

## NESNEYE DAYALI TASARIM VE MODELLEME

Eylül 2017

Yrd.Doç.Dr. Yunus Emre SELÇUK

### GENEL BİLGİLER

#### DERS İÇERİĞİ

- Kısım 1: Gang of Four Tasarım Kalıpları
- Kısım 2: Code Smells, Anti patterns and Refactoring (Birim sınamaları ile)

#### KAYNAKLAR

- Kısım 1:
  - Design Patterns – Elements of Reusable OO Software, Erich Gamma et.al (Gang of Four), Addison-Wesley, 1994
  - Ek kaynaklardan da çalışılması yararlı olacaktır (sonraki yansısı)
- Kısım 2:
  - Refactoring: Improving the Design of Existing Code, Martin Fowler. Addison-Wesley, 1999
  - Anti Patterns: Refactoring Software, Architectures and Projects in Crisis, William H. Brown et.al. Wiley, 1998

1

## NESNEYE DAYALI TASARIM VE MODELLEME

#### KAYNAK KİTAPLAR

- Ek kaynaklar (Elektronik ortamda bulunuyorlar):
  - Design Patterns Explained:A New Perspective on Object-Oriented Design, Alan Shalloway and James R. Trott. Addison-Wesley, 2004 (2nd Ed.)
  - Head First Design Patterns, Elisabeth Freeman et.al, O'Reilly, 2004
  - Design Patterns (Wordware Applications Library), Christopher G. Lasater. Jones & Bartlett Publishers, 2006 (1st ed.)
  - Design Patterns Java Workbook, S. John Metsker, Addison-Wesley, 2002
  - Applied Java Patterns, S. Stelting and O. Maassen, Prentice Hall, 2002
  - ... ve sizin bulup bana önerdiğiniz kaynaklar

#### GEREKSİNİMLER

- Herhangi bir güncel NYP diline hakimiyet
  - Derste kod örnekleri java dili üzerinden verilecektir.
- UML sınıf şemalarına hakimiyet
  - Bu konuda çok eksik öğrenciler olabiliyor. Lütfen birinden inceleyiniz:
    - UML Distilled, 3rd ed. (2003), Martin Fowler, Addison-Wesley.
    - UML 2.0 in a Nutshell, Dan Pilone and Neil Pitman, O'Reilly.
    - Vb.

2

## NESNEYE DAYALI TASARIM VE MODELLEME

### DEĞERLENDİRME

- 1. Ara sınav: %20, Tasarım kalıpları konulu, 8. hafta (10 Kasım 2017)
- 2. Ara sınav: %25, Tasarım kalıpları konulu, 13. hafta (15 Aralık 2017)
- Vize haftalarında değişiklikler olabilir, bölüm web sayfasından duyuruları izleyiniz.
- Proje ödevi: %15 (Seçimlik, Antipatterns hakkında, verilemezse ağırlığı ara sınavlara dağıtıllacak)
- Ara sınav telafi: 15. hafta (29 Aralık 2017)
- Final sınavı: %40, Refactoring konulu, Final haftasında
- Bütünleme sınavı: Final sınavı telafisidir. Bütünleme haftasında

3

## NESNEYE DAYALI TASARIM VE MODELLEMEYE GİRİŞ

### YAZILIM KALİTESİ

- 'İyi' tasarım ile 'kaliteli' yazılıma ulaşmak amaçlanır.
  - Problem alanı hangi düzeyde parçalara ayrılabilirse, o düzeyde parçalara ayrılabilen bir tasarım da 'iyi' olarak nitelendirilebilir.
- Ölçütler:
  - Dış kalite özellikleri (kullanıcıya yönelik)
  - İç kalite özellikleri (programcıya yönelik)
- Dış kalite ölçütleri:
  - Doğruluk, etkinlik, kolaylık, ...
- İç kalite özellikleri:
  - Yeniden kullanabilirlik
  - İlgi alanlarının ayrılması
  - Esneklik
  - Taşınabilirlik
  - Okunabilirlik ve anlaşılabilirlik
  - Sınamabilirlik
  - Bakım kolaylığı
  - ...

4

## NESNEYE DAYALI TASARIM VE MODELLEMEYE GİRİŞ

### YAZILIM KALİTESİ

- Neden 'iyi' bir tasarım?
  - Yazılım projeleri önemli oranda başarısızlığa uğramaktadır.
  - Etken sayısı şüphesiz fazladır, ancak 'tasarımın iyiliği' denetlenebilen bir etkendir.
  - Oluşan bir hatayı düzeltmenin bedeli, yazılım hayat döngüsünde ilerlendikçe üstel olarak artmaktadır.
  - Yazılım geliştirme çabalarının çoğu, bakım aşamasında harcanmaktadır.
- İyi bir tasarımına götüren temel ilkeler:
  - Düşük bağılaşım (Low coupling)
  - Yüksek uyum (High cohesion)
  - İlgi alanlarının ayrılması (Separation of concerns)
    - İlk iki ilke ile açıklanabilir.

5

## NESNEYE DAYALI TASARIM İLKELERİ

### DÜŞÜK BAĞLAŞIM – LOW COUPLING

- Bağılaşım: Bir parçanın diğer parçalara bağımlılık oranı.
  - Parça: Sınıf, alt sistem, paket
- Bağımlılık: Bir sınıfın diğerinin:
  - Hizmetlerinden yararlanması,
  - İç yapısından haberdar olması,
  - Çalışma prensiplerinden haberdar olması,
  - Özelleşmiş veya genelleşmiş hali olması (kalıtım ilişkisi).
- Diğer sınıfların sayısı arttıkça bağılaşım oranı artar.
- Düşük bağılaşımın yararları:
  - Bir sınıfta yapılan değişikliğin geri kalan sınıfların daha azını etkilemesi,
  - Yeniden kullanılabilirliğin artması

6

## NESNEYE DAYALI TASARIM İLKELERİ

### YÜKSEK UYUM – HIGH COHESION

- **Uyum:** Bir parçanın sorumluluklarının birbirleri ile uyumlu olma oranı.
- **Yüksek uyumun yararları:**
  - Sınıfın anlaşılma kolaylığı artar.
  - Yeniden kullanılabilirlik artar.
  - Bakım kolaylığı artar
  - Sınıfın değişikliklerden etkilenme olasılığı düşer.
- **Genellikle:**
  - Düşük bağlaşım getiren bir tasarım yüksek uyumu,
  - Yüksek bağlaşım getiren bir tasarım ise düşük uyumu beraberinde getirir.

7

## NESNEYE DAYALI TASARIM İLKELERİ

### İLGİ ALANLARININ AYRILMASI

- İlgi alanlarının ayrılması ile daha iyi bir tasarıma ulaşılması:
  - Yazılımın sadece belirli bir amaç, kavram veya hedef ile ilgili kısımlarının tanımlanabilmesi, birleştirilebilmesi ve değiştirilebilmesi yeteneğine işaret eder.
  - Sorunu parçalayarak fethet ve parçaları yeniden kullan!
  - Her programlama yaklaşımı, sorunu parçalara bölmek için yollar sunar.
  - Parçaların düşük bağlaşımı ve yüksek uyuma sahip olması istenir.

8

**NESNEYE DAYALI TASARIM VE MODELLEME  
KISIM 1: TASARIM KALIPLARI  
1.0. TASARIM KALIPLARINA GİRİŞ**

1

**TASARIM KALIPLARINA GİRİŞ**

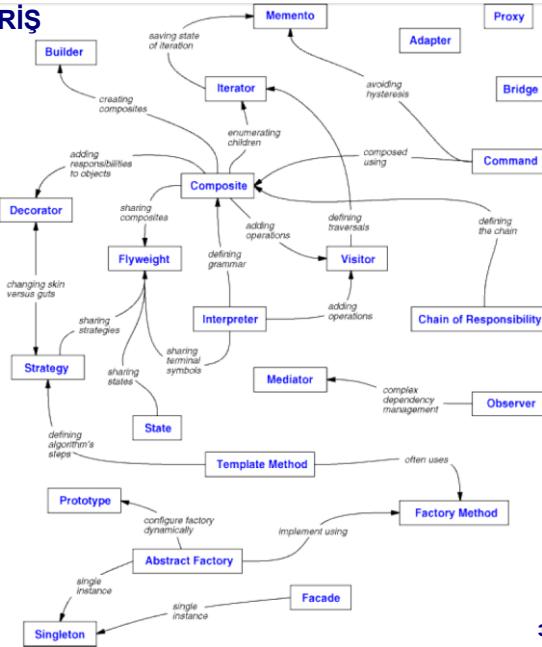
**TASARIM KALIBI (DESIGN PATTERN) NEDİR?**

- Yazılım tasarımda karşılaşılan belirli sorunların yalın ve güzel çözümlerinin tarifidir.
  - Çözüm somut bir gerçekleme olmak zorunda değildir, soyut bir tarif de olabilir.
- Amaç: Tekerleği yeniden keşfetmeden 'iyi' tasarıma ulaşmak.
  - Sorunların esnek biçimde ve yeniden kullanılabilirliği artıracak yönde çözülmesi.
- Bir tasarım kalibinin ana bileşenleri:
  - İsim: Kalibi en güzel şekilde anlatabilir bir veya birkaç kelimeyle isim.
  - Sorun: Kalının çözüldüğü sorunun tarifi.
  - Çözüm: Sorunu çözmek için önerilen tasarımın bileşenleri; bu bileşenlerin ilişkileri, sorumlulukları, birlikte çalışmaları.
  - Tartışma/Sonuçlar: Çözümün sistemin geneline esneklik, genişletilebilirlik, taşınabilirlik gibi yönlerden yaptığı etkiler; çözümün güçlü ve zayıf yönleri, yer-zaman çelişkileri, başka kalıplar ile işbirliği olanakları, vb.

2

## TASARIM KALIPLARINA GİRİŞ

- Kalıplar arasında olası ilişkiler:



## TASARIM KALIPLARINA GİRİŞ

### TASARIM KALIBİ (DESIGN PATTERN) NEDİR?

- Yararları:
  - Tekerleği yeniden icat etmemek.
  - Üzerinde birçok deneyimli kişinin emeği geçtiğinden, 'kaliteli' bir çözüm.
  - Deneyimin yazılı biçimde aktarılabilmesi.
  - Yazılım ekibi arasında ortak bir sözlük oluşturması.
- Tartışma:
  - Bir kişinin 'kalıp' dediğine, diğeri 'basit bir yapı taşı' veya 'temel bir ilke' diyebilir.

## TASARIM KALIPLARINA GİRİŞ

### TASARIM KALIBİ (DESIGN PATTERN) NEDİR?

- Ders notlarındaki kalıplar ve anlatım sırası hakkında:
  - Ders notlarında anlatılan GoF tasarım kalıpları, kaynak kitaptaki sıralamaya sadık kalınarak anlatılmıştır.
  - GoF kalıpları amaçlarına göre 3 kümeye ayrılmıştır:
    - Creational: Yaratımsal. Nesnelerin oluşturulması, temsili ve ilişkilendirilmelerindeki bağımlılığı azaltmaya yönelik kalıplardır.
    - Structural: Yapısal. Sınıflar ve nesnelerin bir araya getirilerek daha büyük yapılar elde edilmesine yönelik kalıplardır.
    - Behavioral: Davranışsal. Sorumlulukların nesnelere dağıtıması ve algoritma seçimine yönelik kalıplardır.

5

## TASARIM KALIPLARINA GİRİŞ

### TASARIM KALIBİ (DESIGN PATTERN) NEDİR?

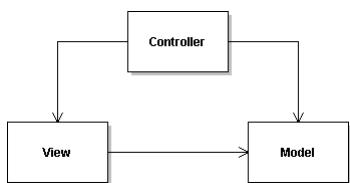
- Ders notlarındaki kalıplar ve anlatım sırası hakkında (devam):
  - GoF kalıplarının bazlarının odağı sınıflar, bazlarının odağı ise nesnelerdir.
    - Sınıfsal kalıplar: Sınıflar ve bunların alt sınıfları arasındaki ilişkilere yönelik kalıplardır. Sınıf yapısı ve kalıtım ilişkisinin doğası gereği derleme zamanında belirlenen ve daha durağan ilişkilerdir.
    - Nesnesel kalıplar: Çeşitli türden nesneler arasındaki ilişkilere yönelik kalıplardır. Kodun çalışması süresince rahatlıkla değiştirilebilir ve daha dinamik ilişkilerdir.
    - Adapter kalıbı GoF tarafından hem sınıfsal hem de nesnesel bir kalıp olarak değerlendirilmiştir. Diğer kalıplar bu iki gruptan sadece birine dahil edilmiştir.

6

## TASARIM KALIPLARINA GİRİŞ

### ÖRNEK TASARIM KALIBI: MODEL-VIEW-CONTROLLER (MVC)

- Sorun:
  - Aynı veriyi farklı biçimlerde gösterebilmek istiyoruz.
  - Kullanıcıya, verinin nasıl ve ne kadarının gösterileceğini seçebilme gibi olanaklar sağlamak istiyoruz.
- Çözüm:
  - Veriyi, verinin gösterimini ve gösterimin yönetimini ayrı sınıflar şeklinde ele almak.



- Model sınıfı:
  - Ham veriyi simgeler.
- View sınıfı:
  - Verinin çeşitli gösterim biçimleri.
- Controller sınıfı:
  - Kullanıcı komutlarını alıp işler.

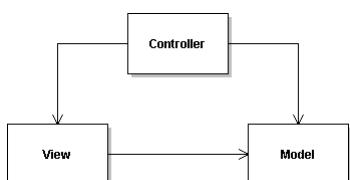
- Benzetim:
  - Bir ressam ve çizdiği insan.
  - Başka?

7

## TASARIM KALIPLARINA GİRİŞ

### MVC TASARIM KALABI

- Yararlar:
  - Gösterim, kontrol ve veriyi birbirlerinden ayırarak;
  - Bunları çalışma anında değiştirebilmek.
  - İlgi alanlarının ayrılmasını sağlamak.
  - Yeniden kullanımı artırmak.
- Çeşitlemeler:
  - Model'in geçirdiği değişiklikleri View'a bildirmesi
  - Birden fazla view:
    - İç içe (nested).
    - Ayrı/tek görevcikte (thread)
    - Örtüşen/ayrik
  - Birden fazla kontrolcü:
    - Farklı denetim biçimleri (klavye, fare, ses, ...)



8

**NESNEYE DAYALI TASARIM VE MODELLEME  
KISIM 1: TASARIM KALIPLARI  
1.1. YARATIMSAL (CREATIONAL) KALIPLAR**

1

**YARATIMSAL (CREATIONAL) KALIPLAR**

**ABSTRACT FACTORY**

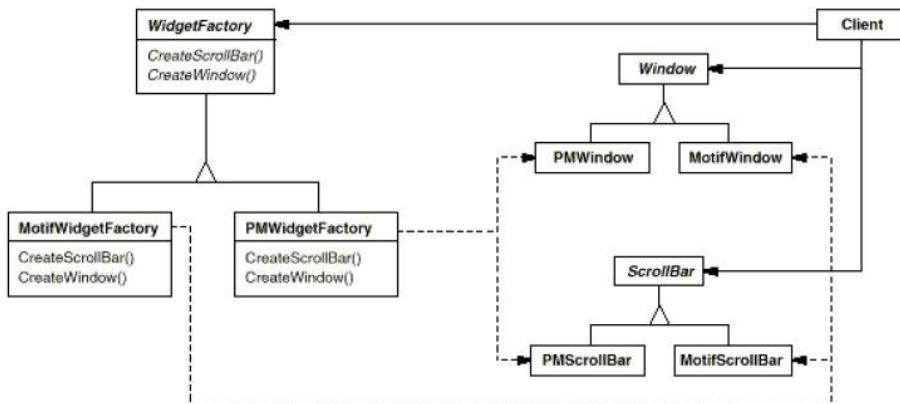
- Amaç:
  - Birbirleri ile ilişkili ya da aynı aileden olan nesneleri, sınıflarını belirtmeden oluşturabilmek.
- Örnek:
  - Java'da çeşitli GUI bileşenlerinin görünüşünün (look-and-feel) çalışma alanında değiştirilebilmesi
- Sorun:
  - Taşınabilirliği sağlamak için, istekçinin hangi tür görünüşün hangi sınıfından nesnelerle gerçekleştiğini bilmemesi gereklidir.

2

## YARATIMSAL (CREATIONAL) KALIPLAR

### ABSTRACT FACTORY:

- Örnek Çözüm:



3

## YARATIMSAL (CREATIONAL) KALIPLAR

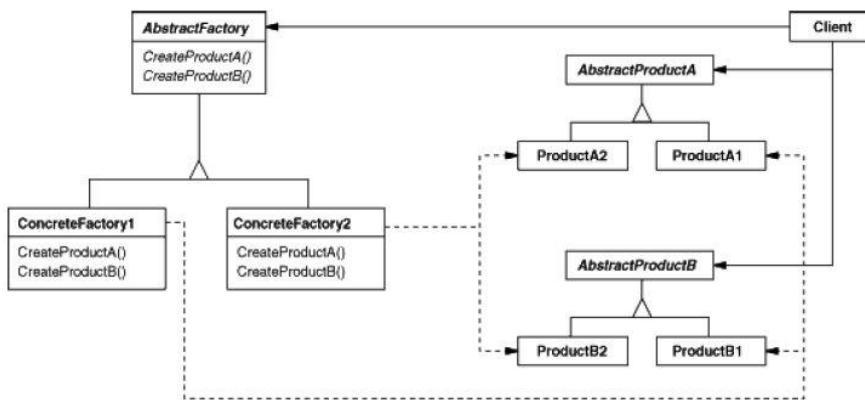
### ABSTRACT FACTORY:

- Diğer seçenekler:
  - Switch – Case:
    - WidgetFactory.setLookAndFeel( val: EnumLookFeel )
    - switch( val ) { ... }
    - Zayıf yönü: Yeni görünüş biçimleri için yeni enum değerleri ve switch içerisine yeni case komutları eklemek gerek.
  - Başka?

4

### ABSTRACT FACTORY:

- Kalıp yapısı:



- Kalıp bileşenleri:

- `ProductXY`: Oluşturulacak nesneler = ürünler
- `AbstractProductX`: X ürünlerinin arayüzlerini tanımlar
- `ConcreteFactoryY`: Y ürünlerini oluşturur
- `AbstractFactory`: Ürün oluşturmak için gerekli komutların imzalarını içerir.

5

### YARATIMSAL (CREATIONAL) KALIPLAR

#### ABSTRACT FACTORY:

- Kalıbın uygulanabileceği anlar:
  - Bir sistemin, kendisine ait ürünlerin nasıl oluşturulduğu, birleştirildiği ve sunulduğundan bağımsız olmasının istediği durumlar
  - Bir sistemin farklı ailelerden herhangi birindeki nesneler tarafından yapılandırılması gerektiğinde
  - İlişkili ürün ailelerinden oluşan nesnelerin birlikte kullanılması gerektiğinde,
  - Bir ürün ailesindeki nesnelerin bir sınıf kütüphanesi olarak sunulacağı, ancak gerçeklemeleri yerine yalnızca ara yüzlerinin açığa çıkarılacağı durumlarda

6

## ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

### ÇÖZÜLECEK PROBLEM:

- Bir 3D oyun motoru hazırlıyoruz.
- Kullanılacak işletim sistemi ve ekran kartına göre OpenGL veya DirectX kütüphanesinin renderer, shader, vb bileşenleri kullanılacak.

- Kaynak: YES

7

## ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

### ÇÖZÜM 1: Switch – Case Kullanımı

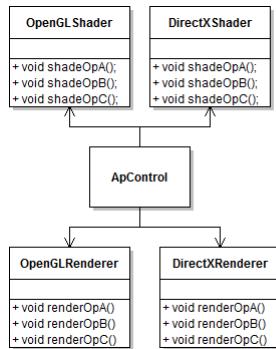
```
package dp.abstractFactory.solution1;
public class ApControl {
    private LibraryType library;
    private OpenGLRenderer or;
    private OpenGLShader os;
    private DirectXRenderer dr;
    private DirectXShader ds;
    public ApControl( ) { /*library nesnesini ilklendirerek kütüphaneyi belirle*/ }
    void doRendering ( ) {
        switch (library){
            case OpenGL:
                or.renderOpA(); or.renderOpB(); or.renderOpC(); break;
            case DirectX:
                dr.renderOpA(); dr.renderOpB(); dr.renderOpC(); break;
        }
    }
    void doShading ( ) {
        switch (library){
            case OpenGL:
                os.shadeOpA(); os.shadeOpB(); os.shadeOpC(); break;
            case DirectX:
                ds.shadeOpA(); ds.shadeOpB(); ds.shadeOpC(); break;
        }
    }
}
```

8

## ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

### ÇÖZÜM 1: Switch – Case Kullanımı

```
public enum LibraryType {  
    OpenGL, DirectX;  
}  
  
public class OpenGLRenderer {  
    public void renderOpA() {}  
    public void renderOpB() {}  
    public void renderOpC() {}  
}  
  
public class OpenGLShader {  
    public void shadeOpA() {}  
    public void shadeOpB() {}  
    public void shadeOpC() {}  
}  
  
public class DirectXRenderer {  
    public void renderOpA() {}  
    public void renderOpB() {}  
    public void renderOpC() {}  
}  
  
public class DirectXShader {  
    public void shadeOpA() {}  
    public void shadeOpB() {}  
    public void shadeOpC() {}  
}
```



9

## ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

### ÇÖZÜMÜN ZAYIF YÖNLERİ:

- Yeni bir kütüphane eklemek için (Ör. DirectX 10, 11 ve 12 için ayrı kütüphaneler) çok fazla yerde kod değişikliği gerek.

