean</code>● <code>addAll(Collection<? extends E>):boolean</code>● <code>removeAll(Collection<?>):boolean</code>● <code>removeIf(Predicate<? super E>):boolean</code>● <code>retainAll(Collection<?>):boolean</code>● <code>clear():void</code>● <code>equals(Object):boolean</code>● <code>hashCode():int</code>● <code>spliterator():Spliterator<E></code>● <code>stream():Stream<E></code>● <code>parallelStream():Stream<E></code>

I Collection<T>
<ul style="list-style-type: none">● <code>Add(T):void</code>● <code>Clear():void</code>● <code>Contains():void</code>● <code>CopyTo(T[],Int32):void</code>● <code>GetEnumerator():IEnumerator</code>● <code>Remove(T):void</code>

Soru ve Cevap Zamani!



Dependency Inversion Principle

Dependency Inversion



- İlk defa 1996 yılında R. C. Martin tarafından C++ Report dergisinde aynı başlıklı makale ile ifade edilmiştir.
- Farklı kişilerce farklı şekillerde ifade edilmiştir.
- Muhtemelen SOLID'in en anlaşılır prensibidir.



Yüksek seviyeli modüller aşağı seviyeli modüllere bağımlı olmamalıdır. İkisi de soyutlamalara bağımlı olmalıdır. Soyutlamalar detaylara bağımlı olmamalı, detaylar soyutlamalara bağımlı olmalıdır.

**High level modules should not depend upon low level modules.
Both should depend upon abstractions. Abstractions should not depend upon details, details should depend upon abstractions.**

Kötü Tasarım Nedir? - I



- Bir tasarımı ne kötü (bad design) yapar?
- Martin'e göre kötü tasarlanmış bir yazılım parçasının üç göstergesi vardır:
 - **Katılık (rigidity)**: Değişiklik yapmak zordur çünkü değişiklikler yayılır.
 - **Kırılganlık (fragility)**: Değişiklikler pek çok yerde kırılmalar oluşturur.
 - **Taşınamazlık (Immobility)**: Asıl uygulama dışında başka uygulamalarda kullanılamaz.

Kötü Tasarım Nedir? - II



- Tüm bunların sebebi modüller arasındaki sağıksız bağımlılıklardır.
- Bağımlılıklar yoluyla değişiklikler dalga daga yayılır.
- Bağımlılıklar yoluyla en ufak bir değişiklik bile farklı kod parçalarının kırılmasına sebep olur.
- Ve modül, bağımlılıkları yüzünden farklı bir uygulamada kullanılamaz, aynı bağımlılıklara sahip olmak ister.

Yukarı-Aşağı Seviyeli İş ve Modül

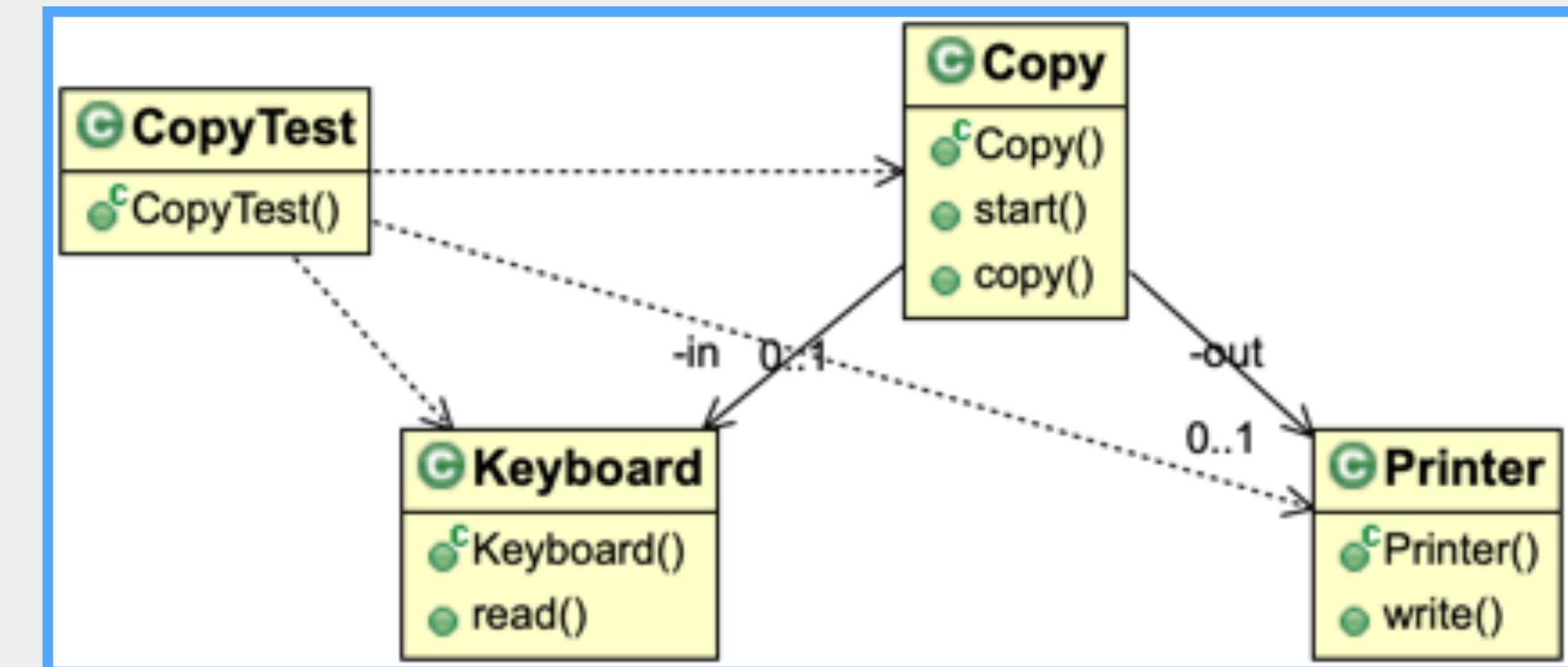


- **DIP**'yi anlamak için yukarı seviyeli ve aşağı seviyeli (high/low-level) iş/modül (policy/module) ayırmayı önemlidir.
- Yukarı seviyeli modül/iş ile kastedilen, uygulamayı oluşturan temel soyutlamalardır.
 - Süreçler ve onları yöneten (süreçsel metotlar) yapılar bu türdendir.
 - Alt seviyeli modül/iş ile kastedilen ise detaylardır.
 - Atomik iş yapan işçi metotlar ve sınıfları bu türdendir.

Copy - I



- **Copy**, **Keyboard**'dan okuyup **Printer**'a yazan bir kopyalama yapısıdır.
- **Copy** yukarı seviyeli bir iş yaparken **Keyboard** ve **Printer** detaydır.
- Bu yapıda aşağı seviyeli iki yapı, **Keyboard** ve **Printer**, tekrar kullanılabilir durumdadır.
- Ama **Copy** için aynı şey söylenemez çünkü kötü tasarımın üç göstergesine de sahiptir.