### TASARIM İPUCU:

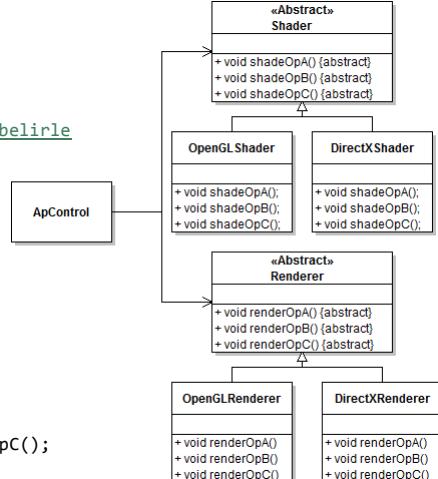
- Switch-case kullanımı çok biçimliliğe gereksinim duyulduğunun ve/veya sorumlulukların atanmasında yanlışlık olduğunun uyarı işaretini olabilir.

10

## ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

### ÇÖZÜM 2: Kalıtım Kullanımı

```
package dp.abstractFactory.solution2;
public class ApControl {
    private LibraryType library;
    private Renderer r;
    private Shader s;
    //library nesnesini ilklendirip kütüphane belirle
    public ApControl( ) {
        switch (library){
            case OpenGL:
                r = new OpenGLRenderer();
                s = new OpenGLShader();
                break;
            case DirectX:
                r = new DirectXRenderer();
                s = new DirectXShader();
                break;
        }
        void doRendering ( ) {
            r.renderOpA(); r.renderOpB(); r.renderOpC();
        }
        void doShading ( ) {
            s.shadeOpA(); s.shadeOpB(); s.shadeOpC();
        }
    }
}
```

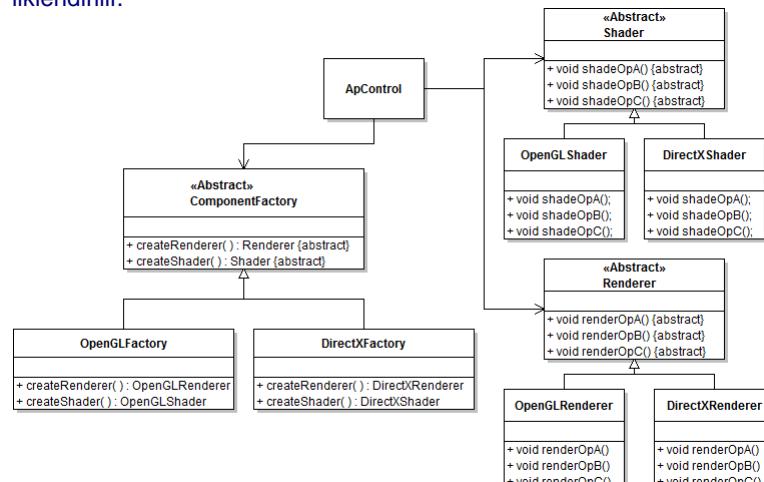


11

## ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

### ÇÖZÜM 3: Abstract Factory Kalibinin Kullanımı

- Ana program kurucusunda uygun bir ComponentFactory gerçeklemesi ilklendirilir.



12

## ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

### ÇÖZÜM 3: Abstract Factory Kalibinin Kullanımı

```
package dp.abstractFactory.solution3;
public class ApControl {
    private ComponentFactory library;
    private Renderer r;
    private Shader s;
    public ApControl( ) {
        /*library nesnesini ilkledirek
         *kütüphane türünü belirle
        library = new OpenGLFactory();
        library = new DirectXFactory();
        */
        r = library.createRenderer();
        s = library.createShader();
    }
    void doRendering ( ) {
        r.renderOpA(); r.renderOpB(); r.renderOpC();
    }
    void doShading ( ) {
        s.shadeOpA(); s.shadeOpB(); s.shadeOpC();
    }
}
```

13

## ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

### ÇÖZÜM 3: Abstract Factory Kalibinin Kullanımı

- Çözümlerdeki soyut sınıflar yerine arayüz kullanımı da mümkündür.
- Tüm çözümlerde library nesnesi ApControl kurucusuna parametre olarak verilirse daha doğru bir iş yapılmış olunur.
  - Neden? Bize ne kazandırır?

```
package dp.abstractFactory.solution3;
public class ApControl {
    private Renderer r;
    private Shader s;
    public ApControl(ComponentFactory aFactory) {
        r = aFactory.createRenderer();
        s = aFactory.createShader();
    }
    void doRendering ( ) {
        r.renderOpA(); r.renderOpB(); r.renderOpC();
    }
    void doShading ( ) {
        s.shadeOpA(); s.shadeOpB(); s.shadeOpC();
    }
}
```

14

## ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

### ÇIKARTILAN DERSLER = TASARIMA AİT GENEL KURALLAR

- Neyin değiştiğini bul ve onu sarmala
  - Böylece bu şeyi soyutlamış olursun
  - Böylece bu şeyi kara kutu yaklaşımı ile kullanabilirsin
  - Böylece bu konu ile ilgili değişiklikler sadece bu kısmı etkiler
  - Örnekte kullanılan yer: Hangi grafik kütüphanesinin kullanılacağı bir bilgisayardan başka bilgisayara geçilince değişeceğinden, bu değişim ComponentFactory sınıfında soyutlanmıştır.
- Kalıtım ilişkisi yerine parça/bütün ilişkisini tercih et
  - İlk çözümündeki (Çözüm 1) zayıf yönlerde dechinildi.
  - İlk çözümde kalıtım daha da kötü biçimde kurulabilirdi: ApControl sınıfından OpenGLApControl ve DirectXApControl sınıfları türetilebilirdi.
    - Sonuçta ortaya benzer kodun birden fazla yerde tekrarlanması nedeniyle kusurlu bir tasarım ortaya çıkacaktı.
- Gerçeklemelere göre değil, arayzlere göre tasarlama yap
  - Böylece kara kutu yaklaşımını kullanabilirsin
  - Örnekte ApControl sınıfı ComponentFactory sınıfından bir renderer ve bir shader oluşturmasını istiyor, bunu nasıl oluşturduğu umurunda değil.

15

## YARATIMSAL (CREATIONAL) KALIPLAR

### FACTORY METHOD:

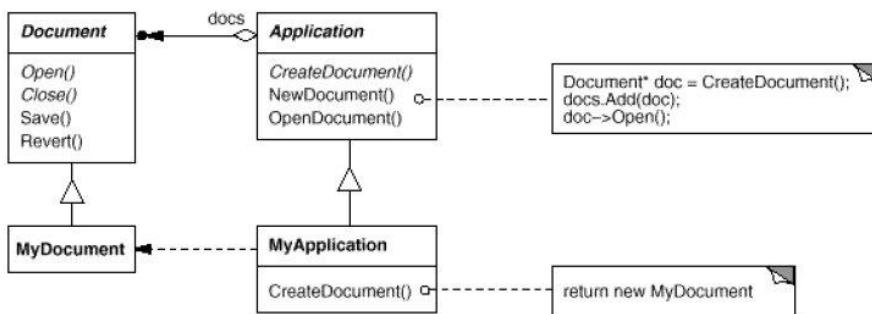
- Amaç:
  - Bir nesne oluşturmak için öyle bir ara yüz sunmak ki, oluşturulacak nesnenin sınıfına bu ara yüzü sağlayan sınıfın alt sınıfları karar verebilsin.
- Örnek:
  - Farklı belge türleri ile çalışabilecek uygulamalar geliştirmeyi sağlayan bir çerçeve program (framework) hazırlanıyor.
  - Temel sınıflar: Belge ve Uygulama.
  - Kullanıcı bu iki sınıfın kalıtım ile yeni sınıflar türeterek, kendi yazılımını hazırlıyor.
- Sorun:
  - Bir belge oluşturmak gereklince (aç, yeni, vb. komutlar) çerçeve program bu işlemi nasıl yapacak?
  - Uygulama, kullanıcının türeteceği yeni belge sınıfları hakkında önceden bilgi sahibi olamaz.

16

## YARATIMSAL (CREATIONAL) KALIPLAR

### FACTORY METHOD:

- Örnek Çözüm:



- Diğer seçenekler:

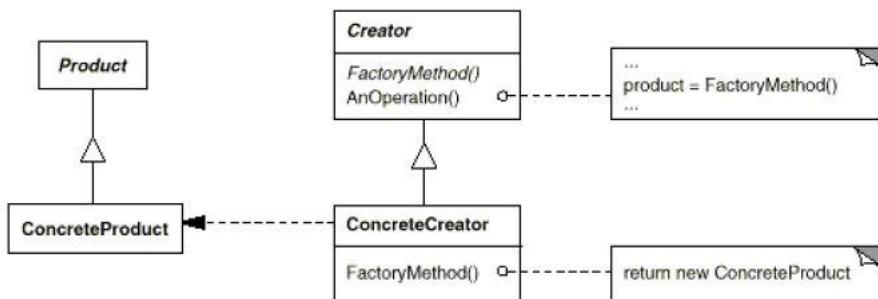
- Java'da Generic sınıflar:
  - class Document<DocType>
  - Zayıf yönü: Dile özel
- Başka?

17

## YARATIMSAL (CREATIONAL) KALIPLAR

### FACTORY METHOD:

- Kalıp yapısı:



18

## YARATIMSAL (CREATIONAL) KALIPLAR

### FACTORY METHOD:

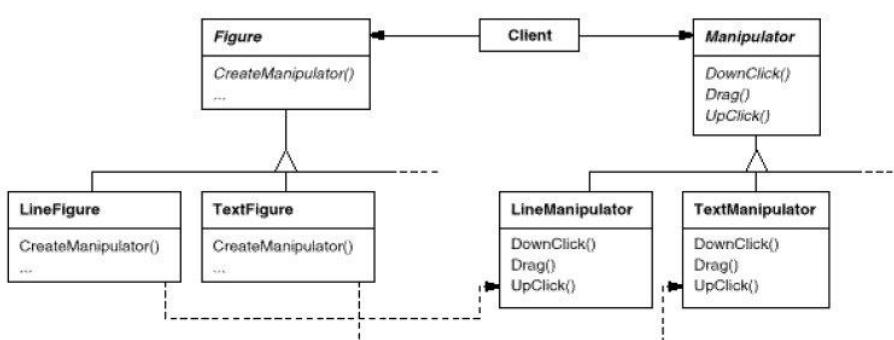
- Kalıbın uygulanabileceği anlar:
  - Yeni nesneler oluşturulması gereken bir sınıfın, oluşturacağı nesnenin türünü bilemediği yerlerde,
  - Bir sınıfın kendi üzerindeki bir sorumluluğu çeşitli yardımcı alt sınıflarına ileteceği ve hangi tür yardımcı alt sınıfın oluşturulması gerektiği bilgisinin ise yerelleştirilmesi gerektiği yerlerde.
  - Örnek: Bir çizim uygulamasında şekiller fare ile değiştirilecek.

19

## YARATIMSAL (CREATIONAL) KALIPLAR

### FACTORY METHOD:

- Örnek: Bir çizim uygulamasında şekiller fare ile değiştirilecek.



- Bazı şekil türleri özel bir değiştiriciye gerek duymuyorsa, **Figure** sınıfı varsayılan bir değiştirici örneği döndürebilir. Bu durumda her bir şekil türü için ayrı bir değiştirici sınıfı yazmaya gerek kalmaz.

20

## ÖRNEK KALIP GERÇEKLEMELERİ – FACTORY METHOD

### ÇÖZÜLECEK PROBLEM:

- E-ticaret sitesi farklı tür ürünler satışa sunabiliyor.
- Site çalışanı, bir müşterinin siparişindeki farklı tür ürünlerini işleme koyacak.
  - Müşteri siparişini işleme adımları belli bir sırada yürütülüyor.
  - Ancak adımlardaki iş mantığı ürün türüne göre çok farklılık gösteriyor.
- Site çalışanına sipariş vermede kullanabileceğim bir araç sağlayalım.
- Kullanım şu şekilde olabilir:

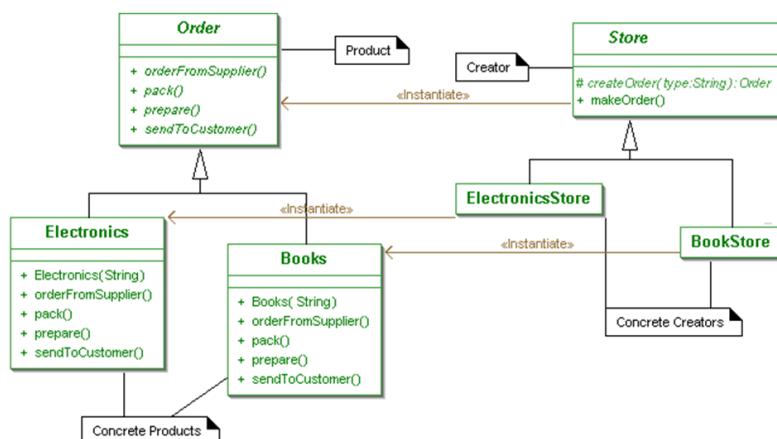
```
package factoryMethod.example;
public class MainApp {
    public static void main(String[] args) {
        Store store;
        //Elektronik ürün siparişi geldi
        store = new ElectronicsStore();
        store.createOrder("5524345678");
        //Kitap siparişi geldi
        store = new BookStore();
        store.createOrder("8693243565");
    }
}
```

- Esinlenme: Head First DP

21

## ÖRNEK KALIP GERÇEKLEMELERİ – FACTORY METHOD

### ÇÖZÜM:



22

## ÖRNEK KALIP GERÇEKLEMELERİ – FACTORY METHOD

### ÇÖZÜM:

```
package dp.factoryMethod.example;
public abstract class Order {
    public abstract void prepare( );
    public abstract void orderFromSupplier( );
    public abstract void pack( );
    public abstract void sendToCustomer( );
}
public class Electronics extends Order {
    private String barcode;
    public Electronics( String barcode ) {
        super( ); this.barcode = barcode;
    }
    public void orderFromSupplier() {
        //Elektronik malzeme için yapılacak işlemlerin tanımlanması
    }
    public void pack() {
        //Elektronik malzeme için yapılacak işlemlerin tanımlanması
    }
    public void prepare() {
        //Elektronik malzeme için yapılacak işlemlerin tanımlanması
    }
    public void sendToCustomer() {
        //Elektronik malzeme için yapılacak işlemlerin tanımlanması
    }
}
public class Books extends Order { /*Coded similarly*/ }
```

23

## ÖRNEK KALIP GERÇEKLEMELERİ – FACTORY METHOD

### ÇÖZÜM:

```
public abstract class Store {
    protected abstract Order createOrder( String type );
    public void makeOrder( String type ) {
        Order newOrder = createOrder( type );
        newOrder.prepare( );
        newOrder.orderFromSupplier( );
        newOrder.pack( );
        newOrder.sendToCustomer( );
    }
}
public class ElectronicsStore extends Store {
    protected Order createOrder( String type ) {
        Electronics newElectronicsOrder = new Electronics( type );
        //Elektronik malzeme siparişi için yapılacak işlemlerin tanımlanması
        return newElectronicsOrder;
    }
}
public class BookStore extends Store {
    protected Order createOrder(String type) {
        Books newBookOrder = new Books(type);
        //Kitap siparişi için yapılacak işlemlerin tanımlanması
        return newBookOrder;
    }
}
```

24

## YARATIMSAL (CREATIONAL) KALIPLAR

### BUILDER:

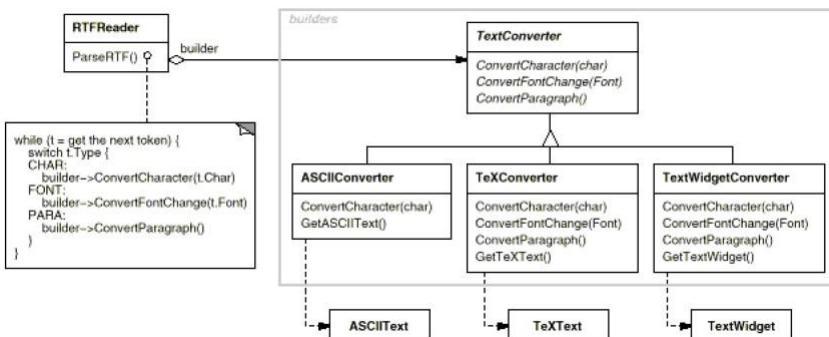
- Amaç:
  - Karmaşık bir nesnenin oluşturulması ile gösterimini birbirinden ayırmak. Böylece aynı oluşturma işlemi farklı gösterimler ortaya çıkarabilir.
- Örnek:
  - Bir RTF metin okuyucu programı, bu belgeyi çeşitli diğer metin türlerine, hatta metni düzenleyebilecek bir GUI bileşenine çevirebilsin.
- Sorun:
  - Bir çok farklı metin türleri var, hepsi hemen gerçeklenmeyecek.
  - Bu yüzden ileride yeni bir metin türü eklemek zor olmamalı.
- Çözüm:
  - Metni okumakla sorumlu sınıf, bir biçim çeviriçi nesnesine çeviri isteğini göndersin.
  - Çeviriçi nesne hem çevrim işleminden, hem de çevrilen metni belli bir biçimde simgelemekle yükümlü olsun.
  - Bu durumda yeni metin türleri için çeviriçi sınıfının alt sınıfları oluşturulabilir.

25

## YARATIMSAL (CREATIONAL) KALIPLAR

### BUILDER:

- Çözüm:



- Çözümün eleştirileri:

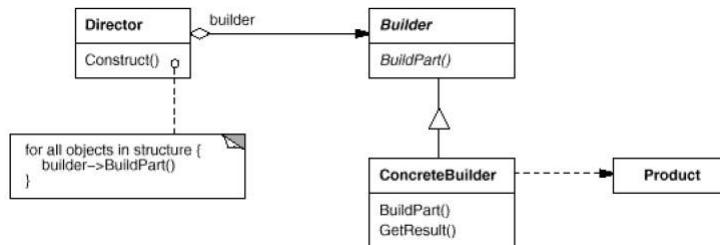
- Çokbılımlılığı tam kullanmadık, switch-case hala duruyor.
- Alt sınıflar üst sınıfın tüm metodlarını gerçeklemelidir. Bu yüzden alt sınıfta anlamı olmayan metodlar boş gövdeli olarak kalmak zorunda.

26

## YARATIMSAL (CREATIONAL) KALIPLAR

### BUILDER:

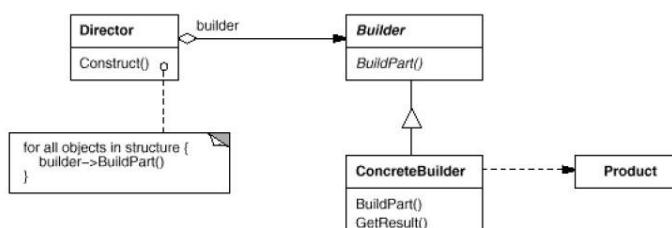
- Kalıp Yapısı:



- Kalıbın yukarıda verilen tasarımında switch-case gözükmüyor ama ConcreteBuilder.buildPart metodunda kendisini gösterecektir. Çünkü Director.construct metodunda "for all objects in structure" döngüsündeki objeler bir şekilde buildPart metoduna parametre olarak aktarılacak ve "parça şu ise şunu yap, bu ise bunu yap" gibi komutlar verilmek zorunda kalınacaktır.

27

### BUILDER:



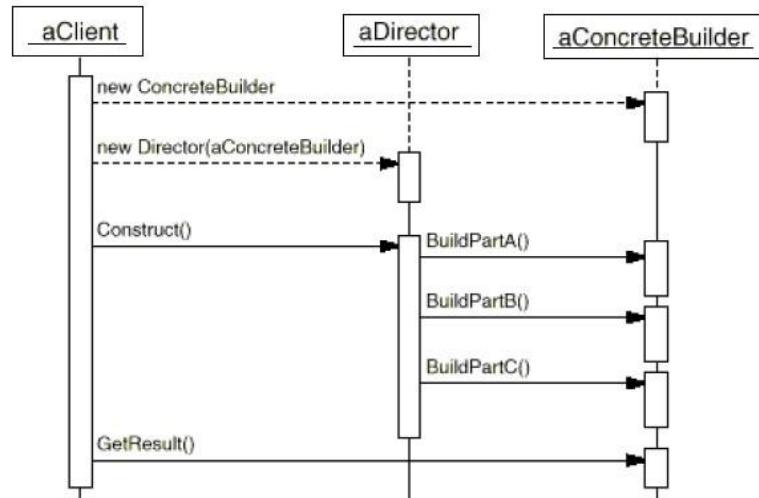
- Kalıp Bileşenleri:
  - Builder: Ürün nesnesinin parçalarını oluşturmak için bir arayüz sunar.
  - Product: Ürün.
  - Director: Ürünü, Builder arayüzüünü kullanarak oluşturur.
  - ConcreteBuilder: Builder gerçeklemesi.
    - Ürün nesnesinin parçalarını oluşturur ve birleştirir. İşin nasıl yapılacağı da burada kodlanır.
    - Oluşturacağı gösterimi tanımlamak ve oluşturma işlemini izlemekle yükümlüdür.
    - Oluşan ürünü Director'a geri vermek üzere bir 'getter' metodu sunar.

28

## YARATIMSAL (CREATIONAL) KALIPLAR

### BUILDER:

- Kalıp bileşenlerinin etkileşimleri:



## YARATIMSAL (CREATIONAL) KALIPLAR

### BUILDER:

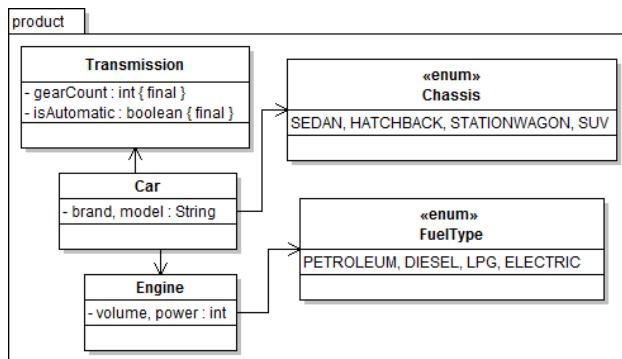
- Kalıbın uygulanabileceği anlar:
  - Karmaşık bir nesneyi oluşturma algoritmasının, nesneyi oluşturan parçalardan bağımsız olması gereği ve bu parçaların nasıl birleştirileceğinden bağımsız olması gereği anlarda.
  - Bir nesneyi oluşturma sürecinin bu nesnenin farklı gösterimlerinin olmasına izin verebilmesi gereği anlarda.
- Sonuçlar:
  - Ürünün iç yapısının en az etki ile değiştirilebilmesi mümkün olur (yeni bir ConcreteBuilder sınıfı yazarak).
  - Önceden gördüğümüz 'factory' kalıpları ile karşılaştırıldığında, ürün oluşturma süreci üzerinde daha fazla denetim sahibi olunmakta (Director örneği Builder örneğine hangi işlemi hangi sırada yapması gereği emrini verir).

30

## ÖRNEK KALIP GERÇEKLEMELERİ – BUILDER

### ÇÖZÜLECEK PROBLEM:

- Farklı marka ve model araçların satıldığı bir bayide markalara özel adlandırma ifadelerine rağmen ortak bir araç modellemesi yapılacak.
- Araçlar kasa, yakıt türü, motor hacmi ve gücü, vb. bileşenlerin bir araya getirilmesi ile karmaşık bir süreçle oluşmaktadır.



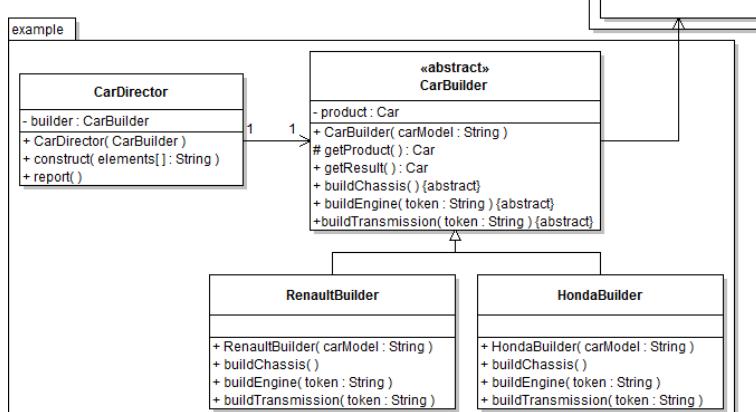
- Kaynak: YES

31

## ÖRNEK KALIP GERÇEKLEMELERİ – BUILDER

### ÖNERİLEN ÇÖZÜM:

- Builder kalibine uygun olarak yapılan tasarım:

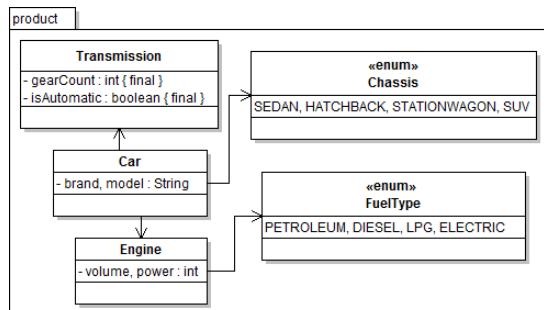


32

## ÖRNEK KALIP GERÇEKLEMELERİ – BUILDER

### ÖNERİLEN ÇÖZÜM:

- Product paketini açarsak:



33

## ÖRNEK KALIP GERÇEKLEMELERİ – BUILDER

### ÖNERİLEN ÇÖZÜM:

- Kaynak kodlar:

```
package dp.builder.example;
import dp.builder.product.*;
public abstract class CarBuilder {
    private Car product;
    public CarBuilder( String carModel ) {
        product = new Car(carModel);
    }
    public Car getResult() {
        if( product == null )
            return null;
        Car clone = new Car(product.getModel());
        clone.setBrand(product.getBrand());
        clone.setChassis(product.getChassis());
        clone.setEngine(product.getEngine());
        clone.setGear(product.getGear());
        return clone;
    }
    protected Car getProduct() { return product; }
    public abstract void buildChassis( );//Sasi'yi modele bağladım.
    public abstract void buildEngine(String token);
    public abstract void buildTransmission(String token);
}
```

34

## ÖRNEK KALIP GERÇEKLEMELERİ – BUILDER

### ÖNERİLEN ÇÖZÜM:

```
public class CarDirector {  
    private CarBuilder builder;  
    public CarDirector(CarBuilder builder) { this.builder = builder; }  
    public void construct(String elements[]) {  
        builder.buildChassis();  
        for( String element : elements ) {  
            int tab = element.indexOf("\t");  
            String item = element.substring(0,tab);  
            String info = element.substring(tab+1,element.length());  
            if( item.compareToIgnoreCase("Engine")== 0 )  
                builder.buildEngine(info);  
            else if( item.compareToIgnoreCase("Gear")== 0 )  
                builder.buildTransmission(info);  
        }  
    }  
    public void report( ) { System.out.println(builder.getResult()); }  
    public static void main(String[] args) {  
        CarDirector director = new CarDirector(  
            new HondaBuilder("Civic Sedan") );  
        String info[] = {"Engine\t125PS","Engine\t1.6L","Gear\t6AT"};  
        director.construct(info);  
        director.report();  
    }  
}
```

35

## ÖRNEK KALIP GERÇEKLEMELERİ – BUILDER

### ÖNERİLEN ÇÖZÜM:

- Somut bir CarBuilder alt sınıfı örneği:

```
package dp.builder.example;  
import dp.builder.product.*;  
public class HondaBuilder extends CarBuilder {  
    public HondaBuilder( String carModel ) {  
        super(carModel);  
        getProduct().setBrand("Honda");  
    }  
    public void buildChassis( ) {  
        if( getProduct().getModel().toUpperCase().contains("BACK") )  
            getProduct().setChassis(Chassis.HATCHBACK);  
        else if( getProduct().getModel().toUpperCase().contains("CR-V") )  
            getProduct().setChassis(Chassis.SUV);  
        else  
            getProduct().setChassis(Chassis.SEDAN);  
    }  
}
```

36

## ÖRNEK KALIP GERÇEKLEMELERİ – BUILDER

### ÖNERİLEN ÇÖZÜM:

- Somut bir CarBuilder alt sınıfı örneği (devam):

```
public void buildEngine(String token) {  
    //Min. values for a Honda engine  
    if( getProduct().getEngine() == null )  
        getProduct().setEngine(new Engine(FuelType.PETROLEUM,1150,60));  
    token = token.toUpperCase();  
    if( token.contains("ECO") )  
        getProduct().getEngine().setFuel(FuelType.LPG);  
    else if( token.contains("PS") ) {  
        int end = token.indexOf("PS");  
        token = token.substring(0, end);  
        getProduct().getEngine().setPower( Integer.parseInt(token) );  
    }  
    else if( token.contains("L") ) {  
        int end = token.indexOf("L");  
        token = token.substring(0, end);  
        getProduct().getEngine().setVolume(  
            (int)(Double.parseDouble(token) * 1000) );  
    }  
}
```

37

## ÖRNEK KALIP GERÇEKLEMELERİ – BUILDER

### ÖNERİLEN ÇÖZÜM:

- Somut bir CarBuilder alt sınıfı örneği (devam):

```
public void buildTransmission(String token) {  
    //By default, AutoTransmission has 4 gears and ManualT has 5.  
    int gearCount = Integer.parseInt(token.substring(0,1));  
    boolean isAutomatic = false;  
    if( token.toUpperCase().contains("AT") )  
        isAutomatic = true;  
    if( isAutomatic && gearCount == 0 )  
        gearCount = 4;  
    if( !isAutomatic && gearCount == 0 )  
        gearCount = 5;  
    getProduct().setGear( new Transmission(gearCount, isAutomatic) );  
}
```

38

## YARATIMSAL (CREATIONAL) KALIPLAR

### PROTOTYPE:

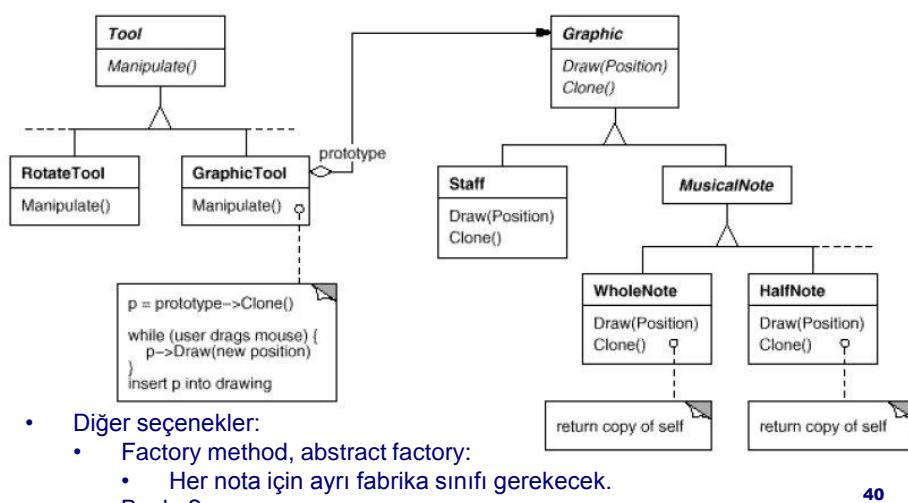
- Amaç:
  - Oluşturulacak nesnelerin türlerini belirlemek için bir prototip nesne kullanmak ve yeni nesneleri bu prototipi kopyalayarak oluşturmak.
- Örnek:
  - Grafik uygulamalarına yönelik mevcut bir çerçeve programını kullanarak, bir müzik besteleme programı yazılacak.
  - Çerçeve programının soyut bir Graphic sınıfı, çeşitli kontrol düğmelerinin yer alabileceği paletleri simgeleyen soyut bir Tool sınıfı, bundan kalıtımıla türetilmiş ve yeni grafik şekillerini çizim alanına eklemeye yarayan GraphicTool sınıfı var.
  - Notaları eklemek için GraphicTool sınıfı kullanılabilir.
- Sorun:
  - Çerçeve programının çeşitli nota sınıfları hakkında önceden bilgisi olamaz.
  - GraphicTool sınıfı bilmediği sınıflardan çizim nesnelerini nasıl oluşturur?

39

## YARATIMSAL (CREATIONAL) KALIPLAR

### PROTOTYPE:

- Örnek Çözüm:



- Diğer seçenekler:

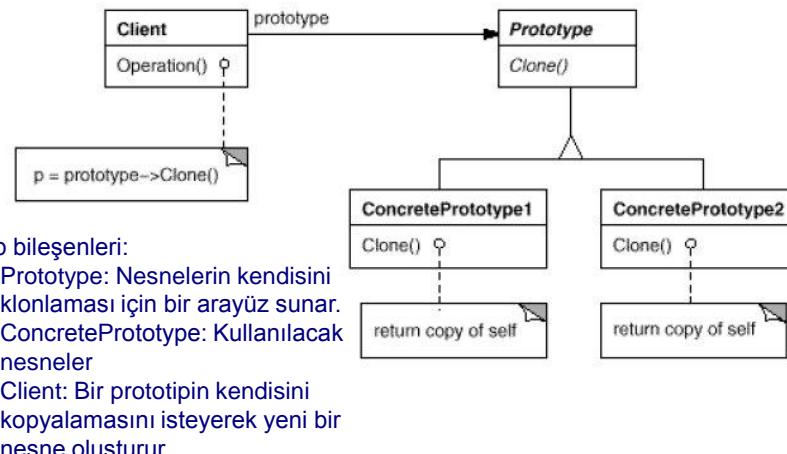
- Factory method, abstract factory:
    - Her nota için ayrı fabrika sınıfı gerekecek.
  - Başka?

40

## YARATIMSAL (CREATIONAL) KALIPLAR

### PROTOTYPE:

- Kalıp yapısı:



41

## YARATIMSAL (CREATIONAL) KALIPLAR

### PROTOTYPE:

- Kalıbin uygulanabileceği anlar:
  - Bir sistemin, kendisine ait ürünlerin nasıl oluşturulduğu, birleştirildiği ve sunulduğundan bağımsız olmasının istediği durumlar
  - ve şu durumlardan biri söz konusu ise (Factory Method kalıbından farklılaşmayı sağlıyor):
    - Oluşturulacak sınıflar çalışma anında belirlenecekse
    - Ürünlerin sınıf hiyerarşisi ile paralel bir factory hiyerarşisi kurmak istenmiyorsa
    - Ürün sınıfının örneklerinin durum uzayı sınırlı ise
      - Bu durumda sınıfın iyi tasarılanıp tasarılanmadığı ayrıca tartışılır!
- Bu kalıp C++ gibi sınıfların doğrudan işlenebilecek varlıklar olmadığı dillerde daha çok yarar sağlar.
- SmallTalk gibi geç ilişkilendirme (late binding) ve Java gibi yansıtma (reflection) yetenekleri olan dillerde bu kalıp yerine dilin yetenekleri kullanılabilir.

42

## YARATIMSAL (CREATIONAL) KALIPLAR

### PROTOTYPE:

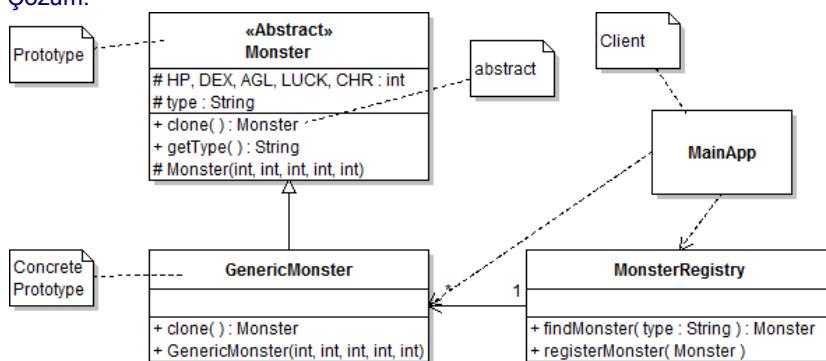
- Tartışmalar:
  - Siğ kopyalama veya derin kopyalama (shallow or deep copy)
    - Siğ kopyalama yetersiz kalabilir
    - Çevrimsellik derin kopyalamada sorun çıkartır.
  - Klonların üyelerine değer atanması için Prototype sınıfında ek metodlar tanımlamak gerekebilir.

43

## ÖRNEK KALIP GERÇEKLEMELERİ – PROTOTYPE

### ÇÖZÜLECEK PROBLEM:

- Bir bilgisayar oyunu için level editörü hazırlanıyor.
- Haritaya çok çeşitli türlerden canavarları kolayca ekleyebilmek istiyoruz.
- Ama bir canavar oluşturmak kolay iş değil, bir sürü durum bilgisi var:
  - HP, DEX, AGL, LUCK, CHR, vb.
- Çözüm:



- Kaynak: Head First DP

44

## ÖRNEK KALIP GERÇEKLEMELERİ – PROTOTYPE

### ÖNERİLEN ÇÖZÜM:

```
package dp.prototype.example;
public abstract class Monster {
    protected int HP, DEX, AGL, LUCK, CHR;
    protected String type;

    public abstract Monster clone( );

    protected Monster(int hp, int dex, int agl, int luck, int chr) {
        HP = hp; DEX = dex; AGL = agl; LUCK = luck; CHR = chr;
    }
    public String getType() { return type; }
    public int getHP() { return HP; }
    public void setHP(int hP) { HP = hP; }
    public int getDEX() { return DEX; }
    public void setDEX(int dEX) { DEX = dEX; }
    public int getAGL() { return AGL; }
    public void setAGL(int aGL) { AGL = aGL; }
    public int getLUCK() { return LUCK; }
    public void setLUCK(int lUCK) { LUCK = lUCK; }
    public int getCHR() { return CHR; }
    public void setCHR(int cHR) { CHR = cHR; }
    public void setType(String type) { this.type = type; }
}
```

45

## ÖRNEK KALIP GERÇEKLEMELERİ – PROTOTYPE

### ÖNERİLEN ÇÖZÜM:

```
package dp.prototype.example;
public class GenericMonster extends Monster {
    public GenericMonster(int hp, int dex, int agl, int luck, int chr) {
        super(hp, dex, agl, luck, chr);
        type = "Generic Monster";
    }
    public Monster clone( ) {
        return new GenericMonster(HP, DEX, AGL, LUCK, CHR);
    }
}
public class MonsterRegistry {
    private HashMap<String, Monster> monsters;
    public MonsterRegistry() { monsters = new HashMap<String,Monster>(); }

    public Monster findMonster( String type ) {
        return monsters.get( type );
    }
    public void registerMonster( Monster m ) {
        monsters.put( m.getType(), m );
    }
}
```

46

## ÖRNEK KALIP GERÇEKLEMELERİ – PROTOTYPE

### ÖNERİLEN ÇÖZÜM:

```
public class MainApp {  
    public static void main(String[] args) {  
        MonsterRegistry reg = new MonsterRegistry();  
        reg.registerMonster( new GenericMonster(10, 10, 10, 10, 10) );  
        Monster mySpecialMonster =  
            reg.findMonster("Generic Monster").clone();  
        mySpecialMonster.setHP(100);  
        mySpecialMonster.setType("Bolum sonu canavarı");  
        reg.registerMonster(mySpecialMonster);  
    }  
}
```

47

## YARATIMSAL (CREATIONAL) KALIPLAR

### SINGLETON:

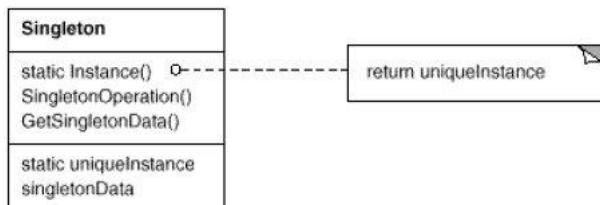
- Amaç:
  - Bir sınıfın sadece bir örneğinin bulunmasını sağlamak ve bu nesneye ortak bir erişim noktası vermek.
- Örnek:
  - Bazı nesnelerin türünün tek örneği olmasının gerekliliği olduğu alanlar bulunabilir.
  - Bir bilgisayarda birden fazla yazıcı tanımlı olabilmesine rağmen bir tek yazdırma biriktiricisi bulunur.
  - Bir işletim sisteminde tek bir dosya sistemi ve pencere yöneticisi bulunur.
- Çözüm 1:
  - Tek olacak nesne, static olarak tanımlanır.
  - Bu nesne farklı sınıflar tarafından kullanılacaksa, bunlardan hangisinin üyesi olacak?

48

## YARATIMSAL (CREATIONAL) KALIPLAR

### SINGLETON:

- Önerilen Çözüm:
  - Bir sınıf'a, kendisinin tek bir örneğinin olmasını sağlama sorumluluğunu atamak.
    - Sınıfın kendi türünden static bir üyesi olur,
    - Sınıfın kurucusu private tanımlanarak kurucuya erişim engellenir,
    - Sınıfın bu üye için bir factory metodu bulunur.
  - Kalıba adını veren bu sınıfın adı Singleton olarak belirlenmiştir.



49

## YARATIMSAL (CREATIONAL) KALIPLAR

### SINGLETON:

- Gerçekleme ayrıntıları:
  - Strategy ve State gerçeklemeleri, aynı zamanda iyi birer Singleton adayıdır (davranışsal kalıplar ileride incelenecək).
  - Bu kalıpta Singleton sınıflarını kalıtım yol ile özelleştirmek, ilk önerilen çözüme göre daha kolaydır.
  - Ancak Singleton nesnesi farklı türden nesneler tarafından kullanılmayacaksas, ilk öneriyi kullanmak yeterli olacaktır.
- Kalıbin kullanılabileceği anlar:
  - Bazı nesnelerin türünün tek örneği olmasının gerekliliği olduğu ve bu nesnelerin bir çok farklı sınıf örneklerince kullanılması istediği anlarda.

### ÖRNEK KALIP GERÇEKLEMELERİ – SINGLETON

- Basit bir kalıp olduğundan bu aşamada örnek yapılmayacaktır.

50

**NESNEYE DAYALI TASARIM VE MODELLEME  
KISIM 1: TASARIM KALIPLARI  
1.2. YAPISAL (STRUCTURAL) KALIPLAR**

**1**

**YAPISAL (STRUCTURAL) KALIPLAR**

**ADAPTER**

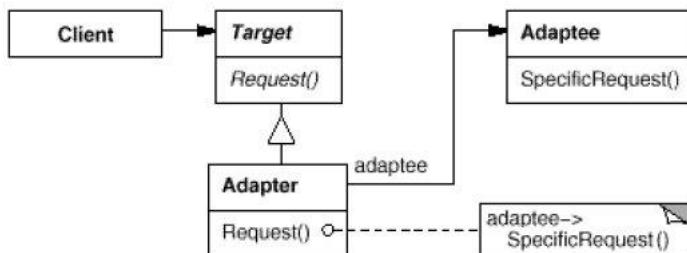
- Amaç:
  - İşlev açısından uyumlu ancak metod adları açısından uyumsuz bir istekçi sınıfı ile bir sunucu sınıfı, birbirleri ile tek yönlü konuşurmak.
- Örnek:
  - İngiliz standardındaki elektrik prizine Avrupa standardındaki cihazı takabilmek için bir adaptör kullanmak gereklidir.
  - Adapter kalıbı da aynı şekilde çalışır.

**2**

## YAPISAL (STRUCTURAL) KALIPLAR

### ADAPTER:

- Kalıp yapısı:



- Kalıp katılımcıları:

- Adaptee: Uyarlanması istenen sınıf (Sunucu)
- Client: Uyarlanacak sınıfa iş yaptırmak isteyen istekçi sınıf
- Target: İstekçinin konuşmak istediği arayüz
- Adapter: Uyarlama işlemini yapan, programcının yazacağı sınıf.
- Diğer sınıflar halihazırda mevcuttur.