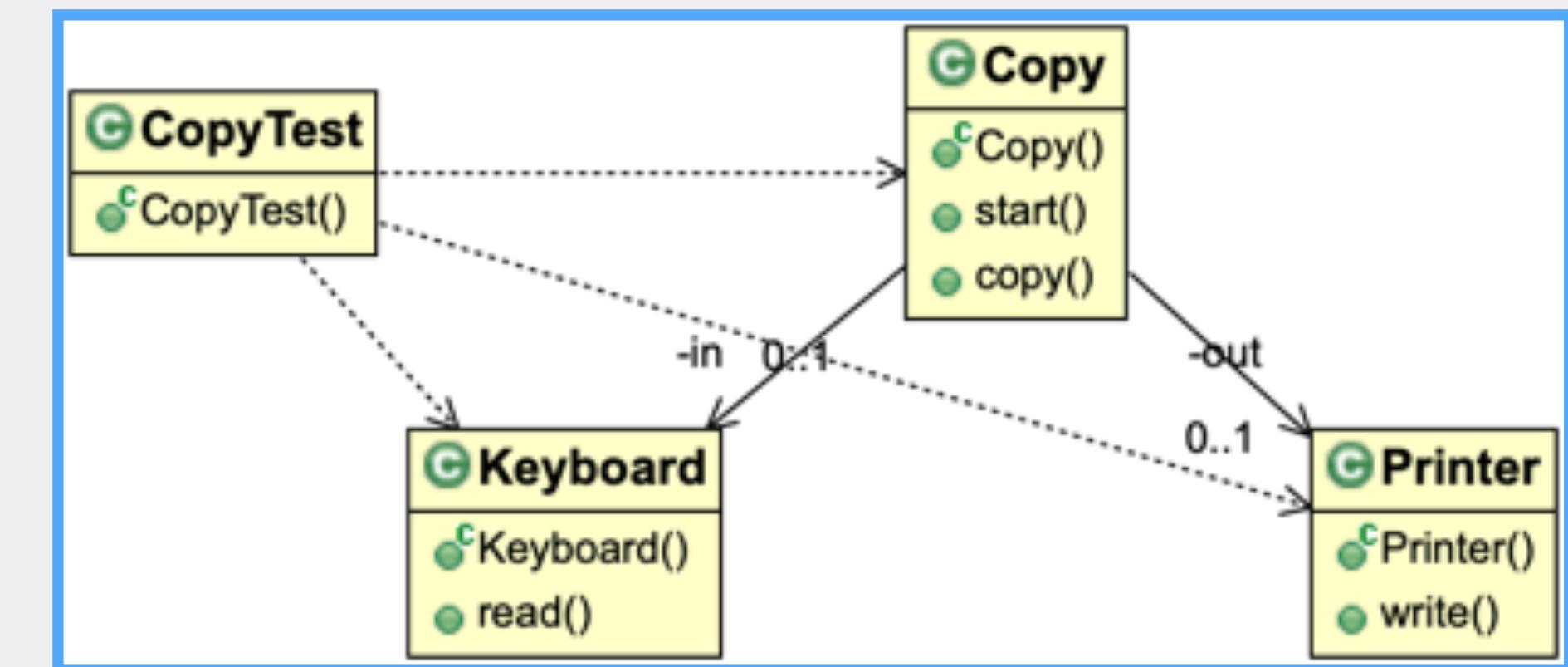


ch03.dip.copy.bad

Copy - II



- **Copy** hem katıdır, **Keyboard** ve **Printer** nesneleri olmadan çalışmaz, hem de kırılgandır, bu nesnelerdeki değişikliklerden etkilenme ihtimali yüksektir.
- **Copy** daima **Keyboard** ve **Printer** nesnelerine ihtiyaç duyar, başka nesnelerle çalışabilmesi için sürekli değişmesi gereklidir ki bu da **OCP**'ye aykırıdır.