3

## YAPISAL (STRUCTURAL) KALIPLAR

### ADAPTER:

- Kalıbın uygulanabileceği anlar:

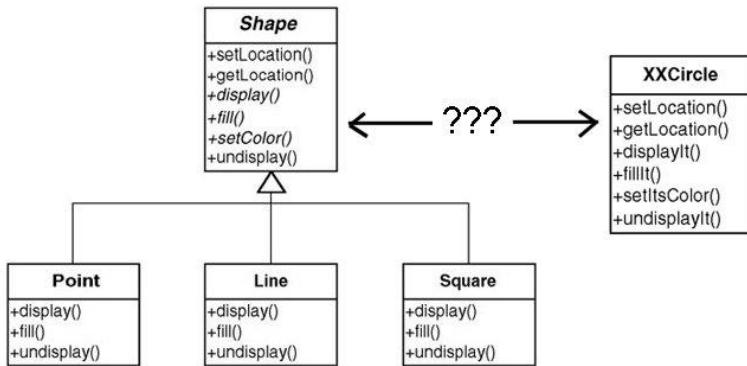
- Elimizdeki bir sınıfın bir istekçi tarafından kullanılabilmesini önleyen tek engelin, istekçinin yolladığı mesajlar ile eldeki sınıfın anladığı mesajların adlarının farklı olduğu anlarda.

4

### ÖRNEK KALIP GERÇEKLEMELERİ – ADAPTER

#### ÇÖZÜLECEK PROBLEM:

- Bir çizim programı üzerinde çalışılıyor.
- Programın bir kısmı halihazırda gerçeklenmiştir (aşağıda solda)



- Daire kodunu yazmadan önce farkettik ki, elimizde hazır bir daire sınıfı var.
- Ancak bu daire sınıfı mevcut programımıza uymuyor (yukarıda sağda). 5

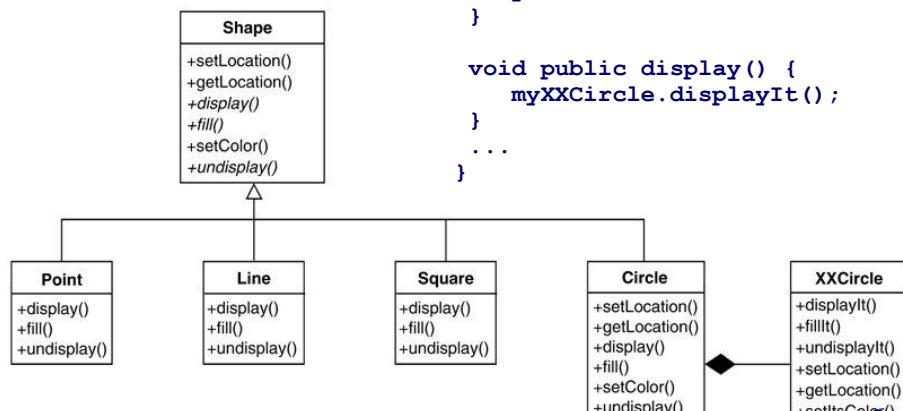
### ÖRNEK KALIP GERÇEKLEMELERİ – ADAPTER

#### ÇÖZÜM:

```

class Circle extends Shape {
    ...
    private XXCircle myXXCircle;
    ...
    public Circle () {
        myXXCircle= new XXCircle();
    }

    void public display() {
        myXXCircle.displayIt();
    }
    ...
}
    
```



## YAPISAL (STRUCTURAL) KALIPLAR

### BRIDGE:

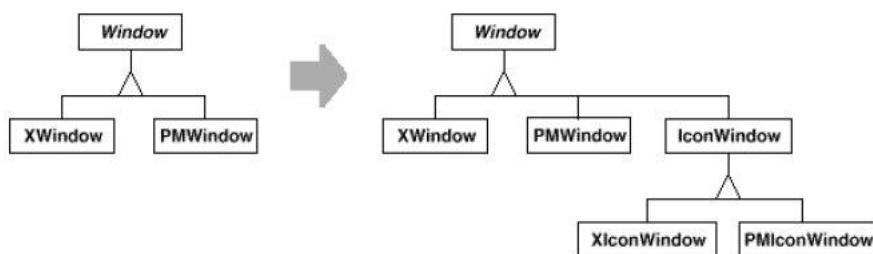
- Amaç:
  - Nesnenin arayüzü ile gerçeklemesini birbirinden ayırmak.
  - 'Handle' olarak da bilinir (Win32 programcılar!)
- Örnek:
  - Bir GUI çerçeve programının pencere soyutlaması.
  - Gnome, KDE gibi farklı masaüstü ortamlarının desteklenmesi isteniyor.
  - Pencere soyut sınıfından kalıtımıla yeni pencere sınıfları türetilebilir.
- Sorun:
  - Pencere soyutlamasının görev çubuğuında simge haline getirilmiş pencereler için de kullanılması isteniyor.
  - Pencere sınıfından kalıtımıla yeni bir IconWindow sınıfı üretmek yetmez, desteklenecek tüm masaüstü ortamları için IconWindow sınıfının alt sınıflarını da oluşturmak gereklidir:

7

## YAPISAL (STRUCTURAL) KALIPLAR

### BRIDGE:

- Sorun:



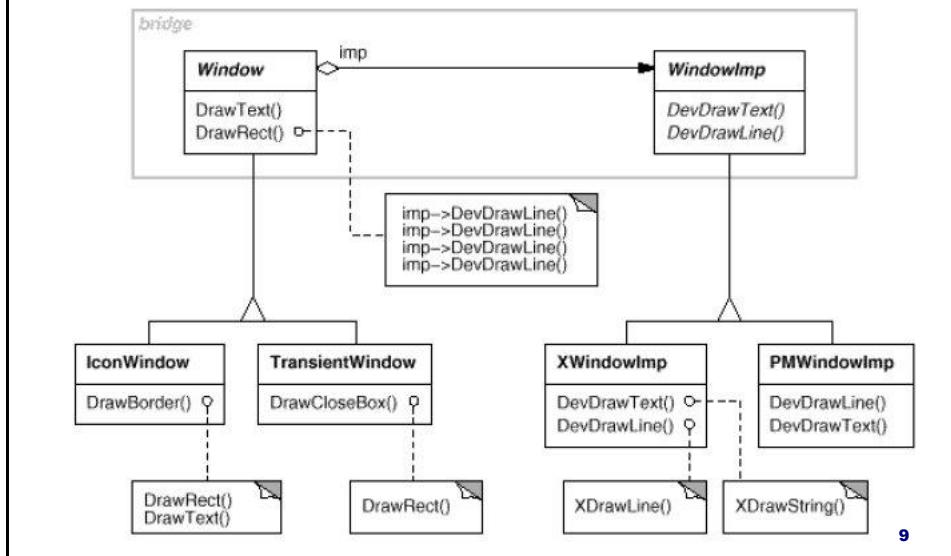
- Çözüm:

- Pencere soyutlaması ve gerçeklemeler ayrı sınıf hiyerarşilerine koyulur
- Farklı gerçeklemeler soyutlama nesnesinde saklanır.
- İşlemler soyutlamalar üzerinden çağrırlar.
- Soyutlama ile gerçekleme arasında köprü kurulmuş olur.

8

## YAPISAL (STRUCTURAL) KALIPLAR

### BRIDGE:

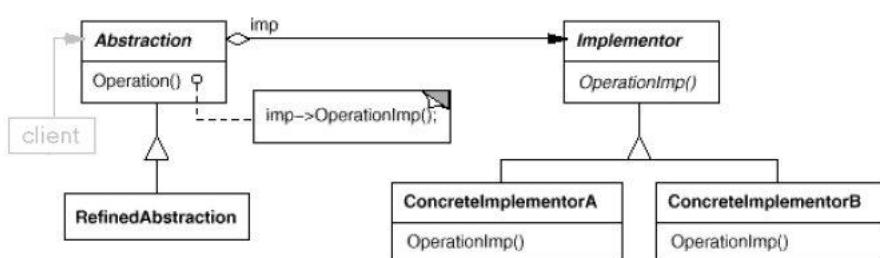


9

## YAPISAL (STRUCTURAL) KALIPLAR

### BRIDGE:

- Kalıp Yapısı:



- Kalıp bileşenleri:

- **Abstraction:** Uygulamanın temel işlemlerini oluşturan arayüzüni belirler.
- **RefinedAbstraction:** Arayüzü özelleştirmeye yarar.
- **Implementor:** Uygulamanın iç yapısına yönelik arayüzünü belirler.
- **ConcretelImplementor:** İç yapıyı gerçekleyen sınıf hiyerarşisi.

10

## YAPISAL (STRUCTURAL) KALIPLAR

### BRIDGE:

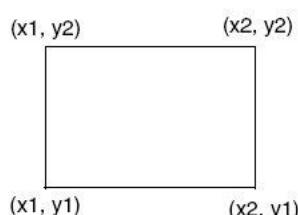
- Kalıbın uygulanabileceği anlar:
  - Bir soyutlama ile gerçeklemeleri arasındaki bağın kalıcı olmasının istenmediği anlarda.
  - Hem soyutlamaların hem de gerçeklemelerin ayrı ayrı özelleştirilmesine gerek duyulduğu anlarda.

11

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### ÇÖZÜLECEK PROBLEM:

- Bir çizim programı üzerinde çalışılıyor.
- İlk aşamada şekiller sadece dikdörtgenlerden oluşuyor, ancak iki farklı dikdörtgen ailesi var.
- Farklı iki aileden olan dikdörtgenler farklı iki yoldan çizilecek.
- Bir dikdörtgen nasıl çizileğini biliyor.



DP1

DP2

Used to draw a line

`draw_a_line( x1, y1, x2, y2 )`

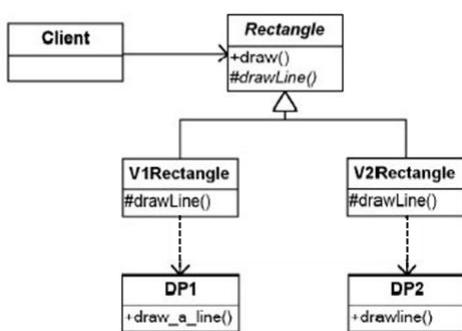
`drawline( x1, x2, y1, y2 )`

12

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### ÖNERİLEN ÇÖZÜM (1):

- İki tip farklı dikdörtgen varsa bunlar bir üst sınıfından kalıtımla türetilir.
- Bu ikisinin çizim metotları istenen iki farklı şekilde gerçekleşir.



- **protected** metodun amacı:
  - **drawLine** metodu nesnenin iç çalışması ile ilgilidir, o yüzden **public** yapılmamalıdır.
  - **private** metodlar kalıtım ile aktarılmaz.
  - **protected** metodlar hem farklı türden nesnelere görünmez, hem de alt sınıflara kalıtımla aktarılırlar.

13

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### ÖNERİLEN ÇÖZÜM (1):

```
package bridge.solution1;
public abstract class Rectangle {
    private double _x1, _x2, _y1, _y2;
    public void draw() {
        drawLine(_x1, _y1, _x2, _y1);
        drawLine(_x2, _y1, _x2, _y2);
        drawLine(_x2, _y2, _x1, _y2);
        drawLine(_x1, _y2, _x1, _y1);
    }
    abstract protected void drawLine ( double x1, double y1,
        double x2, double y2);
}
public class V1Rectangle extends Rectangle {
    protected void drawLine(double x1, double y1,
        double x2, double y2) {
        DP1.draw_a_line(x1,y1,x2,y2);
    }
}
```

14

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### ÖNERİLEN ÇÖZÜM (1):

```
package bridge.solution1;
public abstract class Rectangle {
    private double _x1, _x2, _y1, _y2;
    public void draw() {
        drawLine(_x1,_y1,_x2,_y1);
        drawLine(_x2,_y1,_x2,_y2);
        drawLine(_x2,_y2,_x1,_y2);
        drawLine(_x1,_y2,_x1,_y1);
    }
    abstract protected void drawLine ( double x1, double y1,
                                    double x2, double y2);
}
public class V2Rectangle extends Rectangle {
    protected void drawLine(double x1, double y1,
                           double x2, double y2) {
        DP2.drawLine(x1,x2,y1,y2);
    }
}
```

15

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### ÇÖZÜMÜN ZAYIF YÖNLERİ:

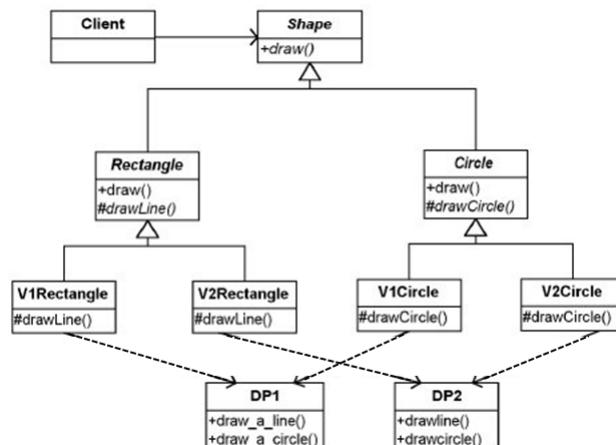
- Yeni bir gereksinim ortaya çıkıyor:
  - Çizim programındaki şekillere iki farklı çember ailesi de ekleniyor.
  - Farklı çember ailelerinin farklı çizim metodları var.
  - 1. dikdörtgen ailesi ile 1. çember ailesi ortaktır.
  - 2. dikdörtgen ailesi ile 2. çember ailesi ortaktır.
  - Bu durumda çember çizim metodları az önceki çözümün statik çizim metodlarında biriktirilebilir.
  - Çizim programının bir şeklin dikdörtgen veya çember olmasını dert etmemesi isteniyor.
  - Bunu karşılamak için dikdörtgen ve çember şekilleri, ortak bir Şekil sınıfı altında birleştirilebilir.
  - Tüm bunlar önceki tasarımlı bozmadan yapılrsa ortaya çıkan sınıf şeması şu şekilde olacaktır:

16

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### ÇÖZÜMÜN ZAYIF YÖNLERİ:

- Yeni bir gereksinimi karşılayan sınıf şeması:



17

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### ÇÖZÜMÜN ZAYIF YÖNLERİ:

- Değerlendirme:
  - Mevcut durumda 4 belirli şekil var (2 aile ve 2 şekil,  $2 \times 2 = 4$ ).
  - Yeni bir statik çizim metodu eklemek gerekirse sınıf sayısı 6 olacak.
  - Bunun üstüne yeni bir şekil (ör. üçgen) eklense sınıf sayısı 9 olacak (3 aile ve 3 şekil,  $3 \times 3 = 9$ ).
  - Her yeni eklenecek çizim metodu veya şekil türü ile sınıf sayısı fırlayacak!

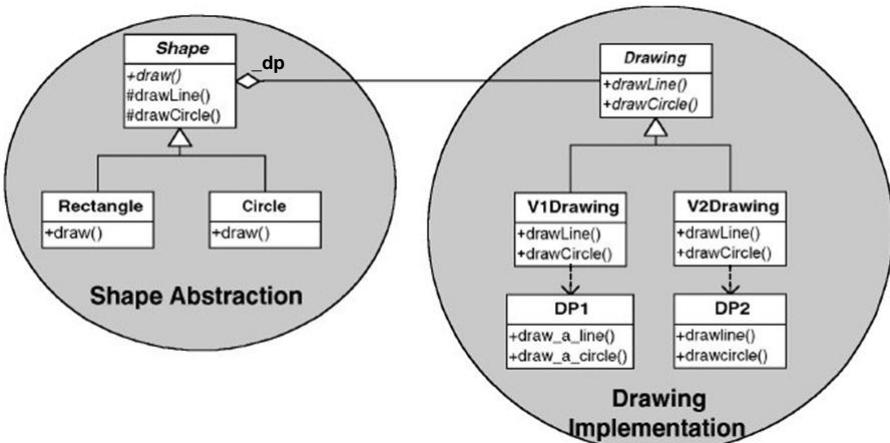
### YENİ ÇÖZÜM ÖNERİSİ:

- Sorunun kaynağı, farklı şekil türleri ile farklı çizim yollarının sıkı birlikteliğidir.
- Şekil türlerini simgeleyen soyutlamalar ile çizim yollarını simgeleyen gerçeklemeleri birbirinden ayırmak, sorunumuzu çözebilir.
- Bridge kalibinin amacını hatırlayın:
  - Nesnenin arayüzü ile gerçeklemesini birbirinden ayırmak.
  - Bir başka deyişle, soyutlamalar ile bunların gerçeklemelerini birbirinden ayırmak.

18

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### BRIDGE KALIBI İLE ÇÖZÜM:



19

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### BRIDGE KALIBI İLE ÇÖZÜM:

```
package bridge.solution2;
public abstract class Shape {
    public abstract void draw( );
    private Drawing _dp;
    public Shape (Drawing dp) { _dp= dp; }
    protected void drawLine(double x1, double y1,
        double x2, double y2) { _dp.drawLine(x1, y1, x2, y2); }
    protected void drawCircle(double x,double y,
        double r ) {_dp.drawCircle(x, y, r); }
}
public class Circle extends Shape {
    private double _x, _y, _r;
    public Circle (Drawing dp,
        double x,double y,double r) {
        super( dp) ;
        _x= x; _y= y; _r= r ;
    }
    public void draw () { drawCircle(_x,_y,_r); }
}
```

20

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### BRIDGE KALIBI İLE ÇÖZÜM:

```
package bridge.solution2;
public class Rectangle extends Shape {
    private double _x1, _y1, _x2, _y2;
    public Rectangle(Drawing dp,double x1,double y1,
                     double x2,double y2) {
        super(dp);
        _x1= x1; _x2= x2; _y1= y1; _y2= y2;
    }
    public void draw() {
        drawLine(_x1,_y1,_x2,_y1);
        drawLine(_x2,_y1,_x2,_y2);
        drawLine(_x2,_y2,_x1,_y2);
        drawLine(_x1,_y2,_x1,_y1);
    }
}
```

21

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### BRIDGE KALIBI İLE ÇÖZÜM:

```
package bridge.solution2;
public abstract class Drawing {
    public abstract void drawLine( double x1, double y1,
                                  double x2, double y2 );
    public abstract void drawCircle (double x,double y,
                                    double r);
}
public class V1Drawing extends Drawing {
    public void drawCircle(double x, double y, double r) {
        DP1.draw_a_circle(x,y,r);
    }
    public void drawLine(double x1, double y1,
                        double x2, double y2) {
        DP1.draw_a_line(x1,y1,x2,y2);
    }
}
```

22

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### BRIDGE KALIBI İLE ÇÖZÜM:

```
package bridge.solution2;
public class V2Drawing extends Drawing {
    public void drawCircle(double x, double y, double r) {
        DP2.drawcircle(x,y,r);
    }
    public void drawLine(double x1, double y1,
                        double x2, double y2) {
        DP2.drawline(x1,x2,y1,y2);
    }
}
```

23

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### BRIDGE KALIBI İLE ÇÖZÜM:

```
package bridge.solution2;
public class DP1 {
    public static void draw_a_line( double x1, double y1,
                                  double x2, double y2 ) {
        //çizimi yap
    }
    public static void draw_a_circle( double x,
                                    double y, double r ) {
        //çizimi yap
    }
}
```

24

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### BRIDGE KALIBI İLE ÇÖZÜM:

```
package bridge.solution2;
public class DP2 {
    public static void drawline( double x1, double x2,
        double y1, double y2 ) {
        //çizimi yap
    }
    public static void drawcircle ( double x, double y,
        double r ) {
        //çizimi yap
    }
}
```

25

## ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

### ÇIKARTILAN DERSLER = TASARIMA AİT GENEL KURALLAR

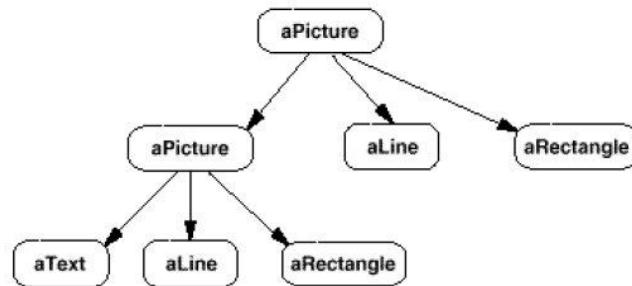
- Neyin değiştiğini bul ve onu sarmala.
  - Değişik türden şekiller ve değişik türden çizimler ortaya çıktı.
- Kalıtım ilişkisi yerine parça/bütün ilişkisini tercih et
  - Şekiller ve çizim türlerini aynı sınıflarda toplamak bağılaşımı arttırdı ve yeni bir şekil/cizim türü eklemek gereklili sınıf sayısını üstel olarak artırdı.

26

## YAPISAL (STRUCTURAL) KALIPLAR

### COMPOSITE:

- Amaç:
  - Hiyerarşi şeklindeki parça-bütün ilişkilerini desteklemeye yönelikir.
  - Bu kalıp sayesinde bireysel nesneler ve bir nesneler bütünü, istekçilere aynı şekilde gözükmür.
- Örnek: Bir grafik çizim programı hazırlanıyor.
  - Çizgi, dörtgen, metin gibi grafik bileşenleri kendilerini çizebilir.
  - Bu bileşenler bir resim adı altında gruplandırılabilir.
  - Bir resim ise alt resim parçalarından oluşabilir.

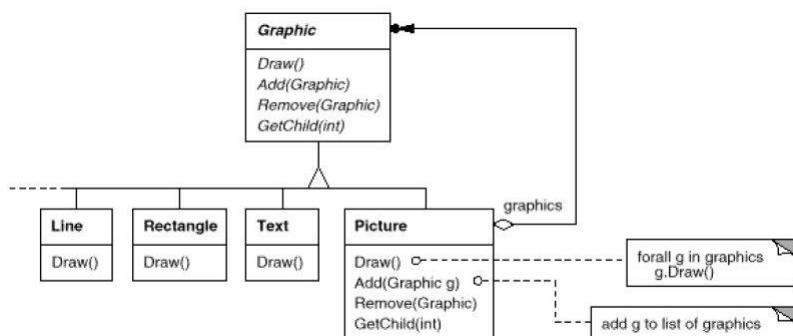


27

## YAPISAL (STRUCTURAL) KALIPLAR

### COMPOSITE:

- Örnek çözüm:



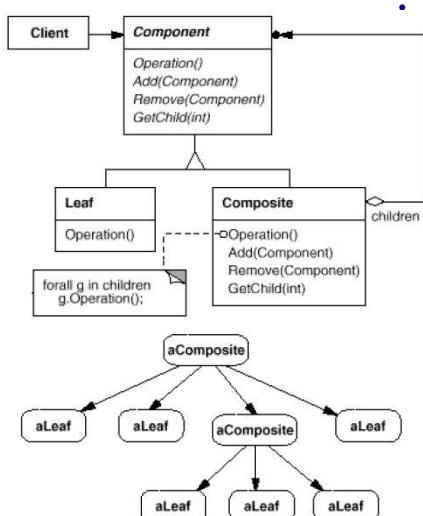
- Add ve Remove komutları ile çeşitli şekiller bir resim grubu altında toplanabilir.
- Bir grubun içeriği şekillerden birine GetChild metodu ile erişilebilir.

28

## YAPISAL (STRUCTURAL) KALIPLAR

### COMPOSITE:

- Kalıp yapısı:



- Kalıp katılımcıları:

- Component:
  - Nesneler topluluğunun katılımcıları için ortak kullanım arayüzü sunar.
  - Bir bütünüń parçaları için erişim ve düzenleme metotları tanımlar.
- Leaf:
  - Topluluktaki parçaları simgeler.
  - Hiyerarşik yapının en altındaki düğümleridir, çocukları olamaz.
- Composite:
  - Hiyerarşik yapının ara düğümleridir.
- Client: Parça ve toplulukları benzer şekilde kullanabilir.
  - Client bir Visitor olabilir.

29

## YAPISAL (STRUCTURAL) KALIPLAR

### COMPOSITE:

- Kalıbın kullanılabileceği anlar:
  - Nesneler arasındaki hiperarşik parça-bütün ilişkilerinin modellenmesi istenildiğinde.
  - Bir nesneler topluluğu ve topluluktaki bireysel nesnelere eş biçimli erişim gereklı olduğunda.
- Gerçekleme ayrıntıları:
  - Alt düğüm, üst düğümden haberdar kılınabilir.
  - Bir düğüm nasıl silinir? Kim siler? Nasıl bir veri yapısı kullanılır?
  - ...
- Kalıbın zayıf yönleri:
  - Kalıtım ilişkisinin özüne aykırı bir davranışta bulunur: Leaf bir alt sınıf olmasına rağmen, üst sınıfı olan Component'teki çocuk yönetim metodlarının bu alt sınıf için bir anlamı yoktur.
  - Halbuki alt sınıf üst sınıftaki tüm metodları kalıtımıla almak zorundadır.
  - Programcının karar vermesi gereken gerçekleme ayrıntısı sayısı, diğer kalıplara göre çok fazladır.

30

## ÖRNEK KALIP GERÇEKLEMELERİ – COMPOSITE

### ÇÖZÜLECEK PROBLEM:

- Makineler ve parçalarından oluşan üretim kataloğumuz var.
- Bu katalogda makinelerin ve/veya parçaların hangi fabrikada üretiltiği, maliyeti, vb. gibi bilgilerden çeşitli raporlar üretilmesi isteniyor.
- Bazı makineler kendi içinde makine parçaları içerebilmektedir.
- Bu durumda Composite kalıbını uygulamak yerinde olacaktır.

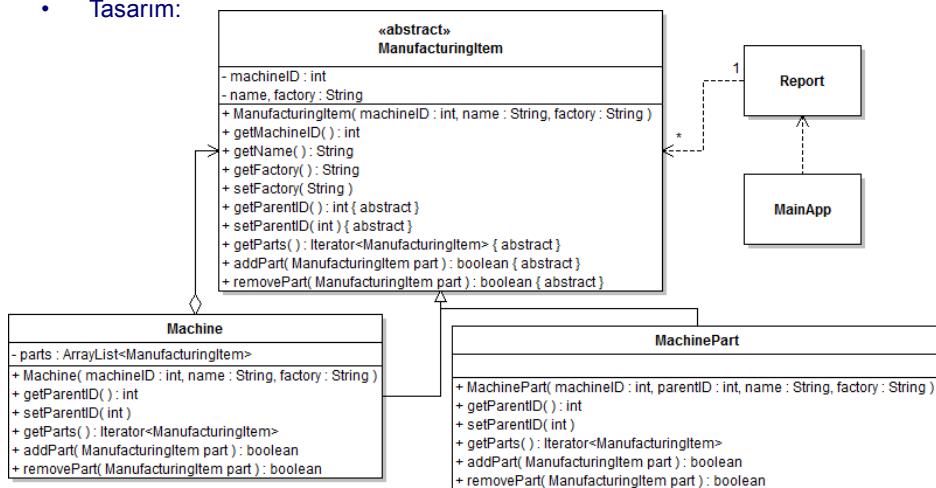
- Esinlenme: DP Java Workbook

31

## ÖRNEK KALIP GERÇEKLEMELERİ – COMPOSITE

### COMPOSITE KALIBI İLE ÇÖZÜM

- Tasarım:



32

## ÖRNEK KALIP GERÇEKLEMELERİ – COMPOSITE

### COMPOSITE KALIBI İLE ÇÖZÜM

- Kaynak kodlar:

```
public abstract class ManufacturingItem {  
    private int machineID;  
    private String name, factory;  
  
    public ManufacturingItem(int machineID, String name, String factory) {  
        this.machineID = machineID; this.name = name;  
        this.factory = factory;  
    }  
    public int getMachineID() { return machineID; }  
    public String getName() { return name; }  
    public String getFactory() { return factory; }  
    public void setFactory(String factory) { this.factory = factory; }  
    public abstract int getParentID();  
    public abstract void setParentID(int parentID);  
    public abstract Iterator<ManufacturingItem> getParts();  
    public abstract boolean addPart( ManufacturingItem part );  
    public abstract boolean removePart( ManufacturingItem part );  
}
```

33

## ÖRNEK KALIP GERÇEKLEMELERİ – COMPOSITE

### COMPOSITE KALIBI İLE ÇÖZÜM

```
public class Machine extends ManufacturingItem {  
    private ArrayList<ManufacturingItem> parts;  
    public Machine( int machineID, String name, String factory ) {  
        super(machineID, name, factory);  
        parts = new ArrayList<ManufacturingItem>();  
    }  
    public int getParentID() { return 0; }  
    public void setParentID(int parentID) { }  
    public boolean addPart( ManufacturingItem part ) {  
        part.setParentID(getMachineID());  
        for( ManufacturingItem aPart : parts )  
            if( aPart == part )  
                return false;  
        return parts.add(part);  
    }  
    public boolean removePart( ManufacturingItem part ) {  
        return parts.remove(part);  
    }  
    public Iterator<ManufacturingItem> getParts( ) {  
        return parts.iterator();  
    }  
}
```

34

## ÖRNEK KALIP GERÇEKLEMELERİ – COMPOSITE

### COMPOSITE KALIBI İLE ÇÖZÜM

```
public class MachinePart extends ManufacturingItem {  
    private int parentID;  
    public MachinePart( int machineID, int parentID,  
                        String name, String factory) {  
        super(machineID, name, factory);  
        this.parentID = parentID;  
    }  
    public int getParentID() { return parentID; }  
    public void setParentID(int parentID) { this.parentID = parentID; }  
    public Iterator<ManufacturingItem> getParts() { return null; }  
    public boolean addPart(ManufacturingItem part) { return false; }  
    public boolean removePart(ManufacturingItem part) { return false; }  
}
```

35

```
public class Report {  
    public static int findItems(ManufacturingItem items[],String factory) {  
        int nr = 0;  
        System.out.println("Items manufactured in " + factory + ":" );  
        System.out.println("Nr.\tID\tName");  
        System.out.println("-----");  
        for( ManufacturingItem item : items ) {  
            if( item==null ) continue;  
            if( item.getFactory().compareToIgnoreCase(factory) == 0 ) {  
                nr++; System.out.println( nr + ".\t" + item.getMachineID()  
                                + "\t" + item.getName() );  
            }  
            Iterator<ManufacturingItem> subItems = item.getParts();  
            if( subItems==null ) continue;  
            while( subItems.hasNext() ) {  
                ManufacturingItem part = subItems.next();  
                if( part.getFactory().compareToIgnoreCase(factory) == 0 ) {  
                    nr++; System.out.println( nr + ".\t" +  
                                    part.getMachineID() + "\t" + part.getName() );  
                }  
            }  
        }  
        if( nr == 0 ) System.out.println("No such item found. ");  
        return nr;  
    }  
}
```

36

## ÖRNEK KALIP GERÇEKLEMELERİ – COMPOSITE

### COMPOSITE KALIBI İLE ÇÖZÜM

```
public class MainApp {  
    public static void main(String[] args) {  
        ManufacturingItem[ ] items = new ManufacturingItem[4];  
        items[0] = new MachinePart(99, 127, "Krank şafı KŞ-7", "Bursa");  
        Machine jenerator = new Machine(135, "Jeneratör JMX-99", "İstanbul");  
        jenerator.addPart(new MachinePart(259,135,"Flans FTR-5","Bursa"));  
        jenerator.addPart(new MachinePart(378,135,"Kasnak KS-9","İstanbul"));  
        jenerator.addPart(new MachinePart(196,135,"Rulman RL-3","Ankara"));  
        items[1] = jenerator;  
        items[2] = new MachinePart(63, 76, "Distribütör DST-12", "Bursa");  
        items[3] = new MachinePart(82, 19, "Pnömatik Vana VPN-7", "Ankara");  
        int sayı = Report.findItems(items, "Bursa");  
        System.out.println("\t" + sayı + " parça bulundu.");  
    }  
}
```

37

## YAPISAL (STRUCTURAL) KALIPLAR

### DECORATOR:

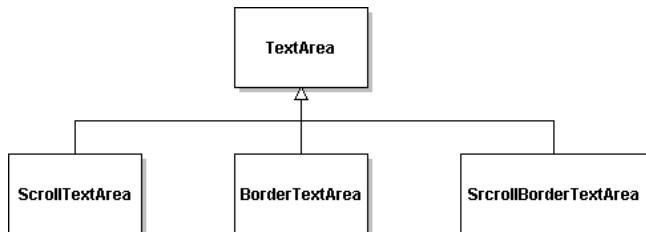
- Amaç:
  - Nesnelere çalışma anında (dinamik olarak) ek sorumluluklar atamak.
    - Bu iş kalıtmıla ancak kodlama anında yapılabilir.
  - Böylece bireysel nesneler özelleştirilebilir.
    - Kalıtımında özelleşme sınıf düzeyinde olduğundan, aynı tipteki bütün nesneler birden özelleşecektir.
- Örnek:
  - Bir GUI çerçeve uygulaması yazılıyor.
  - Metin alanlarını bazen çerçeve ile çevrelemek, bazen kaydırma çubuğu ile donatmak isteniyor.

38

## YAPISAL (STRUCTURAL) KALIPLAR

### DECORATOR:

- Çözüm 1:



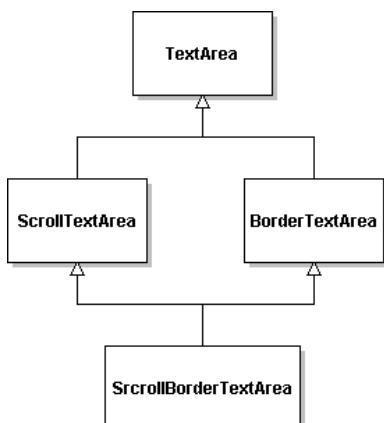
- Çözümün zayıf yönleri:
  - Scroll ve Border yetenekleri iki ayrı yerde tekrar gerçekleşmek zorunda.
  - **TextArea**'ya ek olarak bir de Panel sınıfına Scroll ve Border yeteneği kazandırmak için ek üç sınıf daha gerekecek.

39

## YAPISAL (STRUCTURAL) KALIPLAR

### DECORATOR:

- Çözüm 2:



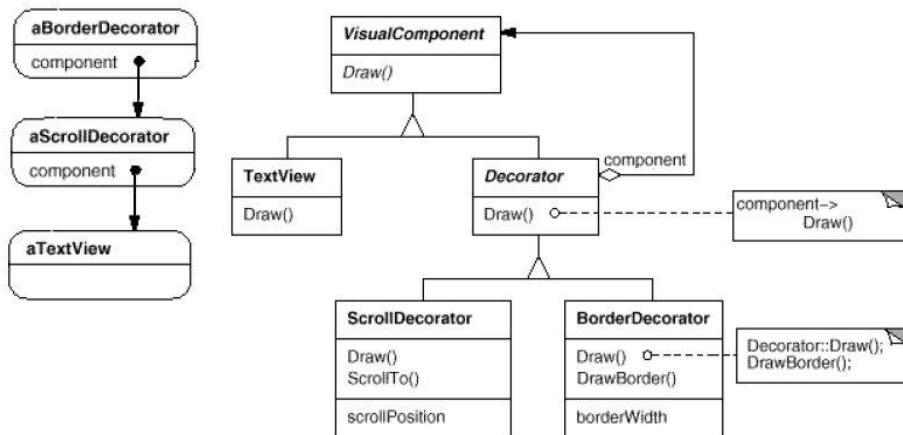
- Çözümün zayıf yönleri:
  - Çoklu kalıtım her dilde desteklenmez.
  - Çoklu kalıtımın bu çözümdeki şekli 'diamond problem' sorununun bir örneğidir.
  - **TextArea**'ya ek olarak bir de Panel sınıfına Scroll ve Border yeteneği kazandırmak için ek üç sınıf daha gerekecek.
    - Üstelik bu durumda bu örneğin Scroll ve Border yeteneklerinin tek yerde gerçekleşmesi avantajı ortadan kalkacak.

40

## YAPISAL (STRUCTURAL) KALIPLAR

### DECORATOR:

- Önerilen çözüm: Sınıfları değil de, nesneleri birbirleri ile ilişkilendirelim

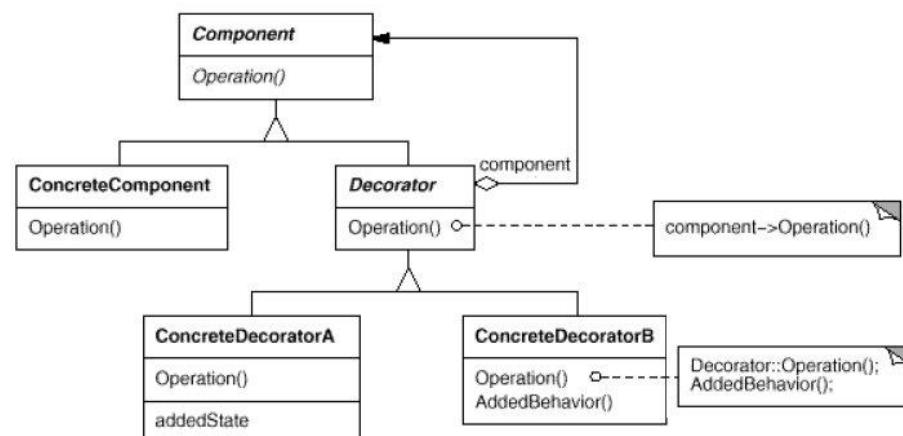


41

## YAPISAL (STRUCTURAL) KALIPLAR

### DECORATOR:

- Kalıp yapısı:



42

## YAPISAL (STRUCTURAL) KALIPLAR

### DECORATOR:

- Kalıp katılımcıları:
  - Component: Bileşen nesnesi
    - Kendisine ek sorumluluklar kazandırılabilenek nesnelerin arayüzüne tanımlar.
    - Bu sorumluluk, sadece bu arayüzde tanımlanan eylemlere kazandırılabilir.
    - Ek sorumlulukların tanımlanacağı donatıcılar da bu arayüzü gerçekleştirebilir.
  - ConcreteComponent: Bileşen gerçeklemesi.
  - Decorator: Donatıcı
    - Bir bileşen nesnesine işaretçi tutar.
  - ConcreteDecorator: Donatıcı gerçeklemesi.

43

## YAPISAL (STRUCTURAL) KALIPLAR

### DECORATOR:

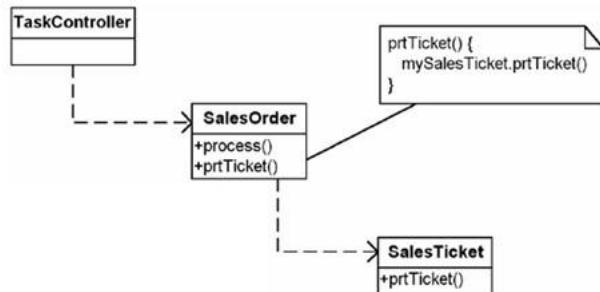
- Kalının kullanılabileceği anları:
  - Tek tek nesnelere dinamik olarak (çalışma anında) yeni sorumluluklar eklenmek istediginde
  - Tek tek nesnelere şeffaf olarak (diğerlerini etkilemeden) yeni sorumluluklar eklenmek istediginde
  - Bazı sorumlulukların iptal edilebileceği veya askıya alınabileceği durumlarda
  - Kalıtımın uygun olmadığı anlarda.
    - Ör: Sınıf kodu mevcut değil, yeni eklemeler üstel artan sınıf sayısına neden olacak, vb.
  - Java'nın Stream tabanlı I/O kütüphanesi bu kalımı kullanır.
- Kalının zayıf yönü:
  - Yeni sorumluluk, ancak şu şekillerde eklenebilir:
    - Eski sorumluluğun tamamen iptal edilmesi.
      - Dikkat: Nesne zinciri kopabilir!
    - Eski sorumluluktan önce ve/veya sonra yeni komutlar eklenmesi.

44

## ÖRNEK KALIP GERÇEKLEMELERİ – DECORATOR

### ÇÖZÜLECEK PROBLEM:

- Satış emirlerinin makbuzlarını basacak şekilde TaskController adlı programımıza yeni bir işlevsellik eklemek istiyoruz.
- İlk çözüm:



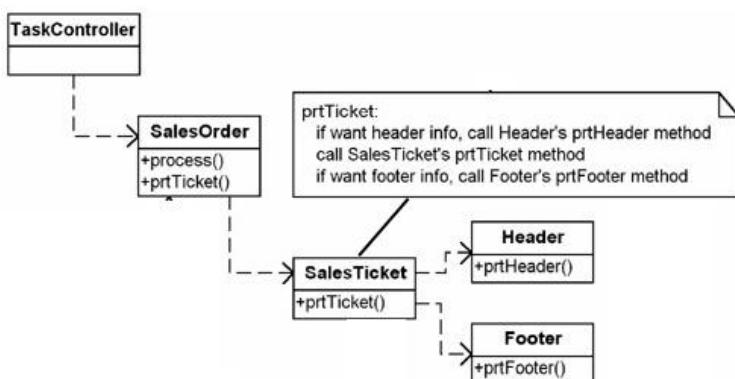
- Yeni gereksinim: Makbuzlara üstbilgi ve altbilgi (header/footer) eklemek.

45

## ÖRNEK KALIP GERÇEKLEMELERİ – DECORATOR

### ÇÖZÜM 1:

- Kullanım ilişkisi ile:



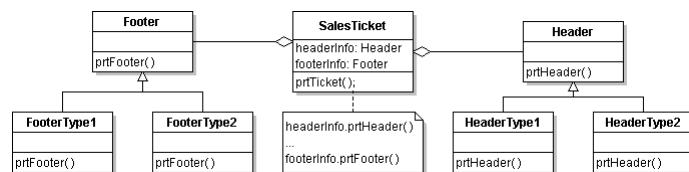
- Makbuz basılacağı zaman SalesTicket sınıfında ayrı if sınımları ile üstbilgi ve/veya altbilgi basılıp basılmayacağına karar verilir.
- Farklı üstbilgi ve altbilgi türleri söz konusu ise ne olacak?

46

## ÖRNEK KALIP GERÇEKLEMELERİ – DECORATOR

### ÇÖZÜM 2:

- Farklı üstbilgi ve altbilgi türleri için ayrı kalıtım hiyerarşileri kullanılabilir:
  - (Henüz incelememi ama bir Strategy kalıbı gerçeklemedesidir)



- Peki ya aynı makbuzda birden fazla farklı üstbilgi ve/veya altbilgi basılması isteniyorsa?
- Örneğin:

HEADER1
HEADER2
Receipt
FOOTER2

HEADER1
Receipt
FOOTER1
FOOTER2

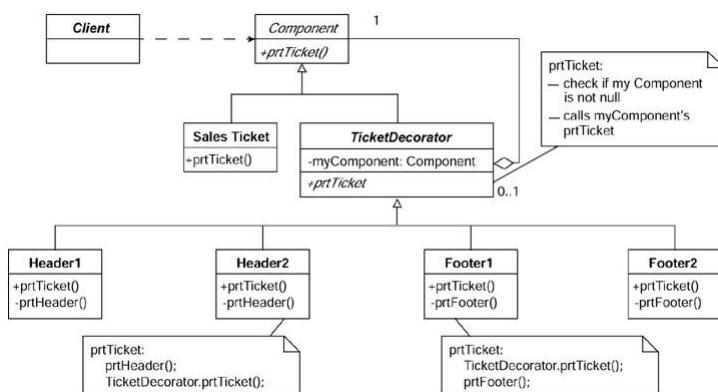
...

- Bu durumda ancak Decorator kalıbı gereksinimi karşılar.