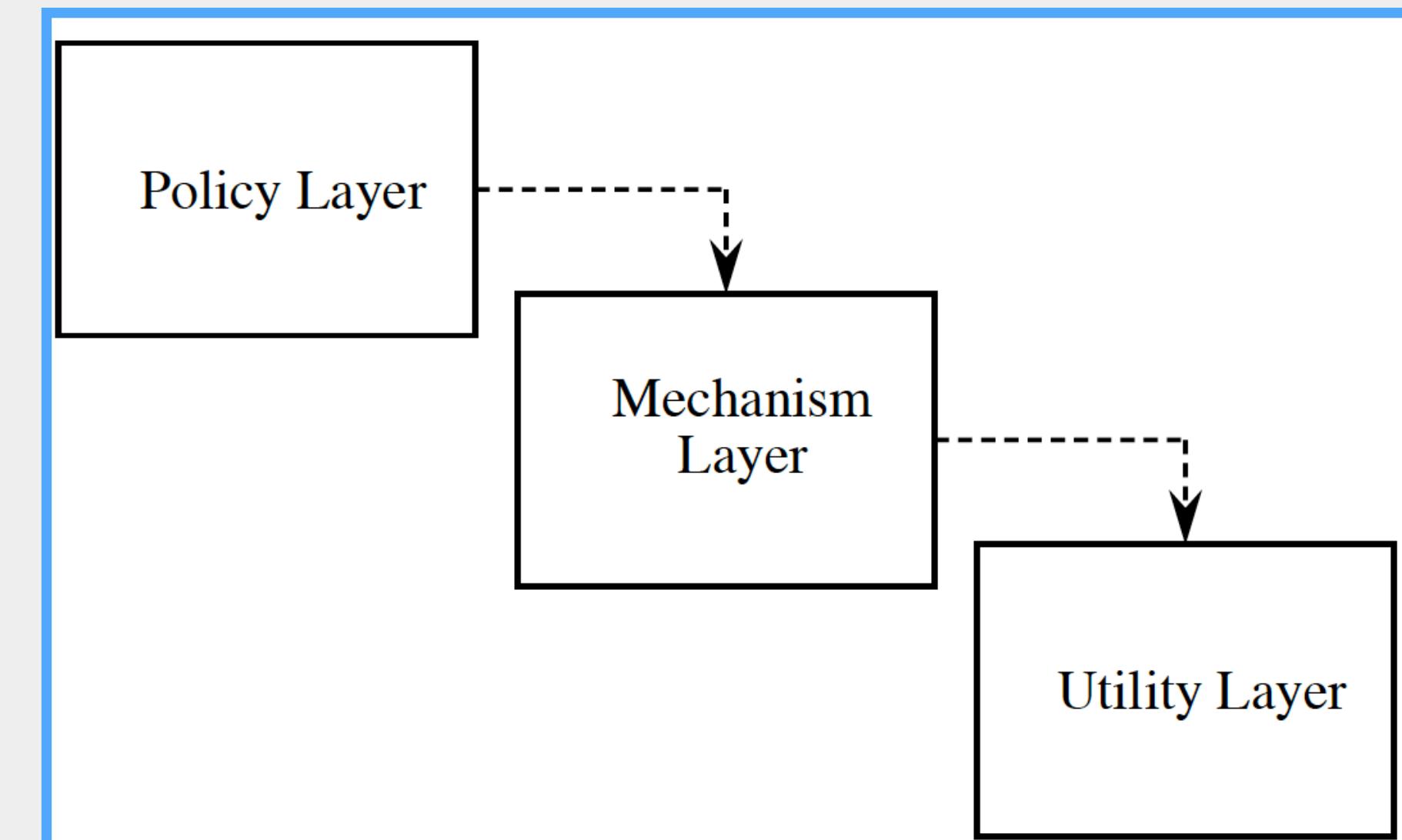


ch03.dip.copy.bad

Bağımlılıklar - I



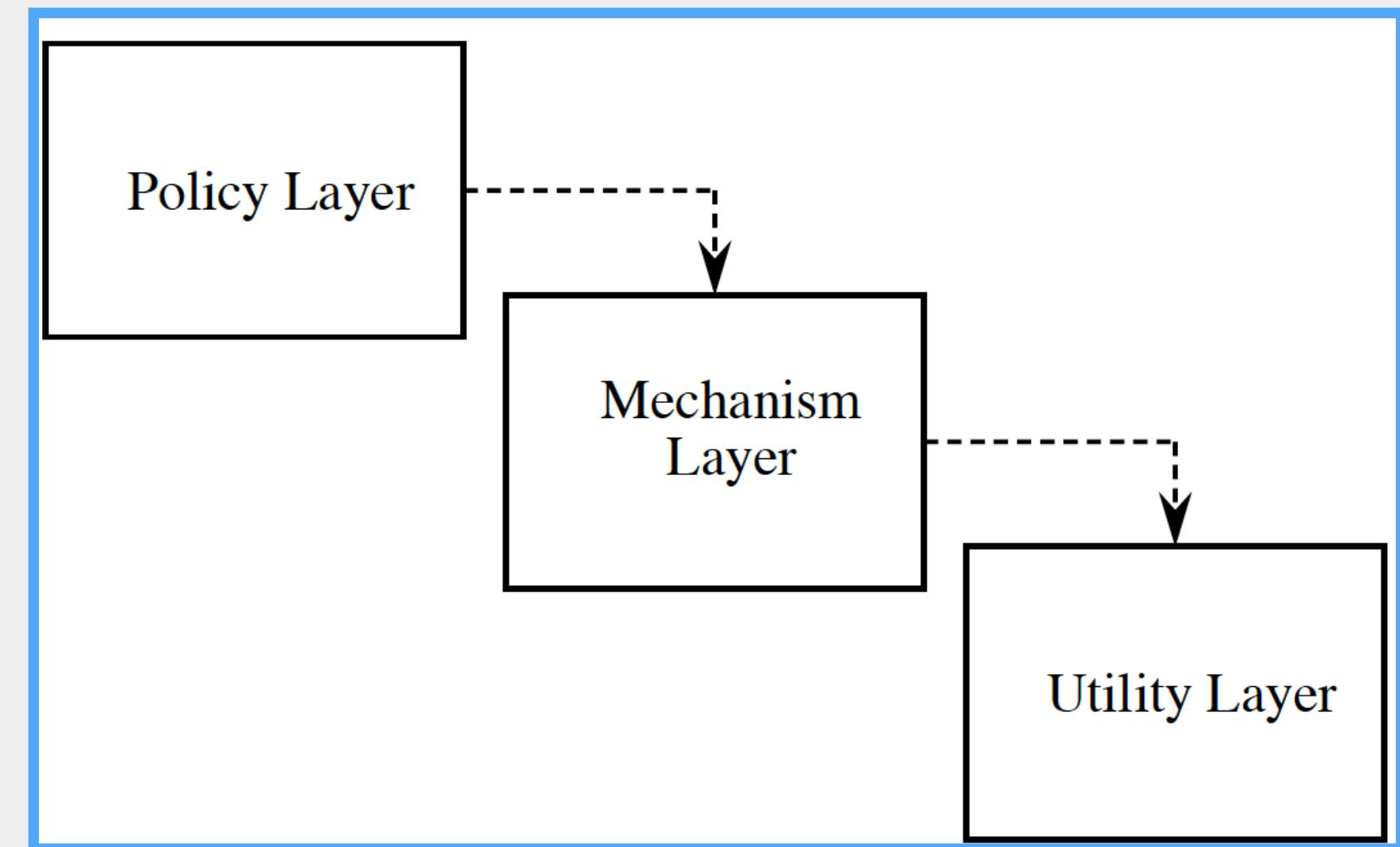
- Sıklıkla üst seviyeli iş yapan modüller, örneğin yönetsel metotlara sahip yapılar, alt seviyeli iş yapan modülleri, işçi metotlara sahip yapıları, iş nesnelerini ya da daha detay seviyede yönetsel metotlara sahip yapıları kullanırlar.
- Structured Analysis and Design gibi yaklaşımlarda üst modülerin alt modülleri, soyutlamaların detayları kullanmalarından dolayısıyla bağımlılıklarından bahsedilir.



Bağımlılıklar - II



- Buradaki en temel problem, geçişken olan bağımlılıklardan dolayı, alt seviye modüllerdeki değişikliklerin üst seviye modüllere sıçrama ihtimalidir.
- Halbuki olması gereken, yukarı seviye modüllerin değişimi belirlemede önceliğe sahip olmaları ve gerektiğinde alt seviye modülleri değişime zorlamalarıdır.
- Ayrıca tekrar kullanımda alt seviye modüller başarılıyken üst seviye modüller başarısızdır.



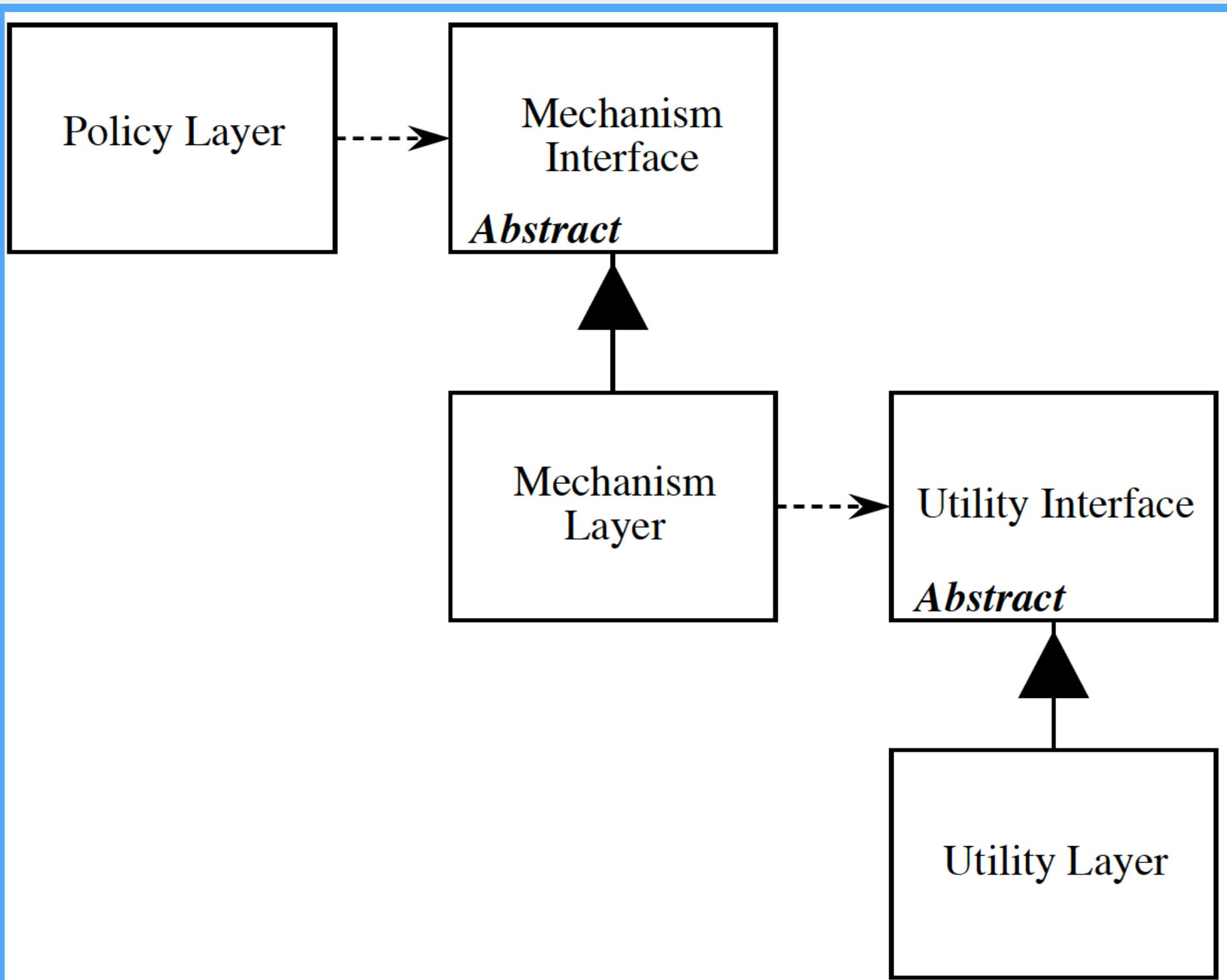


- Çözüm somut yapılar arasındaki bağımlılıkların tamamen soyut bağımlılıklara dönüşecek şekilde tersine çevrilmesidir.
- Bu amaçla her somut yapının soyut bir üst tipi oluşturulmalı ve yukarı seviyeli iş yapan yapıların bağımlılıkları soyut tiplere çevrilmelidir.
- Bu şekilde üst seviyeli soyut yapılar ile onların detayları arasında soyutlama tabakası konarak, değişimin yayılması önlenmelidir.