47

## ÖRNEK KALIP GERÇEKLEMELERİ – DECORATOR

### ÖNERİLEN ÇÖZÜM:



48

## ÖRNEK KALIP GERÇEKLEMELERİ – DECORATOR

### ÖNERİLEN ÇÖZÜM:

```
package decorator.example;
public abstract class Component {
    abstract public void prtTicket();
}
public class SalesTicket extends Component {
    public void prtTicket() {
        // place sales ticket printing code here
        System.out.println(getClass().getName());
    }
}
public abstract class TicketDecorator extends Component {
    private Component myTrailer;
    public TicketDecorator (Component myComponent) {
        myTrailer= myComponent;
    }
    public void callTrailer () {
        if (myTrailer != null) myTrailer.prtTicket();
    }
}
```

49

## ÖRNEK KALIP GERÇEKLEMELERİ – DECORATOR

### ÖNERİLEN ÇÖZÜM:

```
package decorator.example;
public class Header1 extends TicketDecorator {
    public Header1 (Component myComponent) {
        super(myComponent);
    }
    public void prtTicket () {
        // place printing header 1 code here
        System.out.println(getClass().getName());
        super.callTrailer();
    }
}
public class Header2 extends TicketDecorator {
    public Header2 (Component myComponent) {
        super(myComponent);
    }
    public void prtTicket () {
        // place printing header 2 code here
        System.out.println(getClass().getName());
        super.callTrailer();
    }
}
```

50

## ÖRNEK KALIP GERÇEKLEMELERİ – DECORATOR

### ÖNERİLEN ÇÖZÜM:

```
package decorator.example;
public class Footer1 extends TicketDecorator {
    public Footer1 (Component myComponent) {
        super(myComponent);
    }
    public void prtTicket() {
        super.callTrailer();
        // place printing footer 1 code here
        System.out.println(getClass().getName());
    }
}
public class Footer2 extends TicketDecorator {
    public Footer2 (Component myComponent) {
        super(myComponent);
    }
    public void prtTicket() {
        super.callTrailer();
        // place printing footer 2 code here
        System.out.println(getClass().getName());
    }
}
```

51

## ÖRNEK KALIP GERÇEKLEMELERİ – DECORATOR

### ÖNERİLEN ÇÖZÜM:

- Peki Client sınıfı nasıl kodlanacak?
  - Ne tür üst/alt bilgi bileşimi seçileceğine (ilgili dekorasyona) Client karar verebilir.
  - Tasarımı bu aşamada FactoryMethod kalıbı ile geliştirebiliriz.

```
package decorator.example;
public class Client {
    public static void main(String[] args) {
        Factory myFactory = new Factory();
        Component myComponent = myFactory.getComponent();
        myComponent.prtTicket();
    }
}
```

52

## ÖRNEK KALIP GERÇEKLEMELERİ – DECORATOR

### ÖNERİLEN ÇÖZÜM:

- Factory sınıfının gerçeklemesi:

```
package decorator.example;
public class Factory {
    public Component getComponent () {
        Component myComponent;
        myComponent= new SalesTicket();
        myComponent= new Footer1(myComponent);
        myComponent= new Header1(myComponent);
        return myComponent;
        //VEYA: Aşağıdaki kod aynı etkiyi yapar
        //return(new Header1(new Footer1(new SalesTicket())));
    }
}
```

- Çalışma sonucu:

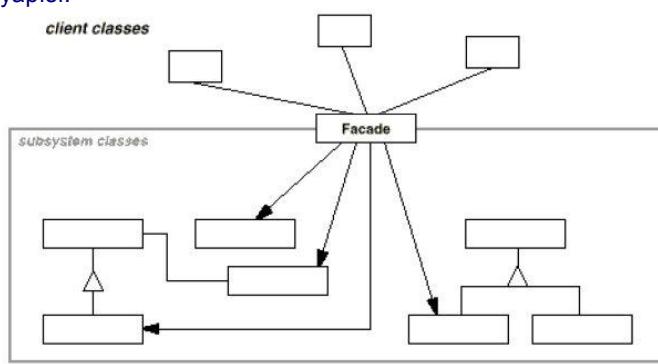
```
HEADER1  
Receipt  
FOOTER1
```

53

## YAPISAL (STRUCTURAL) KALIPLAR

### FACADE:

- Amaç:
  - Çeşitli alt sistemlerde bulunan farklı arayüzleri tek ve daha üst düzey bir arayüzde toplamak.
  - Böylece istekçiler alt sistemler ile ilgili alt düzey ayrıntılardan soyutlanabilir.
- Kalıp yapısı:



54

## YAPISAL (STRUCTURAL) KALIPLAR

### FACADE:

- Kalıbın uygulanabileceği anlar:
  - Karmaşık bir alt sisteme basit bir arayüz sağlamak istenildiğinde,
  - Alt sistemlerin çeşitli katmanlara ayrılması istenildiğinde,
  - Bazı istekçilerin sistemin belli bir kısmıyla bağlaşımının azaltılması istenildiğinde.

### ÖRNEK KALIP GERÇEKLEMELERİ – FACADE

- Basit bir kalıp olduğundan örnek yapılmayacaktır.

55

## YAPISAL (STRUCTURAL) KALIPLAR

### FLYWEIGHT:

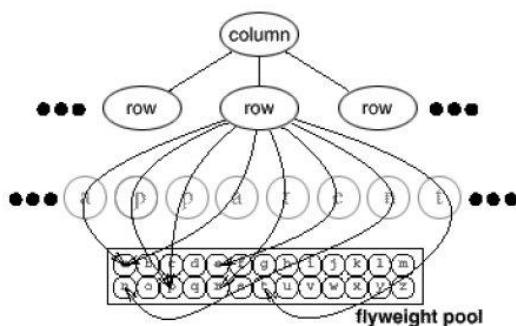
- Amaç:
  - Paylaşım yolu ile, çok sayıda basit nesnenin kullanımının desteklenmesi.
- Örnek:
  - Bir metin düzenleme programı yazılıyor.
  - Programın font ve diğer biçimlendirme bilgilerine göre, metin bilgisini çok net göstermesi isteniyor (excellent rendering).
  - Harfleri grafik nesneleri olarak gösterip, en iyi sonucu almak istiyoruz.
  - Metin sütunlar ve satırlar içerisinde bulunan harfler olarak ele alınabilir.
- Sorun:
  - Ancak her harfi ayrı bir nesne örneği olarak ele alırsak, küçük bir belgede bile binlerce nesne olacaktır.
  - Bu nesnelerin bellekte ve ikincil saklama ortamlarında kaplayacağı yer çok aşırı olacaktır.
  - Bu dersin üç sayfalık öneri formunda bile, boşluklarla birlikte 5000 kadar harf bulunmakta.

56

## YAPISAL (STRUCTURAL) KALIPLAR

### FLYWEIGHT:

- Çözüm:
  - Unicode alfabetesindeki her harfi bir harf nesnesi olarak ele alalım,
  - Metnin sütun ve satırlarında yeni bir harf nesnesi oluşturmak yerine,
  - Her bir harfi simgeleyen tek harf nesnesine, o harfin yer aldığı konumlardan referans verelim.
- Diğer bir deyişle, harf nesnelerinin ortak bir havuzdan paylaşımı kullanımını sağlayalım.
- Bu tip paylaşılan nesnelere flyweight adı verilir.

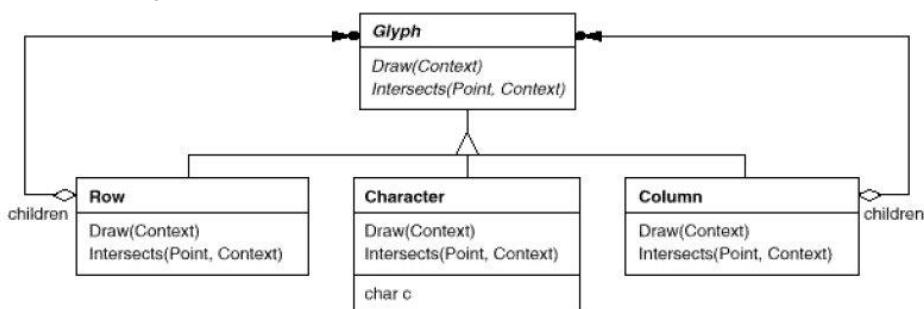


57

## YAPISAL (STRUCTURAL) KALIPLAR

### FLYWEIGHT:

- Örnek çözüm:

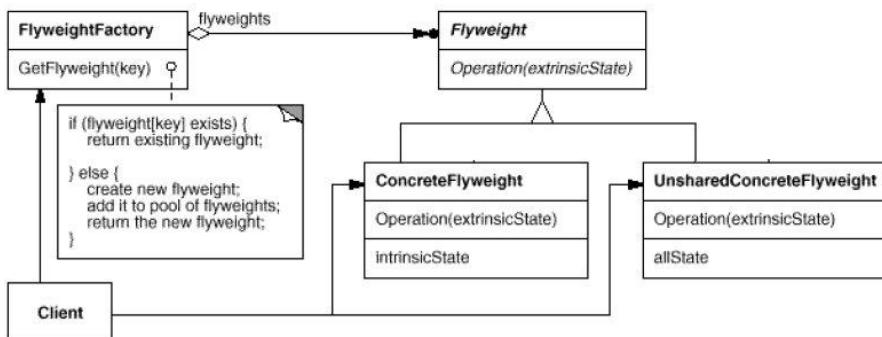


58

## YAPISAL (STRUCTURAL) KALIPLAR

### FLYWEIGHT:

- Kalıp yapısı:



59

## YAPISAL (STRUCTURAL) KALIPLAR

### FLYWEIGHT:

- Gerçekleme ayrıntıları:

- Paylaşılacak nesneleri üretmek üzere bir fabrika nesnesi oluşturulabilir, hatta oluşturulmalıdır da.
  - Aksi halde istemciler paylaşılan nesnenin birden fazla örneğini oluşturabilir ve bu da kalının amacı ile çelişir.
  - Flyweight nesneleri bir bakıma Singleton nesneleridir ancak fabrika nesnesi aynı anahtar için aynı nesnenin dönmesini sağlayacağından Singleton mekanizmasını ayrıca kurmaya gerek yoktur.
  - Erişimi iyice kısıtlamak için Flyweight nesnelerinin operation metodlarını bir arayüzde tanımlayıp Flyweight sınıfını fabrikanın private iç sınıfı olarak kodlayarak fabrikanın nesne işaretçisi yerine arayüz işaretçisi göndermesi sağlanabilir (örnekte kodlanacak).

60

## YAPISAL (STRUCTURAL) KALIPLAR

### FLYWEIGHT:

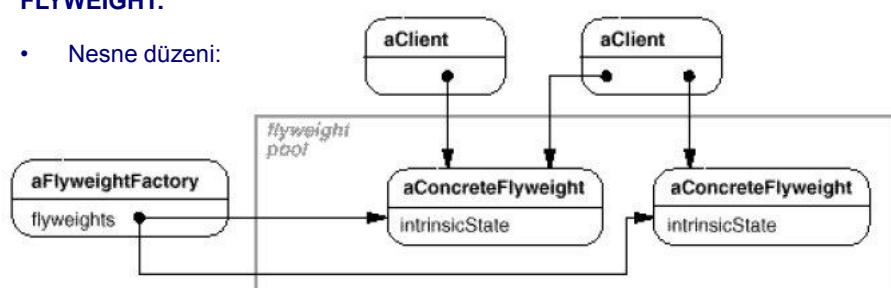
- Gerçekleme ayrıntıları (devam):
  - Kalının etkili olabilmesi için, paylaşılacak nesnelerin durum bilgisinin büyük bir kısmı çalışma alanında hesaplanabilir olmalıdır.
    - Bu sayede flyweight nesnelerinin saklanması gereken durum bilgisi çok azalır ve kalının etkinliği artar.
    - Saklanması gereken durum bilgisine içsel (intrinsic), hesaplanabilen ve böylece harici bir nesneden bu nesneye bir metod çağrı ile aktarılabilen durum bilgisine ise dışsal (extrinsic) adı verilir.
    - Alternatif terminoloji: intrinsic → immutable, extrinsic → mutable
      - İçsel kısmı oluşturan üyeler final olarak tanımlanabilir.
  - Her flyweight nesnesinin paylaşılması gerekmeyebilir.

61

## YAPISAL (STRUCTURAL) KALIPLAR

### FLYWEIGHT:

- Nesne düzeni:



- Kalıp bileşenleri:

- Flyweight: Paylaşılacak nesnelerin arayüzüne tanımlar.
- FlyweightFactory: Paylaşılacak nesneleri üretir.
- Client: Flyweight nesnelerine olan referansları idare eder ve bunların dışsal durumlarını hesaplar (veya flyweight'leri tam anlamıyla nesne yapmaktan çok daha verimli bir biçimde saklar).

62

## YAPISAL (STRUCTURAL) KALIPLAR

### FLYWEIGHT:

- Kalıbin kullanılabileceği anlar:
  - Şu koşulların tümünün doğru olması gerekmektedir:
  - Bir uygulamanın çok fazla sayıda nesne kullandığı,
  - Nesnelerin çok fazla olması nedeniyle saklama masraflarının çok yüksek olması,
  - Nesnelerin durum bilgisinin çoğunun dışsallaştırılabilmesi,
  - Dışsal durumları ayrılan nesnelerin bir çok grubunun yerini, paylaşabilecek az sayıda nesnenin alabileceği,
  - Uygulamanın nesne kimliğine bağlı olmadığı anlarda.

63

## ÖRNEK KALIP GERÇEKLEMELERİ – FLYWEIGHT

### ÇÖZÜLECEK PROBLEM:

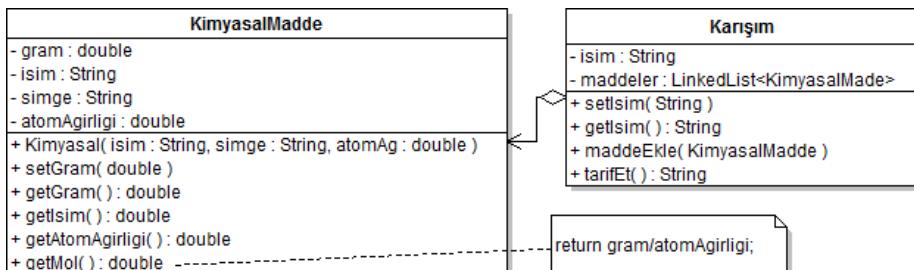
- Bir kimya fabrikasının üretim bilgi sistemi hazırlanıyor.
- Fabrikada bir çok kimyasal madde üretilmektedir.
- Fabrikada birkaç kimyasal maddenin birleşiminden oluşan çok sayıda karışım da hazırlanmaktadır.
- Aynı madde doğal olarak birden fazla karışımda yer almaktedir.

- Kaynak: DP Java Workbook p128

64

## ÖRNEK KALIP GERÇEKLEMELERİ – FLYWEIGHT

### İLK ÇÖZÜM:



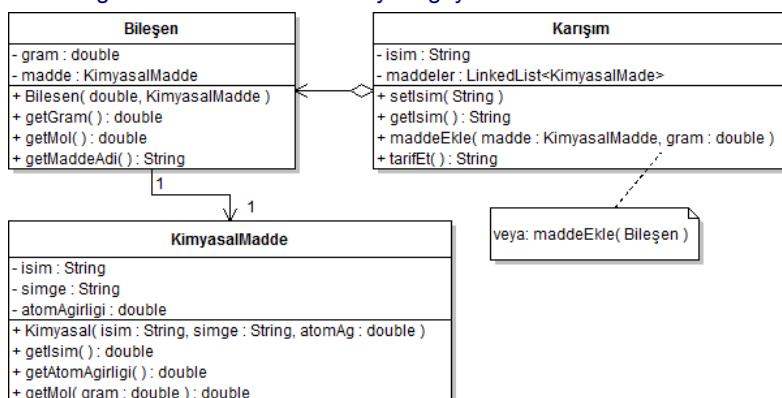
- Örneğin 100gr barutta 75gr potasyum nitrat, 15gr karbon, 10gr kükürt bulunur.
- Yukarıdaki çözüm bu tür bileşikleri destekler ancak örneğin karbon başka gramajla başka karışımlarda da bulunacaktır.
- Sadece gram bilgisi farklı olan, kalan durum bilgisi aynı olan çok sayıda karbon nesnesi mi oluşturulmalıdır?
- Gram bilgisi madde sınıfından ayrılmalı, ayrı bir sınıfta yer almalıdır.

65

## ÖRNEK KALIP GERÇEKLEMELERİ – FLYWEIGHT

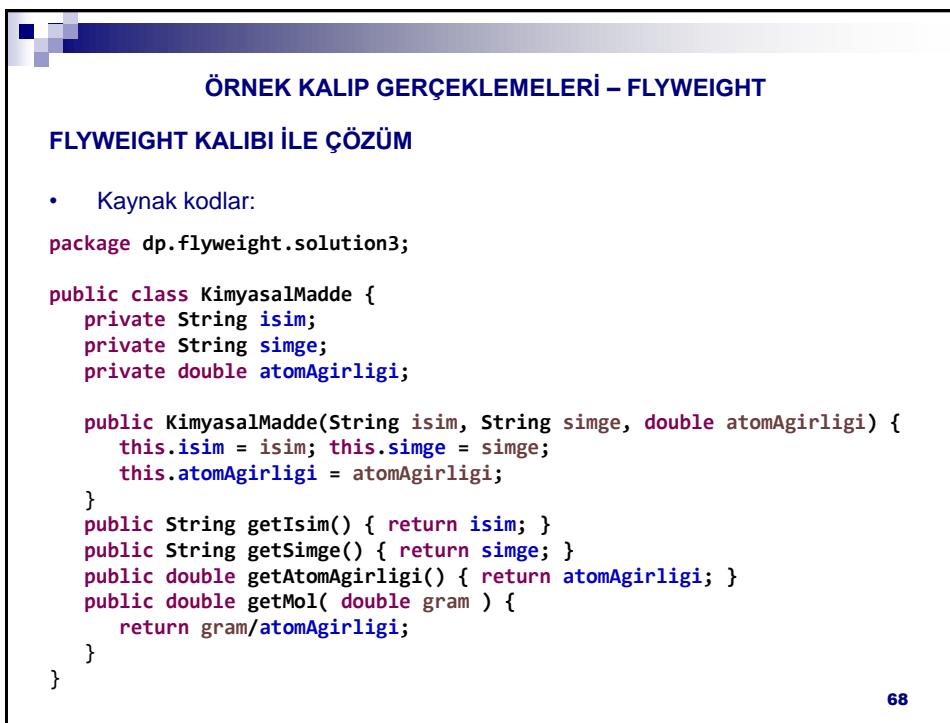
### DAHA DOĞRU MODELLEME:

- Gram bilgisinin madde sınıfından ayrıldığı yeni tasarım:



- Bu noktada gram, dışsal (extrinsic) bilgi olarak dikkatimizi çeker.
- Çok sayıda madde nesnesi olacağını da düşünüp Flyweight kalıbına yöneliriz.

66



## ÖRNEK KALIP GERÇEKLEMELERİ – FLYWEIGHT

### FLYWEIGHT KALIBI İLE ÇÖZÜM

- Kaynak kodlar:

```
package dp.flyweight.solution3;
import java.util.*;
public class MaddeFabrikasi {
    private static HashMap<String, KimyasalMadde> maddeler = new
        HashMap<String, KimyasalMadde>();
    static {
        maddeler.put( "C", new KimyasalMadde("Karbon", "C", 12.0107) );
        maddeler.put( "S", new KimyasalMadde("Kükürt", "S", 32.066) );
        maddeler.put( "KNO3", new KimyasalMadde("Potasyum Nitrat", "KNO3",
            101.103) );
        maddeler.put( "NaNO3", new KimyasalMadde("Sodyum Nitrat", "NaNO3",
            84.9947) );
    }
    public static KimyasalMadde maddeBul( String simge ) {
        return maddeler.get(simge);
    }
}
```

69

## ÖRNEK KALIP GERÇEKLEMELERİ – FLYWEIGHT

### FLYWEIGHT KALIBI İLE ÇÖZÜM

- Kaynak kodlar:

```
package dp.flyweight.solution3;

public class BileSEN {
    private double gram;
    private KimyasalMadde madde;

    public BileSEN( double gram, KimyasalMadde madde ) {
        this.gram = gram; this.madde = madde;
    }
    public double getGram( ) { return gram; }
    public double getMol( ) { return madde.getMol(gram); }
    public String getMaddeAdi( ) { return madde.getIsim( ); }
}
```

70

## ÖRNEK KALIP GERÇEKLEMELERİ – FLYWEIGHT

### FLYWEIGHT KALIBI İLE ÇÖZÜM

- Kaynak kodlar:

```
package dp.flyweight.solution3;
import java.util.*;
public class Karisim {
    private String isim;
    private LinkedList<Bilesen> maddeler = new LinkedList<Bilesen>();
    public String getIsim() { return isim; }
    public void setIsim(String isim) { this.isim = isim; }
    public void maddeEkle( String simge, double gram ) {
        maddeler.add( new Bilesen(gram, MaddeFabrikasi.maddeBul(simge)) );
    }
    public String tarifEt( ) {
        String tarif = isim;
        for( Bilesen madde : maddeler )
            tarif += "\n" + madde.getGram() + "gr " + madde.getMaddeAdi();
        return tarif;
    }
}
```

71

## ÖRNEK KALIP GERÇEKLEMELERİ – FLYWEIGHT

### FLYWEIGHT KALIBI İLE ÇÖZÜM

- Kaynak kodlar:

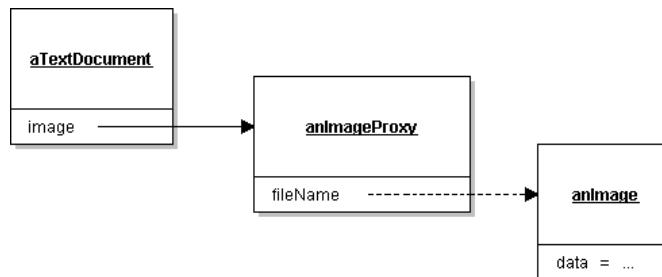
```
package dp.flyweight.solution3;
public class MainApp {
    public static void main(String[] args) {
        Karisim karaBarut = new Karisim();
        karaBarut.setIsim("Kara barut");
        karaBarut.maddeEkle("NaNO3", 75);
        karaBarut.maddeEkle("C", 15);
        karaBarut.maddeEkle("S", 10);
        System.out.println( karaBarut.tarifEt() );
        Karisim temizBarut = new Karisim();
        temizBarut.setIsim("Kara barut");
        temizBarut.maddeEkle("KNO3", 75);
        temizBarut.maddeEkle("C", 15);
        temizBarut.maddeEkle("S", 10);
        System.out.println( temizBarut.tarifEt() );
    }
}
```

72

## YAPISAL (STRUCTURAL) KALIPLAR

### PROXY:

- Amaç:
  - Bir nesneye erişimi denetlemek için, bu nesnenin yerine geçecek bir başka nesneyi araya koymak.
- Örnek:
  - Ekrana bir resim çizek zaman alıcı bir işlem olabilir.
  - Bu nedenle kelime işlem programları belgedeki resimleri ancak ilgili sayfa ekranda gösterileceği zamanda yükler ve çizer.