Bağımlılıklar - IV



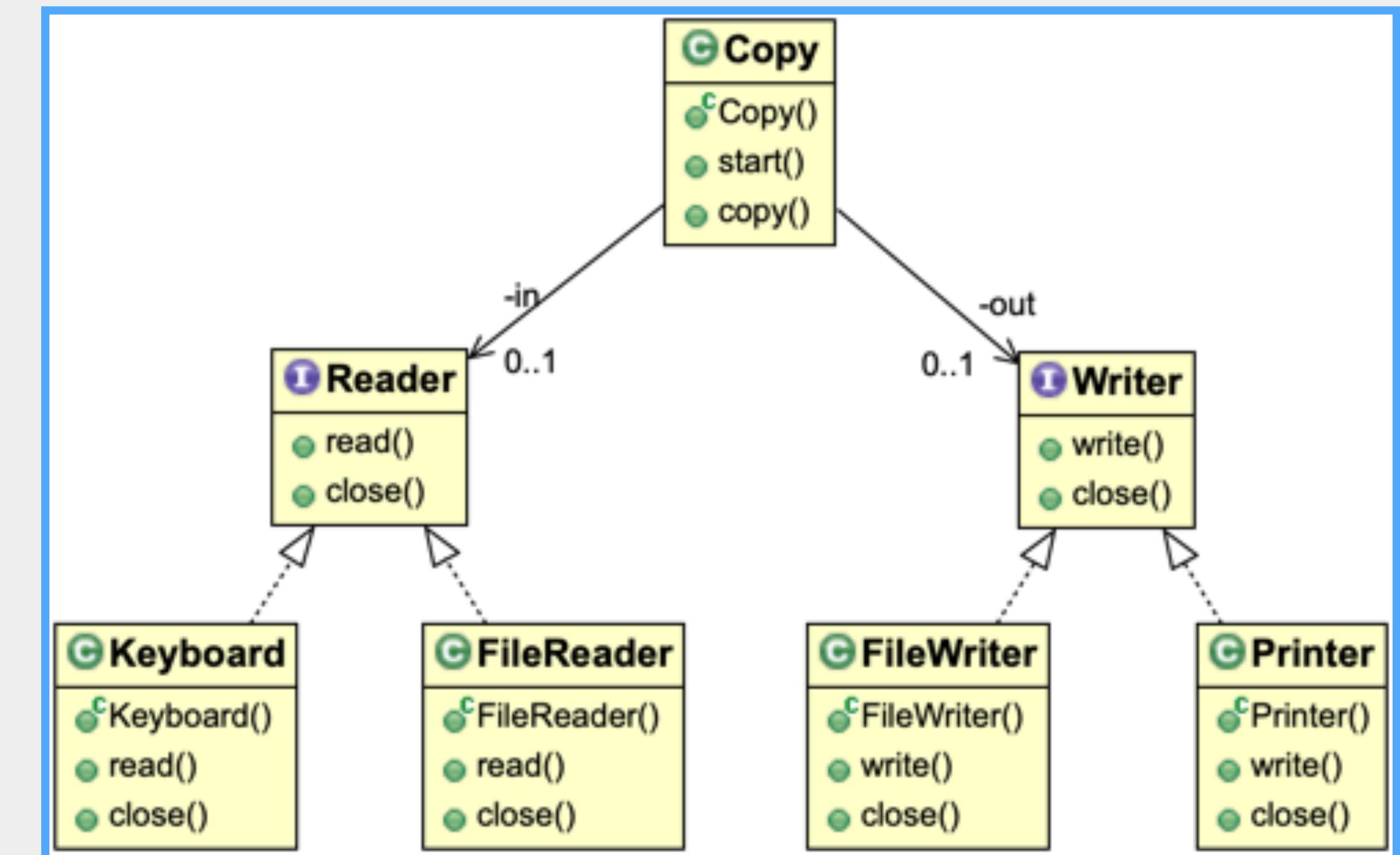
- Olması gereken yandaki modeldir.
- Her katman soyut bir arayüze temsil edilmeli ve üst katmanlar alt katmanlardan bu arayüz üzerinden hizmet almalıdır.
- Bu şekilde hiç bir katman doğrudan diğer katmana bağlı olmayacağıdır.
- Bağımlılıkların geçişkenliği de ortadan kalkacaktır.



Copy - III



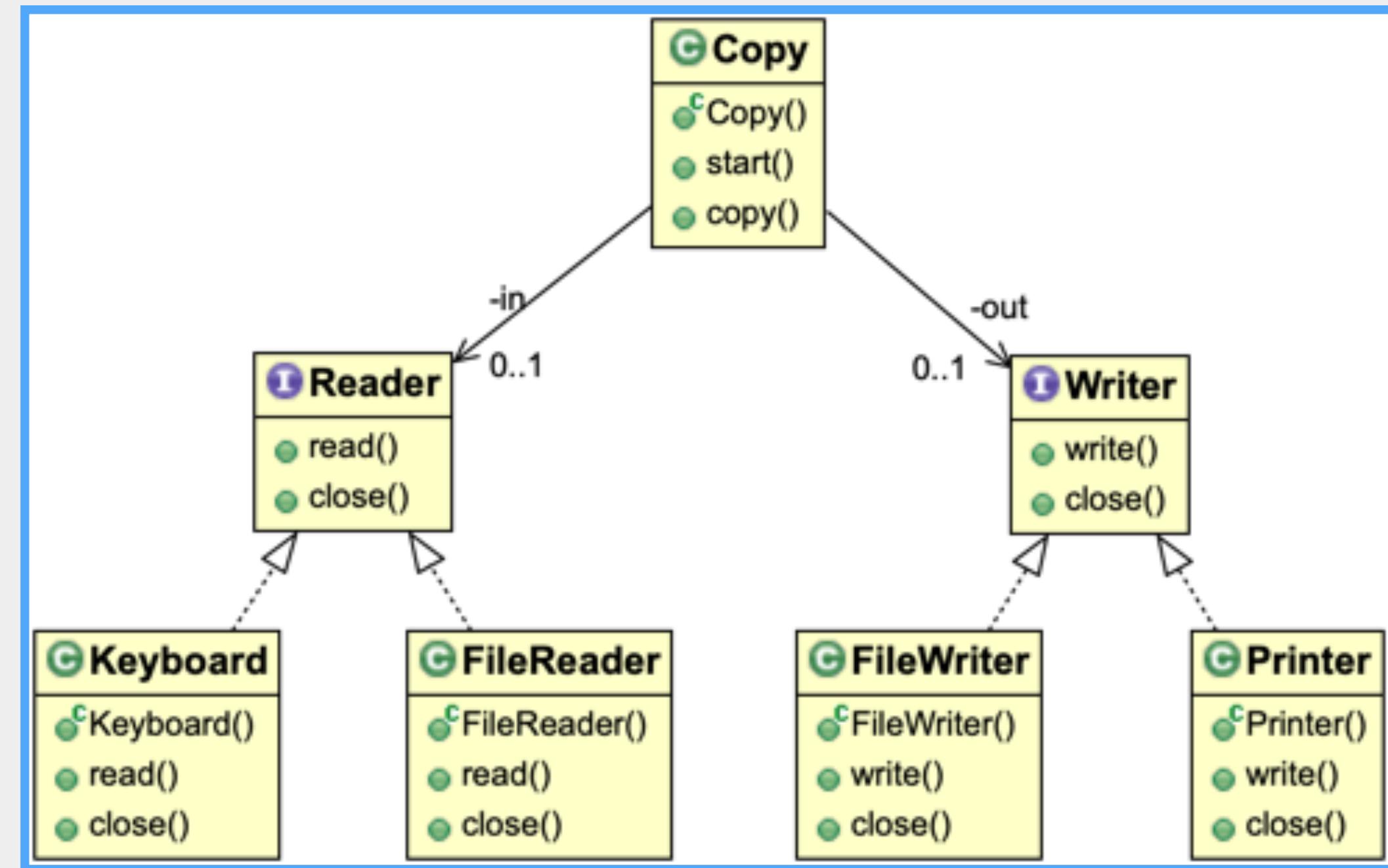
- Örnek yapı **DIP** ile yandaki modele dönüştürülebilir.
 - Bir öncekinden somut yapılara olan bağımlılıklar tersine çevrilerek tamamen soyut yapılara dayanacak hale getirilmiştir.
 - Bu durumda arayüzlerde değişiklik yapılmadığı müddetçe kötü tasarımın sonucu olan üç durumu görme ihtimali azalır.



ch03.dip.paper.solution

Copy - IV

- **Copy**'nin somut yapı olan **Keyboard** ve **Printer**'a olan bağımlılığı ortadan kalkmıştır.
- **Copy** böylece okuma ve yazma işlemi yapan detaylardan tamamen bağımsızdır, bunlardaki değişimlerden de yalıtılmıştır.
- **Copy**, **Keyboard** ve **Printer**' değil ama **Reader** ve **Writer** olan her uygulamada tekrar kullanılabilir.

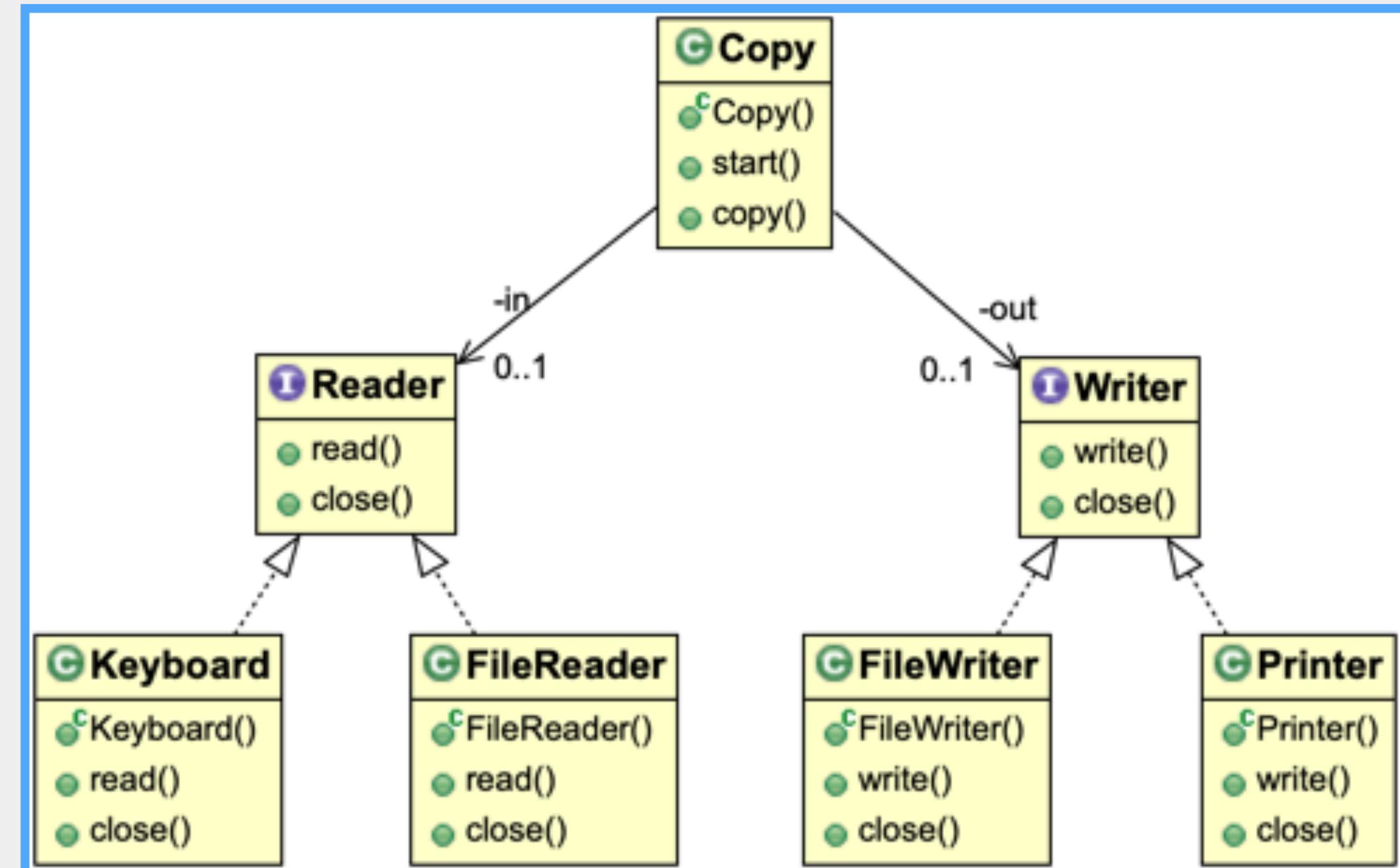


ch03.dip.paper.solution

Copy - V



- Bu çözümde de **Copy**'nin soyut da olsa **Reader** ve **Writer**'a olan bağımlılığı eleştirilebilir.
- Bu türden soyut bağımlılıklar **Adaptör** kalıbıyla rahatlıkla aşılabilir.
- Kaldı ki sıfır bağımlılık söz konusu değildir.



ch03.dip.paper.solution

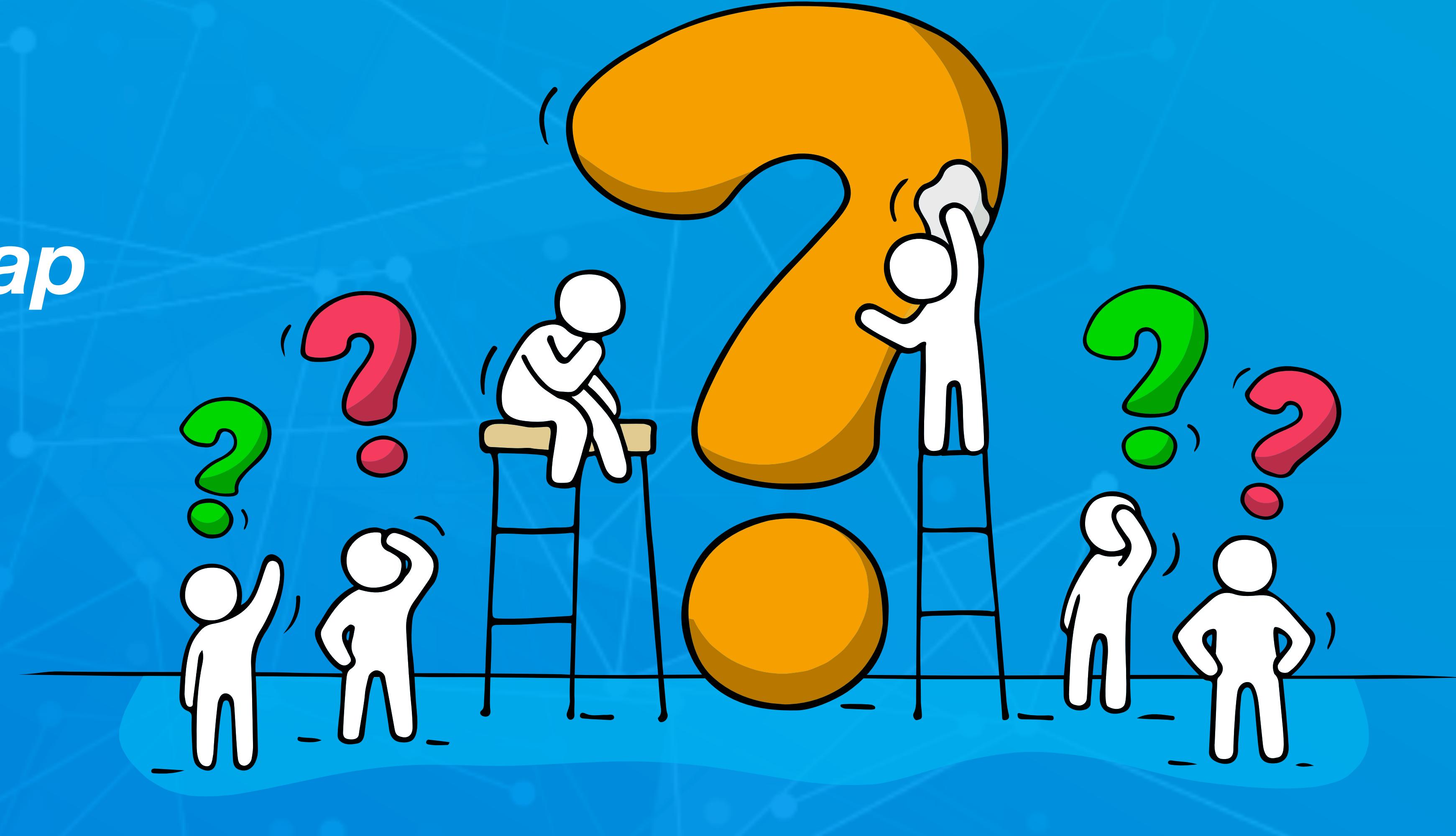


- DIP, soyutlamalar ile detaylarını birbirinden ayırarak bağımlılıkları dolayısıyla da değişimi yönetmeyi amaçlar.
- Bu şekilde yazılımların bakımlanabilirliği çok daha rahat hale gelecektir.