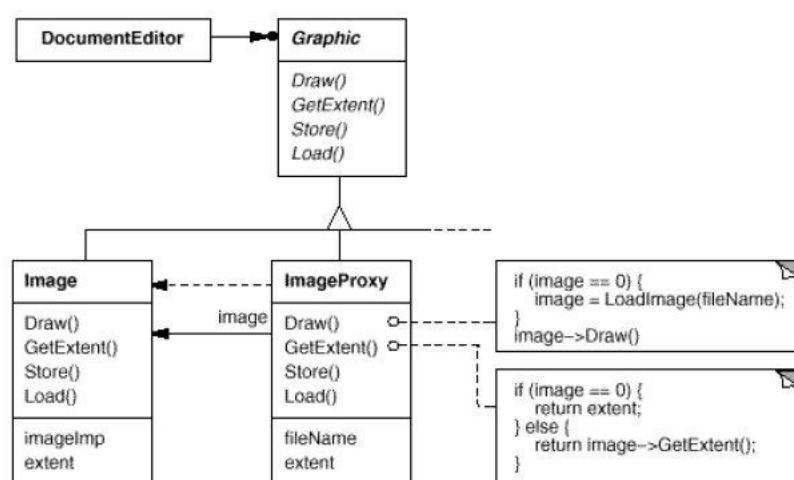


73

## YAPISAL (STRUCTURAL) KALIPLAR

### PROXY:

- Örnek çözüm:

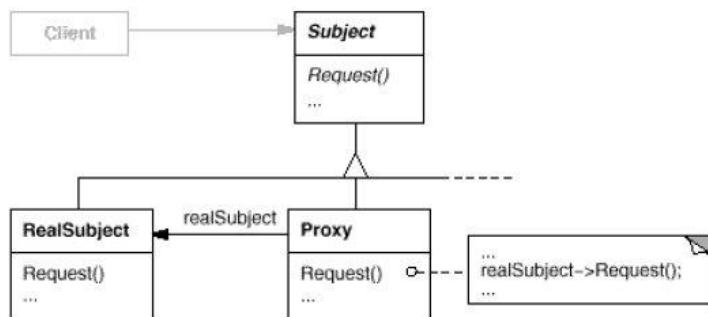


74

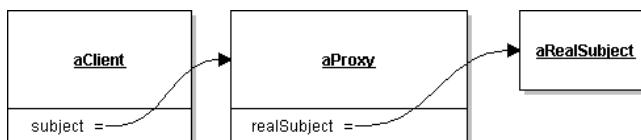
## YAPISAL (STRUCTURAL) KALIPLAR

### PROXY:

- Kalıp yapısı :



- Bellekteki durum :



75

## YAPISAL (STRUCTURAL) KALIPLAR

### PROXY:

- Kalıp bileşenleri:
  - RealSubject: Erişimi denetlenecek veya oluşturulması uzun sürecek olan asıl nesne
  - Proxy: Asıl nesnenin yerine geçecek olan vekil.
  - Subject: Vekilin asının yerine geçebilmesi için, vekil ve asıl nesneler için ortak bir arayüz sunar.
- Kalıbın uygulanabileceği anlar:
  - Herhangi bir nesneye, basit bir işaretçiden daha fazlasının gereği her durumda kullanılabilir.
  - Gereken yetenekler proxy sınıfına kodlanır.
  - İşaret edilen nesne tamamen farklı bir adres uzayında bulunabilir.

76

## ÖRNEK KALIP GERÇEKLEMELERİ – PROXY

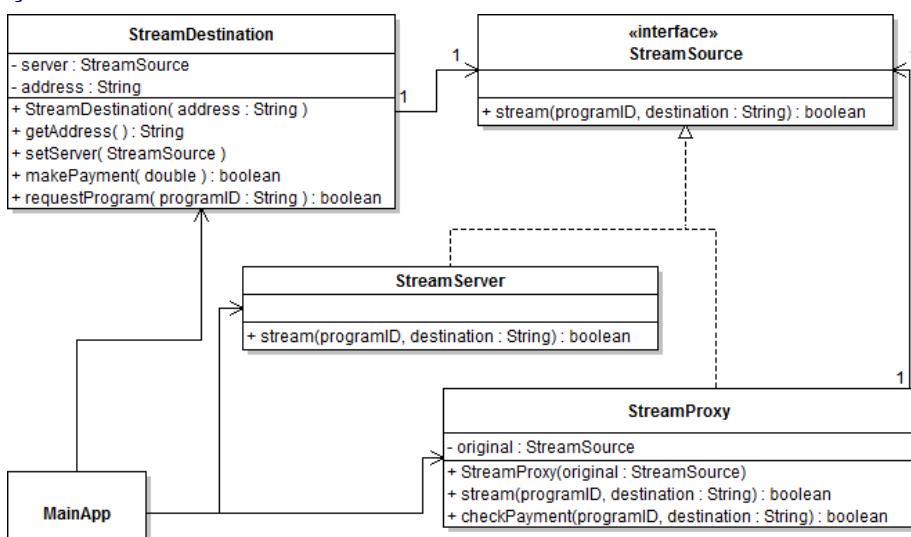
### ÇÖZÜLECEK PROBLEM:

- Biz izlediğin-kadar-öde sunucusuna gelen taleplerin emniyet altına alınması.
  - Sunucu kendisine verilen adrese istenilen programı gönderecek.
  - Ancak göndermeden önce istemcinin ödeme yapıp yapmadığını bakılmalı.
  - Sunucu yazılımında bu yetenek yok ve yazılımın kaynak koduna erişemiyoruz.
  - Araya koyacağımız bir vekil bu işi çözer.
    - (Protection proxy)

77

## ÖRNEK KALIP GERÇEKLEMELERİ – PROXY

### ÇÖZÜM:



78

## ÖRNEK KALIP GERÇEKLEMELERİ – PROXY

### ÇÖZÜM:

- Kaynak kodlar:

```
package dp.proxy.example;
public interface StreamSource {
    public boolean stream(String programID, String destination);
}

public class StreamServer implements StreamSource {
    public boolean stream(String programID, String destination) {
        // Programı hazırla ve gönder
        return true;
    }
}
```

79

## ÖRNEK KALIP GERÇEKLEMELERİ – PROXY

### ÇÖZÜM:

- Kaynak kodlar (devam):

```
public class StreamDestination {
    private StreamSource server;
    private String address;

    public StreamDestination(String address) {this.address = address; }
    public String getAddress() { return address; }
    public void setServer(StreamSource server) { this.server = server; }
    public boolean makePayment( double amount ) {
        //ödemeyi yap.
        return true;
    }
    public boolean requestProgram( String programID ) {
        return server.stream(programID, address);
    }
}
public class MainApp {
    public static void main(String[] args) {
        StreamServer realServer = new StreamServer();
        StreamProxy proxyServer = new StreamProxy(realServer);
        StreamDestination client = new StreamDestination("12:aa:34:ff:54");
        client.setServer(proxyServer);
    }
}
```

80

**NESNEYE DAYALI TASARIM VE MODELLEME  
KISIM 1: TASARIM KALIPLARI  
1.3. DAVRANIŞSAL (BEHAVIORAL) KALIPLAR**

1

**DAVRANIŞSAL (BEHAVIORAL) KALIPLAR**

**CHAIN OF RESPONSIBILITY:**

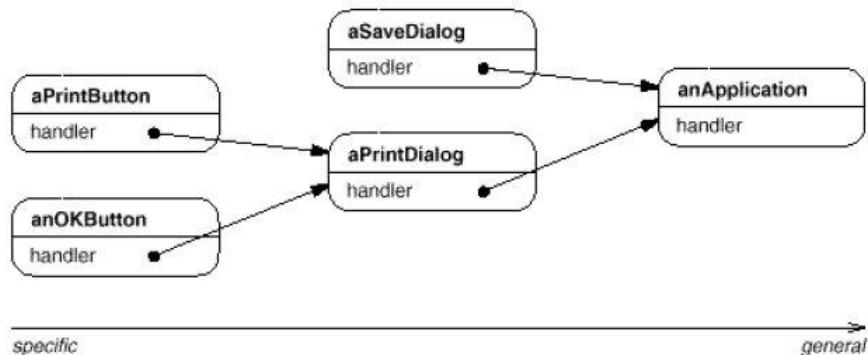
- Amaç:
  - Birbirleri ile ilişkili bir nesneler zinciri veya hiyerarşisindeki bir nesnenin, aldığı bir mesajı işleyemeyeceği durumlarda, bu mesajın gerekli işlemi yapabilecek bir başka nesneye iletmek.
  - Mesajın işlenmemesinin nedeni nesnenin arayüzünde o mesajın olmaması değil, nesnenin mesajı işleyecek yetki veya bilgiye sahip olmamasıdır.
- Örnek:
  - Kullanıcı bir dialog kutusundaki bileşenlerden birinin işlevi hakkında yardım almak istiyor.
  - Eğer o bileşene özel bir yardım içeriği varsa gösterilir, aksi halde yardım mesajı o bileşeni içeren bir üst bileşene ilettilir.
  - İletim mesaj bir nesne tarafından cevaplanana kadar veya zincirin sonuna gelinceye dek sürer.
  - Bir bileşen, hangi üst bileşen içinde yer aldığı bileyebilir.

2

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### CHAIN OF RESPONSIBILITY:

- Nesnelerin örnek durumu:
  - Ör: OK ve Print düğmeleri bir PrintDialog içinde bulunuyorlar.

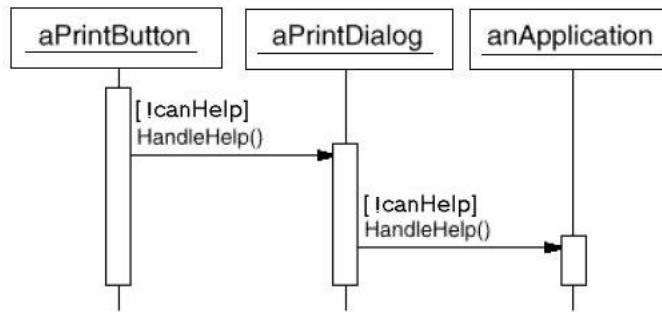


3

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### CHAIN OF RESPONSIBILITY:

- Çözüm:
  - Bileşenler mesajın iletimi için ortak bir arayüze sahip olmalıdır.
  - Aksi halde if-else/switch-case ile üst bileşenin tipini test edip uygun bir metodunu çağırmak gerekecektir.
  - Önerilen çözümde bir nesne sadece o mesajı yanıtlayıp yanıtlayamayacağını bilmek zorundadır.

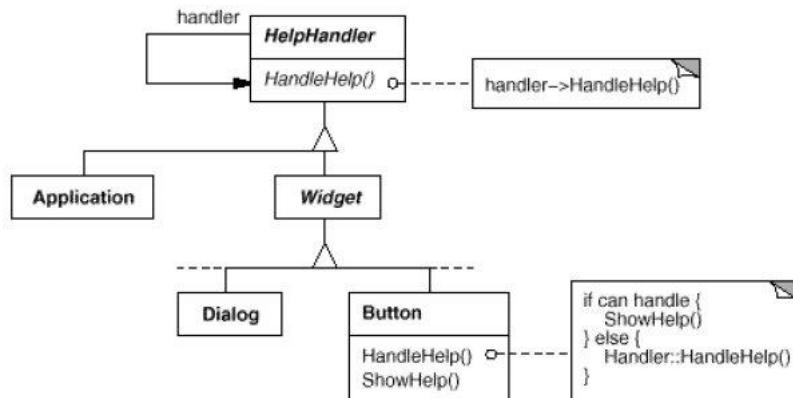


4

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

### CHAIN OF RESPONSIBILITY:

- Çözümü oluşturacak sınıf yapısı:

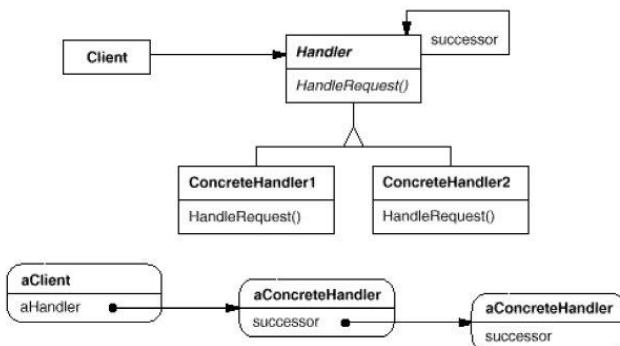


5

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

### CHAIN OF RESPONSIBILITY:

- Kalıp yapısı:



- Kalıp bileşenleri:

- Client: Yanıtlanacak mesajın kaynağı.
- Handler: Yanıtlanacak mesajı tanımlayan arayüz.
  - Zincir bağlantısını da (successor) gerçekleyebilir (Soyut sınıf olarak).
- Concrete HandlerX: Mesajı yanıtlar veya bağıltılı olduğu bir başkasına ileter.

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### CHAIN OF RESPONSIBILITY:

- Kalının uygulanabileceği anlar:
  - Bir mesajı yanıtlayabilecek birden fazla nesnenin bulunduğu ve yanıt verecek nesnenin önceden bilinmeyeceği anlarda.
  - Bir mesajı yanıtlayabilecek nesnelerin çalışma alanında tanımlanabileceği anlarda.
- Kalının zayıf yönleri:
  - Zincir doğru yapılandırılmazsa, bir mesajın yanıtlanacağının garantisı bulunmaz.
- Gerçekleme ayrıntıları:
  - Halihazırda bir zincir yoksa, zinciri oluşturacak metot(lar) Handler soyut sınıfında veya ConcreteHandler gerçeklemelerinde ayrıca kodlanır.
  - İsteğin birden fazla türden olabileceği durumlarda ya birden fazla HandleRequestX metotları tanımlanır, ya da isteği simgeleyen bir kalım hiyerarşisi kurulabilir.

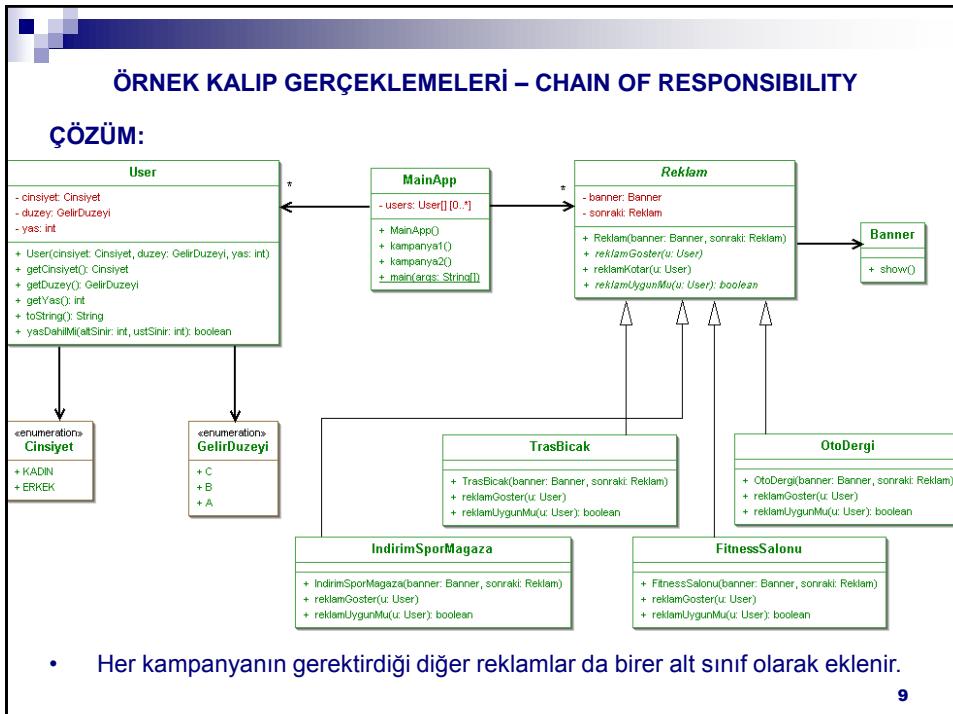
7

## ÖRNEK KALIP GERÇEKLEMELERİ – CHAIN OF RESPONSIBILITY

### ÇÖZÜLECEK PROBLEM:

- Bir reklam yönetim yazılımı hazırlanıyor.
  - Farklı web sitelerinde kampanyaların çeşitli kurallara göre çeşitli reklamlar gösterilmek isteniyor.
  - Kampanya 1:
    - Spor sitesinin 18-39 yaş arası erkek ziyaretçilere traş bıçağı reklamı göster, 18-39 yaş arası kadın ziyaretçilere fitness salonu reklamı göster, diğer ziyaretçilere spor mağazasının indirim kampanyasının reklamını göster.
  - Kampanya 2:
    - Haber sitesinin 18-29 yaş arası A gelir grubu erkek ziyaretçilere otomobil dergisi reklamını göster, aksi halde spor sitesinin kampanyasını aynen uygula.
  - Kampanya 3:
    - Ziyaretçinin coğrafi bölgesi ve gelir düzeyine göre portföydeki lokanta reklamlarından birini göster.
  - ...

• Kaynak: YES 8



## ÖRNEK KALIP GERÇEKLEMELERİ – CHAIN OF RESPONSIBILITY

### ÇÖZÜM:

- Kaynak kodlar (devam)

```
package dp.chainOfResponsibility.example;
public class TrasBicak extends Reklam {
    public TrasBicak(Banner banner, Reklam sonraki) {
        super(banner, sonraki);
    }
    public void reklamGoster(User u) {
        //Gerekli komutlar
    }
    public boolean reklamUygunMu(User u) {
        if( u.getCinsiyet() == Cinsiyet.ERKEK )
            if( u.yasDahilMi(18, 39))
                return true;
        return false;
    }
}
```

11

## ÖRNEK KALIP GERÇEKLEMELERİ – CHAIN OF RESPONSIBILITY

### ÇÖZÜM:

- Kaynak kodlar (devam)

```
package dp.chainOfResponsibility.example;
public class FitnessSalonu extends Reklam {
    public FitnessSalonu(Banner banner, Reklam sonraki) {
        super(banner, sonraki);
    }
    public void reklamGoster(User u) {
        //Gerekli komutlar
    }
    public boolean reklamUygunMu(User u) {
        if( u.getCinsiyet() == Cinsiyet.KADIN )
            if( u.yasDahilMi(18, 39))
                return true;
        return false;
    }
}
```

12

## ÖRNEK KALIP GERÇEKLEMELERİ – CHAIN OF RESPONSIBILITY

### ÇÖZÜM:

- Kaynak kodlar (devam)

```
package dp.chainOfResponsibility.example;
public class OtoDergi extends Reklam {
    public OtoDergi(Banner banner, Reklam sonraki) {
        super(banner, sonraki);
    }
    public void reklamGoster(User u) {
        //Gerekli komutlar
    }
    public boolean reklamUygunMu(User u) {
        if( u.getCinsiyet() == Cinsiyet.ERKEK
            && u.getDuzey() == GelirDuzeyi.A )
            if( u.yasDahilMi(18, 29))
                return true;
        return false;
    }
}
```

13

## ÖRNEK KALIP GERÇEKLEMELERİ – CHAIN OF RESPONSIBILITY

### ÇÖZÜM:

- Kaynak kodlar (devam)

```
package dp.chainOfResponsibility.example;
public class IndirimSporMagaza extends Reklam {
    public IndirimSporMagaza(Banner banner, Reklam sonraki) {
        super(banner, sonraki);
    }
    public void reklamGoster(User u) {
        //Gerekli komutlar
    }
    public boolean reklamUygunMu(User u) { return true; }
}
```

- Her kampanyanın gerektirdiği diğer reklamlar da birer alt sınıf olarak eklenir.

14

## ÖRNEK KALIP GERÇEKLEMELERİ – CHAIN OF RESPONSIBILITY

### ÇÖZÜM:

- Kaynak kodlar (devam)

```
package dp.chainOfResponsibility.example;
public class MainApp {
    private User[] users;
    public MainApp() {
        users = new User[3];
        users[0] = new User( Cinsiyet.ERKEK, GelirDuzeyi.C, 23 );
        users[1] = new User( Cinsiyet.ERKEK, GelirDuzeyi.A, 53 );
        users[2] = new User( Cinsiyet.KADIN, GelirDuzeyi.B, 33 );
    }
    public void kampanyal() {
        Reklam[] reklamlar = new Reklam[3];
        reklamlar[2] = new IndirimSporMagaza(new Banner(), null);
        reklamlar[1] = new FitnessSalonu(new Banner(), reklamlar[2]);
        reklamlar[0] = new TrasBicak(new Banner(), reklamlar[1]);
        reklamlar[0].reklamKotar(users[0]);
        reklamlar[0].reklamKotar(users[1]);
        reklamlar[0].reklamKotar(users[2]);
    }
    //sonraki yansıda devam edecek
```

15

## ÖRNEK KALIP GERÇEKLEMELERİ – CHAIN OF RESPONSIBILITY

### ÇÖZÜM:

- Kaynak kodlar (devam)

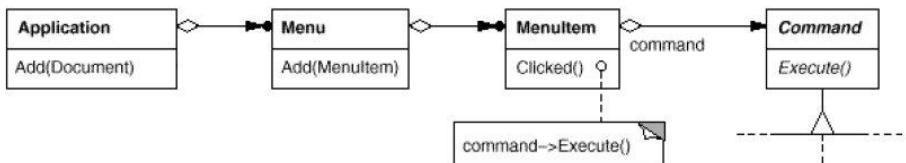
```
public void kampanya2() {
    Reklam[] reklamlar = new Reklam[4];
    reklamlar[3] = new IndirimSporMagaza(new Banner(), null);
    reklamlar[2] = new FitnessSalonu(new Banner(), reklamlar[3]);
    reklamlar[1] = new TrasBicak(new Banner(), reklamlar[2]);
    reklamlar[0] = new OtoDergi(new Banner(), reklamlar[1]);
    reklamlar[0].reklamKotar(users[0]);
    reklamlar[0].reklamKotar(users[1]);
    reklamlar[0].reklamKotar(users[2]);
}
public static void main(String[] args) {
    MainApp app = new MainApp();
    app.kampanyal();
    app.kampanya2();
}
} //end class
```

16

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### COMMAND:

- Amaç:
  - Nesnelere giden istekleri de birer nesne haline çevirmek.
  - Bir başka deyişle: Metotları nesne yapmak!
  - Böylece mesajları kuyruklaşmak veya mesajların güncesini tutmak gibi işlemler mümkün olabilir.
- Örnekler:
  - GUI kütüphanelerinde, örneğin bir menü seçeneği veya bir düğmeye tıklandığında herhangi bir nesnenin herhangi bir metodunu çalıştırırmak
  - Çözüm: 'Herhangi bir nesnenin herhangi bir metodu' yerine, belli bir arayüzün belli bir metodu çalıştırılır.