- Çok değişmesi beklenmeyen bir iş mantığının geliştirilmesi söz konusudur.
- İki yaklaşımından hangisini tercih edersiniz:
 - Doğrudan bir sınıfta ilgili metotlarla iş mantığını yerine getirmek,
 - Metotları önce bir arayüze koyup sonra onları gerçekleştiren bir sınıfta iş mantığı yerine getirmek.
- Her iki yaklaşımın da lehine ve aleyhine olan argümanlar neler olabilir?

Soru ve Cevap Zamani!



Online Kaynaklar

Online Kaynaklar - I



- https://lostechies.com/wp-content/uploads/2011/03/pablos_solid_ebook.pdf
- <https://blog.ndepend.com/defense-solid-principles/>
- <https://docs.microsoft.com/en-us/archive/msdn-magazine/2014/may/csharp-best-practices-dangers-of-violating-solid-principles-in-csharp>
- <https://softwareengineering.stackexchange.com/questions/155628/what-is-the-difference-between-single-responsibility-principle-and-separation-of>



- <https://www.tonymarston.net/php-mysql/not-so-solid-oo-principles.html>
- <http://qualityisspeed.blogspot.com/2014/08/why-i-dont-teach-solid.html>
- <https://news.ycombinator.com/item?id=8929458>
- <https://codeblog.jonskeet.uk/2013/03/15/the-open-closed-principle-in-review/>

Diğer Prensipler

En Az Bilgi Prensibi



- Nesnelerin birbirlerini bilmeleri kaçınılmaz olduğundan, bilmenin keyfiyetinin öne çıktığını daha önce belirtmiştık.
- Keyfiyeti de “arayüz” olarak belirlemiştik. Yani,
 - Nesneler birbirlerini arayüzleri üzerinden bilmeliler,
 - Nesneler birbirlerinin gerçek tiplerini değil, en genel arayüz tipini bilmeliler.
 - Nesneler birbirlerinin gerçekleştirmeye detaylarını bilmemeliler.

Hangi Nesneler Bilinmeli



- Nesnelerin birbirlerini bilme konusunda ele alınması gereken bir başka nokta ise, bir nesnenin bilebileceği nesnelerin hangileri olduğuyla ilgilidir.
- Prensip şudur:
 - Bir nesne olabildiğince az sayıda nesne bilmeli,
 - Bir nesne, işini yapmak için kendisinden hizmet alması kaçınılmaz olan az sayıda nesneyi bilmeli.
- Bu, **Demeter Kanunu (Law of Demeter)** ya da **En Az Bilgi Prensibi (Principle of Least Knowledge)** olarak da bilinir.

En Az Bilgi Prensibi - I



- Bu prensibe göre bir nesnenin metodlarında, kendisi üzerinde metod çağrıları yapabileceği nesneler ancak şunlar olabilir:
 - O nesnenin instance variableları,
 - O nesnenin metodlarına geçilen nesneler,
 - O nesnenin o metodunda oluşturulan nesneler.
- Yani bir nesne ancak arkadaşlarıyla konuşur, yabancılarla konuşmaz!

En Az Bilgi Prensibi - II



```
public class A{  
    private B b;  
    public void f(C c){  
        b.g(); // 1- Yapılabilir  
        c.u(); // 2- Yapılabilir  
        D d = new D();  
        d.v(); // 3- Yapılabilir  
        E e = c.w(); // Yapma bunu!!!  
        e.z(); // E'den iş isteme, C'den, E'den iş  
    }  
}
```



En Az Bilgi Prensibi - III



```
public class Car{  
    private Engine engine;  
    public void go(City city){  
        engine.start(); // 1- Yapılabilir  
        int distane = c.getDistance(); // 2- Yapılabilir  
        Gas gas = new Gas(distance)  
        gas.discard(); // 3- Yapılabilir  
        Mayor m = city.getMayor(); // Yapma bunu!!!  
        m.howAreYou(); // !!!  
    }  
}
```

En Az Bilgi Prensibi - III



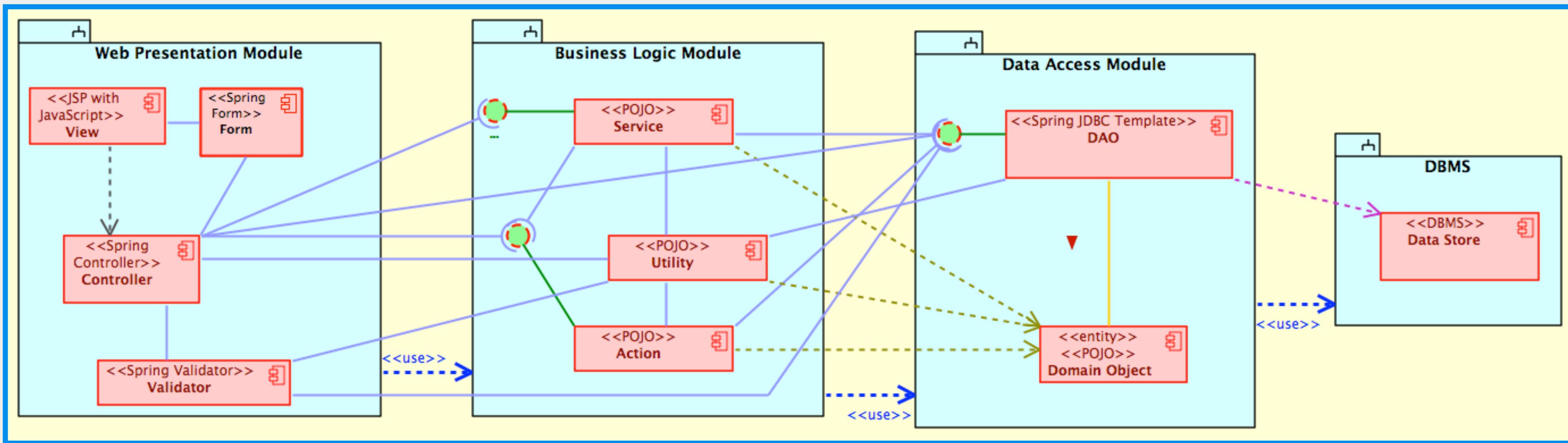
- Demeter Kanunu, en temelde şunu söyler:
 - Her yazılım birimi, diğer birimler hakkında sınırlı bilgiye sahip olmalı.
 - Bu sınırlı bilgi, sadece ve sadece birimin yakınındaki birimlerle ilgili olmalı.
 - Her birim sadece ve sadece “arkadaşlarıyla” konuşmalı, “yabancılarla” konuşmamalı.

Don't talk to strangers!

Mimari Düzen



- Demeter Kanunu, sadece sınıf düzeyinde değil, mimari düzeyde de geçerlidir.
- Buna uyulmadığında daha karmaşık ve zor değişen mimariler ortaya çıkar.



Uygulama

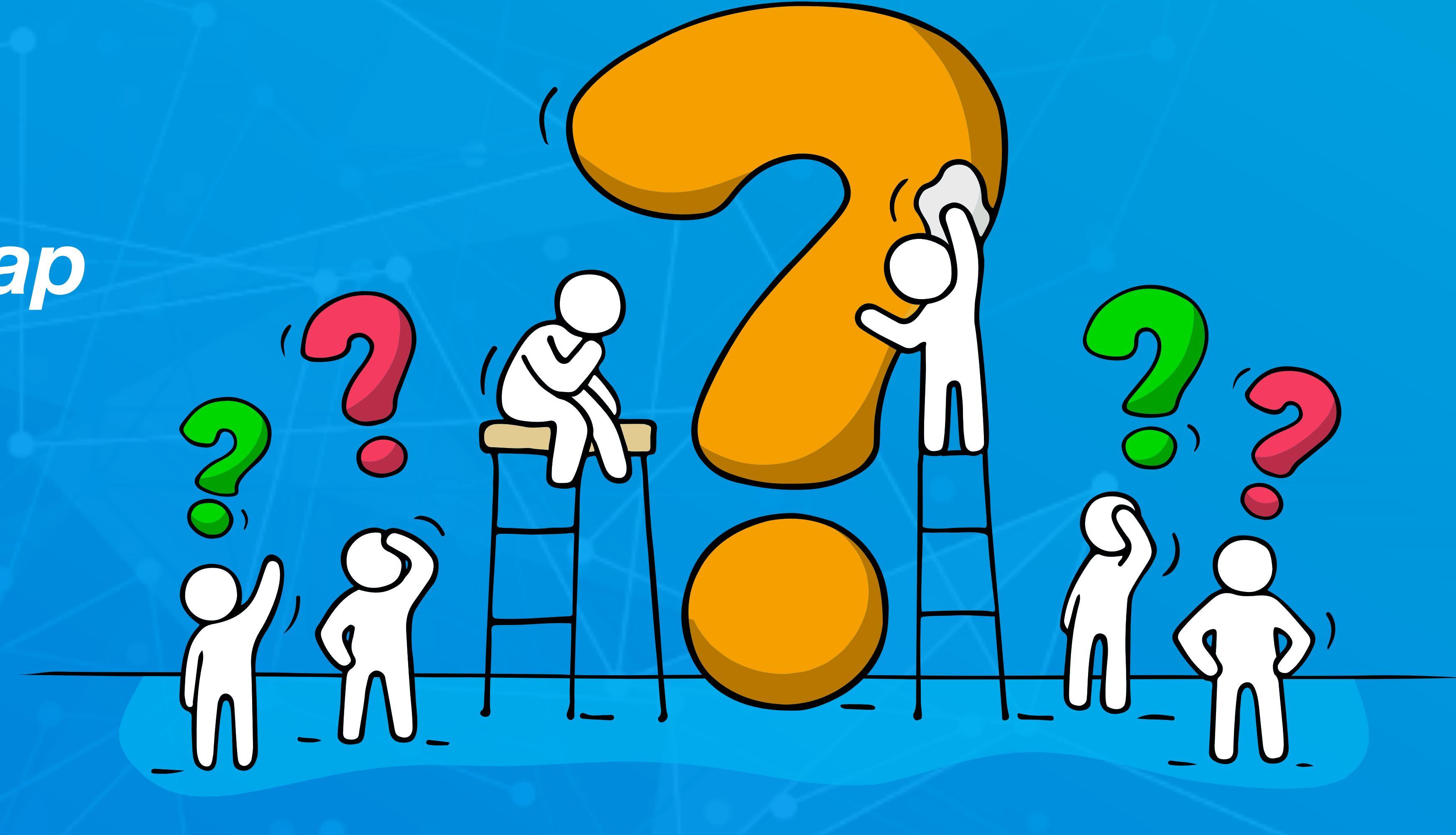


- En az bilgi prensibine ters iki örnek senaryo verin, öyle ki birisi kötü bir uygulama iken diğerinin makul bir uygulama olsun.



- En az bilgi prensibi bize genel olarak her nesnenin ancak ve ancak, kendi sorumluluğunu yerine getirmek için tabi olarak sahip olduğu ya da bildiği nesnelerle haberleşebileceğini söyler.
- Ayrıca bu prensip bize dolaylı olarak nesnelerin sorumluluklarını yerine getirmek için istediği nesnelerden hizmet alması gerektiğini, bilgi (başka bir nesne) almaması gerektiğini söyler.

Soru ve Cevap Zamani!

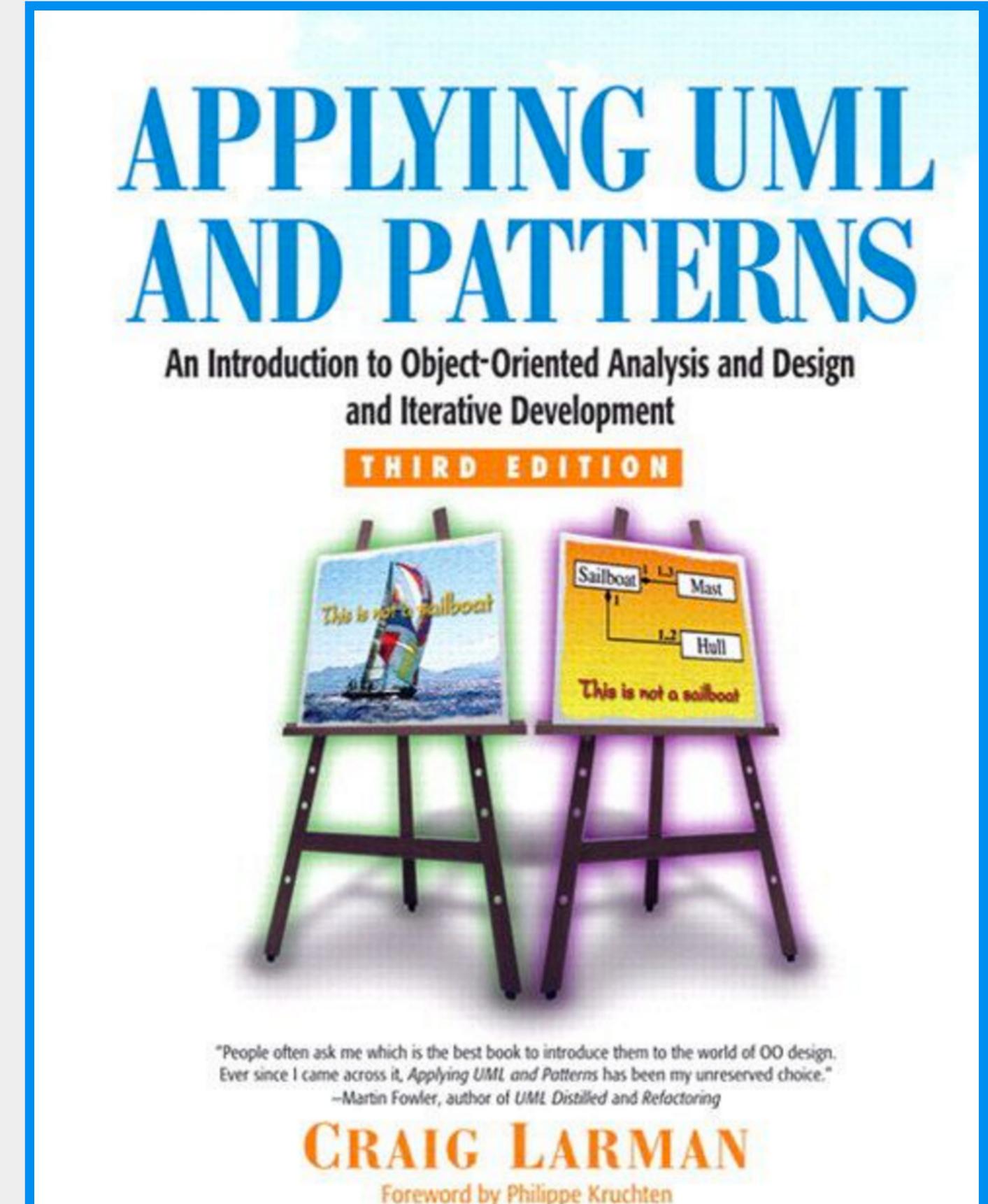


GRASP





- GRASP, “General Responsibility Assignment Software Patterns”ın kısaltmasıdır.
- Craig Larman, “Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development” isimli kitabında anlatmaktadır.
- Kitap giriş seviyesinde bir kitaptır.





- Larman kitabının 3. baskısının 216. sayfasında şöyle demektedir:

The GRASP patterns are a learning aid to help one understand essential object design, and apply design reasoning in a methodical, rational, explainable way. This approach to understanding and using design principles is based on patterns of assigning responsibilities.

- Sorumlulukları bulmak ve onları nesnelere atamak en temel hedeftir.

Sorumluluklar



- Larman sorumlulukları (responsibilities) ikiye ayırır:
 - **Knowing**
 - knowing about private encapsulated data
 - knowing about related objects
 - knowing about things it can derive or calculate
 - **Doing**
 - doing something itself, such as creating an object or doing a calculation
 - initiating action in other objects
 - controlling and coordinating activities in other objects



- **GRASP** patternları şunlardır:
 - Information Expert
 - Creator
 - High Cohesion
 - Low Coupling
 - Controller
 - Polymorphism
 - Pure Fabrication
 - Indirection
 - Protected Variations



- **Pattern Name:** Information Expert
- **Problem It Solves:** What is a basic principle by which to assign responsibilities to objects?
- **Solution:** Assign a responsibility to the class that has the information needed to fulfill it.
- Bu prensip, temel nesne tanımını (encapsulation) vurgular ve veri bağımlılıklarını ortadan kaldırmayı amaçlar.

Creator - I



- **Pattern Name:** Creator
- **Problem It Solves:** Who should be responsible for creating a new instance of some class?

Creator - II



- **Solution:** Assign class **B** the responsibility to create an instance of class **A** if one or more of the following is true:
 - **B** aggregates **A** objects.
 - **B** contains **A** objects.
 - **B** records instances of **A** objects.
 - **B** closely uses **A** objects.
 - **B** has the initializing data that will be passed to **A** when it is created (thus **B** is an **Expert** with respect to creating **A**).

Creator - III



- **B** is a creator of **A** objects.
- If more than one option applies, prefer a class **B** which aggregates or contains class **A**.
- This is alternative to **Factory** design pattern

Low Coupling



- **Pattern Name:** Low Coupling
- **Problem It Solves:** How to support low dependency, low change impact, and increased reuse?
- **Solution:** Assign a responsibility so that coupling remains low.
 - Bu prensip de **Information Expert**'e benzer şekilde temel nesne tanımını (encapsulation) vurgular ve veri bağımlılıklarını ortadan kaldırmayı amaçlar.
 - Veriyi gezdirmek yerine en yakın yerde işlemeyi tavsiye eder.

High Cohesion



- **Pattern Name:** High Cohesion
- **Problem It Solves:** How to keep complexity manageable?
- **Solution:** Assign a responsibility so that cohesion remains high.
- Fonksiyonel birlikteliği vuyrgular.

Controller - I



- **Pattern Name:** Controller
- **Problem It Solves:** Who should be responsible for handling an input system event?

Controller - II



- **Solution:** Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:
 - Represents the overall system, device, or subsystem (facade controller).
 - Represents a use case scenario within which the system event occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <Use-CaseName>Session (use-case or session controller).
 - Use the same controller class for all system events in the same use case scenario.



- Informally, a session is an instance of a conversation with an actor.
- Sessions can be of any length, but are often organized in terms of use cases (use case sessions).
- Corollary: Note that "window," "applet," "widget," "view," and "document" classes are not on this list.
- Such classes should not fulfill the tasks associated with system events, they typically receive these events and delegate them to a controller.

Controller - IV



- Normally, a controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity.
- It does not do much work itself.

Bloated Controller - I



- Poorly designed, a controller class will have low cohesion - unfocused and handling too many areas of responsibility; this is called a bloated controller. Signs of bloating include:
- There is only a single controller class receiving all system events in the system, and there are many of them. This sometimes happens if a facade controller is chosen.
- The controller itself performs many of the tasks necessary to fulfill the system event, without delegating the work.

Bloated Controller - II



- A controller has many attributes, and maintains significant information about the system or domain, which should have been distributed to other objects, or duplicates information found elsewhere.
- There are several cures to a bloated controller, including:
 - Add more controllers system does not have to have only one. Instead of facade controllers, use use-case controllers.
 - Design the controller so that it primarily delegates the fulfillment of each system operation responsibility on to other objects.

Polymorphism



- **Pattern Name:** Polymorphism
- **Problem It Solves:** How to handle alternatives based on type? How to create pluggable software components?
- **Solution:** When related alternatives or behaviors vary by type (class), assign responsibility for the behavior—using polymorphic operations—to the types for which the behavior varies.
- Corollary: Do not test for the type of an object and use conditional logic to perform varying alternatives based on type.

Pure Fabrication - I



- **Pattern Name:** Pure Fabrication
- **Problem It Solves:** What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate?

Pure Fabrication - II



- **Solution:** Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept—something made up, to support high cohesion, low coupling, and reuse.
- Such a class is a fabrication of the imagination. Ideally, the responsibilities assigned to this fabrication support high cohesion and low coupling, so that the design of the fabrication is very clean, or pure—hence a pure fabrication.
- Finally, a pure fabrication implies making something up, which we do when we're desperate!

Indirection



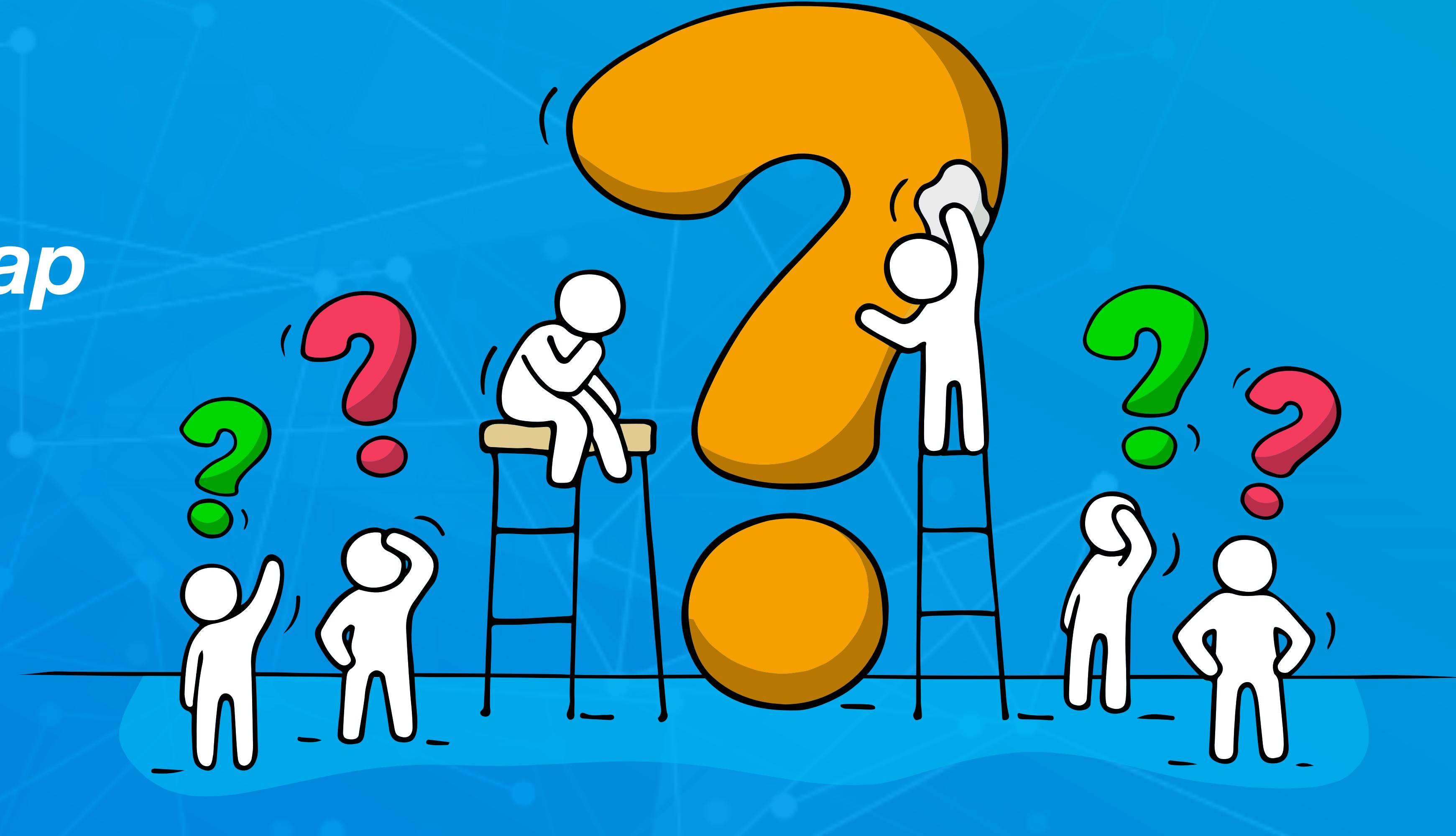
- **Pattern Name:** Indirection
- **Problem It Solves:** Where to assign a responsibility, to avoid direct coupling between two (or more) things? How to de-couple objects so that low coupling is supported and reuse potential remains higher?
- **Solution:** Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled. The intermediary creates an indirection between the other components.

Protected Variations



- **Pattern Name:** Protected Variations
- **Problem It Solves:** How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
- **Solution:** Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

Soru ve Cevap Zamani!





Ödevler



1.“An Empirical Comparison of Modularity of Procedural and OO Software” başlıklı makaleyi inceleyip tartışın.

Bölüm Sonu

*Soru ve Cevap
Zamani!*