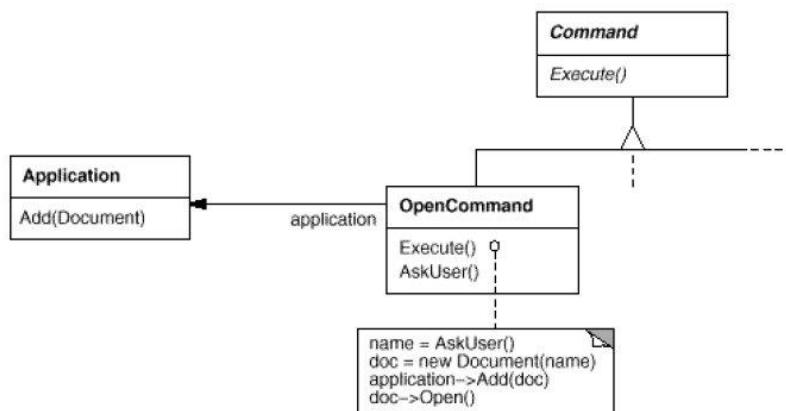


17

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### COMMAND:

- Çözümün mümkün kıldığı olasılıklar:
  - Bir menü seçeneği ile (MenuItem nesnesi) bir dosya açma komutu ilişkilendirilebilir.

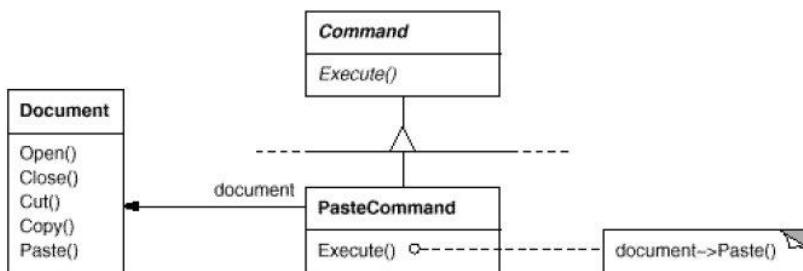


18

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### COMMAND:

- Çözümün mümkün kıldığı olasılıklar:
  - Önceki şekildeki Application sınıfı birden fazla Document örneği içerebilir.
  - Bunlardan biri üzerinde metin yapıştırma komutunun gerçekleşmesi:

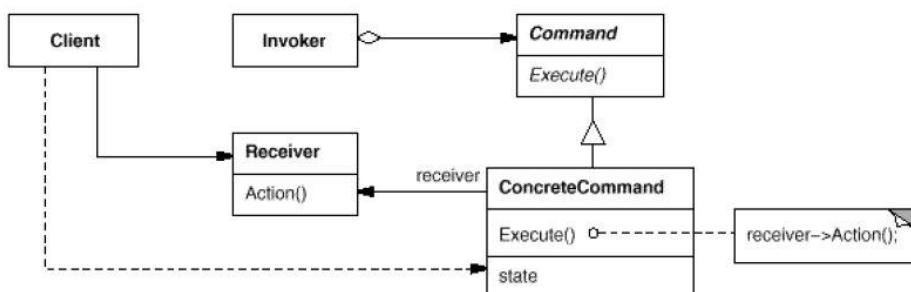


19

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### COMMAND:

- Kalıp yapısı:



20

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### COMMAND:

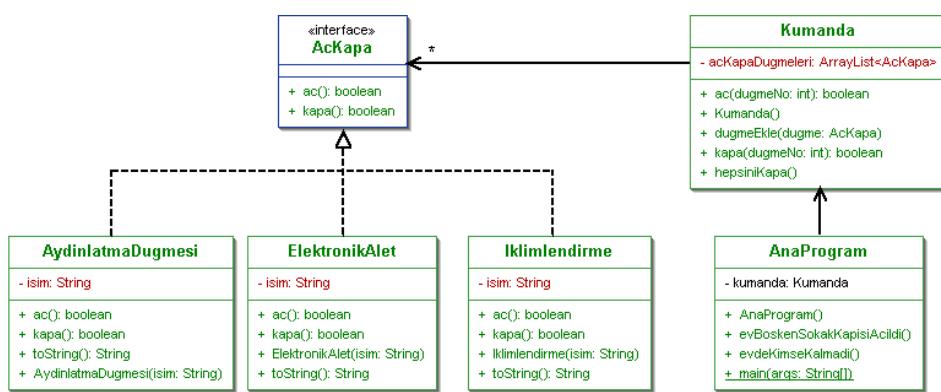
- Kalıp bileşenleri:
  - Command: Komut tanımlama arayüzü
  - ConcreteCommand: Komut gerçeklemeleri. Komutlar belli bir alıcının eylemlerini kullanacak şekilde gerçekleştirilebilir.
  - Client: Bir komut gerçeklemesi nesnesi oluşturur ve komutun alıcısını belirler.
  - Receiver: Alıcı nesne gerçeklemesi. Komut gerçeklemesi hakkında bilgi sahibi olması gerekebilir. Herhangi bir nesne alıcı olabilir, veya herhangi bir alıcı olmayan bir komut nesnesi de bulunabilir.

21

## ÖRNEK KALIP GERÇEKLEMELERİ – COMMAND

### ÇÖZÜLECEK PROBLEM:

- Bir akıllı ev yazılımı üretiyoruz. Yazılımda basit bir açma/kapama kumandası ekranı var. Kumanda ekranındaki aç/kapa düğmelerine odaların ışıklarını, televizyonları, vb. açma/kapama işlevleri kazandıracağız.



- Esinlenme: Head First DP

22

## ÖRNEK KALIP GERÇEKLEMELERİ – COMMAND

### ÇÖZÜM:

- Kaynak kodlar

```
package dp.command.example;
public interface AcKapa {
    public boolean ac();
    public boolean kapa();
}

package dp.command.example;
public class AydinlatmaDugmesi implements AcKapa {
    private String isim;
    public AydinlatmaDugmesi(String isim) { this.isim = isim; }
    public String toString() { return "AcKapa:" + isim; }
    public boolean ac() { return true; }
    public boolean kapa() { return true; }
}
```

23

## ÖRNEK KALIP GERÇEKLEMELERİ – COMMAND

### ÇÖZÜM:

- Kaynak kodlar (devam)

```
package dp.command.example;
import java.util.*;
public class Kumanda {
    private ArrayList<AcKapa> acKapaDugmeleri;
    public Kumanda() { acKapaDugmeleri = new ArrayList<AcKapa>(); }
    public void dugmeEkle(AcKapa dugme) { acKapaDugmeleri.add(dugme); }
    public boolean ac(int dugmeNo) {
        if( acKapaDugmeleri.get(dugmeNo) == null )
            return false;
        return acKapaDugmeleri.get(dugmeNo).ac();
    }
    public boolean kapa(int dugmeNo) {
        if( acKapaDugmeleri.get(dugmeNo) == null )
            return false;
        return acKapaDugmeleri.get(dugmeNo).kapa();
    }
    public void hepsiniKapa() {
        for( AcKapa dugme : acKapaDugmeleri )
            dugme.kapa();
    }
}
```

24

## ÖRNEK KALIP GERÇEKLEMELERİ – COMMAND

### ÇÖZÜM:

- Kaynak kodlar (devam)

```
public class AnaProgram {  
    Kumanda kumanda;  
    public AnaProgram( ) {  
        kumanda = new Kumanda();  
        kumanda.dugmeEkle( new AydinlatmaDugmesi( "Antre" ) );  
        kumanda.dugmeEkle( new AydinlatmaDugmesi( "Salon" ) );  
        kumanda.dugmeEkle( new AydinlatmaDugmesi( "Mutfak" ) );  
        kumanda.dugmeEkle( new AydinlatmaDugmesi( "Oda" ) );  
        kumanda.dugmeEkle( new ElektronikAlet( "Vestel TV" ) );  
        kumanda.dugmeEkle( new Iklimlendirme( "Kombi" ) );  
    }  
    public void evBoskenSokakKapisiAcildi( ) {  
        kumanda.ac(0); kumanda.ac(5);  
    }  
    public void evdeKimseKalmadi( ) { kumanda.hepsiniKapa( ); }  
    public static void main(String[] args) {  
        AnaProgram prg = new AnaProgram();  
        prg.evBoskenSokakKapisiAcildi();  
        System.out.println("Evden çıkışlıyor");  
        prg.evdeKimseKalmadi();  
    }  
}
```

25

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### ITERATOR:

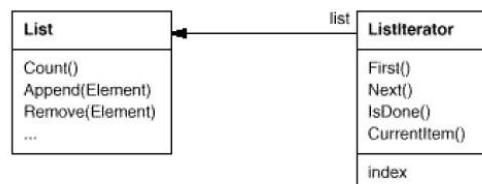
- Amaç:
  - Parçalardan oluşan bir bileşenin parçalarını,
  - bileşenlerin iç yapısını açığa çıkarmadan,
  - istenilen bir biçimde dolaşmak.
- Örnek:
  - Bir veri yapısı içerisinde dolaşmak.
- Sorun:
  - Veri yapısının arayüzüne birçok farklı dolaşım türü ile şışirmek istemeyiz.
- Not: Java'da herhangi bir Collection'dan, kendisini gezmek üzere, bir Iterator nesnesi istenebilir, ancak bu nesne Iterator tasarım kalıbına tam olarak uymaz
  - Java'nın Iterator'u daha basit.
  - Basit demek kötü demek değildir, çoğu durumda Java'nın Iterator'u da işimize yarar.
  - KISS: Java'nın Iterator'u işinize yarıyorsa onu kullanın.

26

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### ITERATOR:

- 1. çözüm:



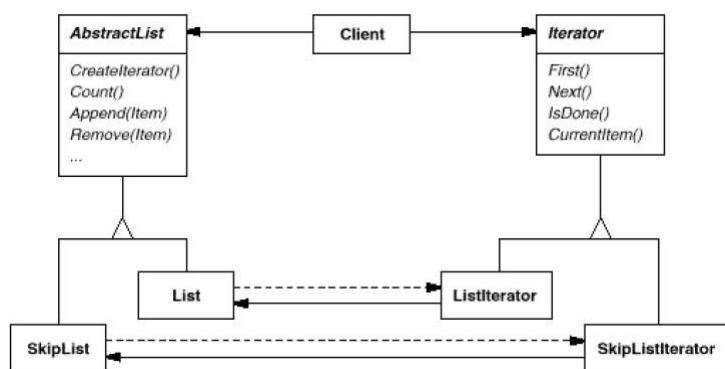
- Çözümün güçlü yönü:
  - Veri yapısı ile dolaşım biçimi birbirinden soyutlanmıştır.
- Çözümün zayıf yönü:
  - Belli bir veri yapısı ile belli bir dolaşım biçimi arasında bağılilik oluşmuştur.
  - Bu bağılılığı azaltmak üzere genel bir liste yapısı arayüzü ve genel bir dolaşım biçimi arayüzü oluşturulabilir.

27

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### ITERATOR:

- 2. çözüm:

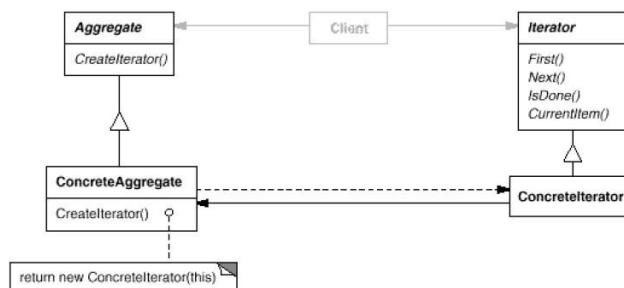


28

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### ITERATOR:

- Kalıp yapısı:



- Kalıp bileşenleri:

- Aggragate/ConcreteAggragate: Parçaları arasında dolaşılacak nesnenin arayüzü / gerçeklemesi.
- (Gezgin)Iterator/ConcreterIterator: Dolaşma işleminin arayüzü / gerçeklemesi.
- `java.util` paketindeki veri yapıları bu kalıbı kullanır.

29

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### ITERATOR:

- Gerçekleme ayrıntıları:
  - Parçalar arasında dolaşırken değiştirme işlemine izin verilecek mi?
  - Dolaşım algoritmasını kim gerçekleyecek?
    - Gezgin'e komutlar verme sırasını Client (istekçi) belirler ve parçaları gezginden tek tek okur.
  - İstekçi bir dolaşım istemini gezgin'e verir ve gezgin parçaları bir dizide döndürür.

30

## ÖRNEK KALIP GERÇEKLEMELERİ – ITERATOR

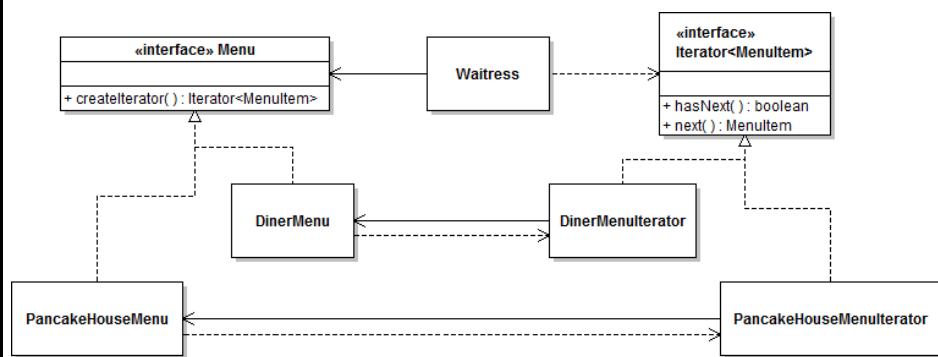
### ÇÖZÜLECEK PROBLEM:

- Online yemek siparişi verebileceğiniz bir firma hızla kurulmak isteniyor.
- Bu amaçla biri kahvaltılıkta, biri yemekte uzman iki firma satın alınıyor.
- Menü elemanları gerçeklemesi aynı denk gelmiş ama her firmanın kendi ayrı yemek menüsü gerçeklemesi var.
- Farklı menü gerçeklemelerinden yola çıkarak, birleştirilmiş menü müşterilere ortak bir kod üzerinden nasıl sunulacak?
  - Ortak kod söz konusu olmazsa, her yeni satın alınan firma için menü sunma koduna yeni bir kod bloğu eklenmek zorunda kalınacak!
  - İşler daha da çetrefil hale gelebilir:
    - İki firma da vejetaryenleri düşünüp hangi yemeklerinin et içermediğini işaretlemişler.
    - Bir vejetaryen menü sunmak için iki farklı firmanın menüsünü ayrı ayrı dolaşarak et içermeyenleri göstermek gerekecek.
  - Böylesi bir acemi çözümün kaynak kodu: Eclipse'ten.
- Kaynak: Head First DP

31

## ÖRNEK KALIP GERÇEKLEMELERİ – ITERATOR

### ITERATOR KALIBI İLE ÇÖZÜM:



32

## ÖRNEK KALIP GERÇEKLEMELERİ – ITERATOR

### ITERATOR KALIBI İLE ÇÖZÜM:

```
package iterator.example;
import java.util.*;
public class Waitress {
    private Menu pancakeHouseMenu, dinerMenu;
    public Waitress(Menu pancakeHouseMenu, Menu dinerMenu) {
        this.pancakeHouseMenu = pancakeHouseMenu; this.dinerMenu = dinerMenu;
    }
    public void printMenu() {
        Iterator<MenuItem> pancakeIterator = pancakeHouseMenu.createIterator();
        Iterator<MenuItem> dinerIterator = dinerMenu.createIterator();
        System.out.println("MENU\n----\nBREAKFAST");
        printMenu(pancakeIterator);
        System.out.println("\nLUNCH");
        printMenu(dinerIterator);
    }
    private void printMenu(Iterator<MenuItem> iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            printMenuItem(menuItem);
        }
    }
    private void printMenuItem(MenuItem menuItem) {
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + " -- ");
        System.out.println(menuItem.getDescription());
    }
}
```

33

## ÖRNEK KALIP GERÇEKLEMELERİ – ITERATOR

### ITERATOR KALIBI İLE ÇÖZÜM:

```
public void printVegetarianMenu() {
    System.out.println("\nVEGETARIAN MENU\n----\nBREAKFAST");
    printVegetarianMenu(pancakeHouseMenu.createIterator());
    System.out.println("\nLUNCH");
    printVegetarianMenu(dinerMenu.createIterator());
}

private void printVegetarianMenu(Iterator<MenuItem> iterator) {
    while (iterator.hasNext()) {
        MenuItem menuItem = iterator.next();
        if (menuItem.isVegetarian()) {
            printMenuItem(menuItem);
        }
    }
}
```

34

## ÖRNEK KALIP GERÇEKLEMELERİ – ITERATOR

### ITERATOR KALIBI İLE ÇÖZÜM:

```
public interface Menu {  
    public java.util.Iterator<MenuItem> createIterator();  
}  
public class DinerMenu implements Menu {  
    static final int MAX_ITEMS = 6; int numberOfItems = 0; MenuItem[] menuItems;  
    public DinerMenu() {  
        menuItems = new MenuItem[MAX_ITEMS];  
        addItem("Vegetarian BLT",  
                "(Fakin') Bacon with lettuce&tomato on whole wheat", true, 2.99);  
        addItem("Regular BLT",  
                "Bacon with lettuce & tomato on whole wheat", false, 2.99);  
        //add more items to menu  
    }  
    public String toString() { return "Diner Menu"; }  
    public void addItem(String name, String description,  
                       boolean vegetarian, double price) {  
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);  
        if (numberOfItems >= MAX_ITEMS) {  
            System.err.println("Sorry, menu is full! Can't add item");  
        } else { menuItems[numberOfItems] = menuItem; numberOfItems++; }  
    }  
    public MenuItem[] getMenuItems() { return menuItems; }  
    /*Iterator kalibini kullanan iyileştirilmiş tasarımdan gelen ek*/  
    public Iterator<MenuItem> createIterator() {  
        return new DinerMenuIterator(menuItems);  
    }  
}
```

35

## ÖRNEK KALIP GERÇEKLEMELERİ – ITERATOR

### ITERATOR KALIBI İLE ÇÖZÜM:

```
package dp.iterator.example;  
import java.util.*;  
public class PancakeHouseMenu implements Menu {  
    ArrayList<MenuItem> menuItems;  
    public PancakeHouseMenu() {  
        menuItems = new ArrayList<MenuItem>();  
        addItem("K&B's Pancake Breakfast",  
                "Pancakes with scrambled eggs, and toast", true, 2.99);  
        addItem("Regular Pancake Breakfast",  
                "Pancakes with fried eggs, sausage", false, 2.99);  
        //add more items to menu  
    }  
    public String toString() { return "Pancake House Menu"; }  
    public void addItem(String name, String description,  
                       boolean vegetarian, double price) {  
        MenuItem menuItem = new MenuItem(name, description, vegetarian, price);  
        menuItems.add(menuItem);  
    }  
    public ArrayList<MenuItem> getMenuItems() { return menuItems; }  
    /*Iterator kalibini kullanan iyileştirilmiş tasarımdan gelen ek*/  
    public Iterator<MenuItem> createIterator() {  
        return new PancakeHouseMenuIterator(menuItems);  
    }  
}
```

36

## ÖRNEK KALIP GERÇEKLEMELERİ – ITERATOR

### ITERATOR KALIBI İLE ÇÖZÜM:

```
package dp.iterator.example;
import java.util.*;
public class DinerMenuItemIterator implements Iterator<MenuItem>{
    MenuItem[] list; int position = 0;
    //implementing the interface by using arrays. Full version is in zip file.
}

package dp.iterator.example;
import java.util.*;
public class PancakeHouseMenuItemIterator implements Iterator<MenuItem> {
    ArrayList<MenuItem> items; int position = 0;
    //implementing the interface by using ArrayLists. Full version is in zip file.
}
```

37

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### MEMENTO:

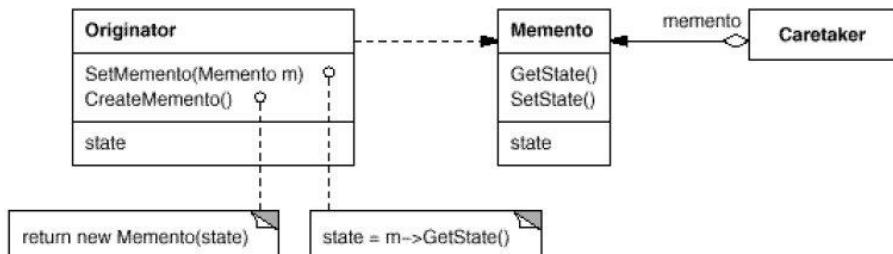
- Amaç:
  - Bir nesnenin durum bilgisini sarmalama (encapsulation) ilkesini bozmadan elde etmek.
  - Böylece nesnelerin ikincil saklama ortamlarında saklanması, geri yüklenmesi, ağ üzerinden aktarımı mümkün olabilir.
  - Yine bu şekilde sona yapılan işlemi geri alma düzeneği (undo) kurulabilir.
  - Durum bilgisinin sadece bir kısmının saklanması da yeterli olabilir.

38

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

### MEMENTO:

- Kalıp yapısı:



- Kalıp bileşenleri:

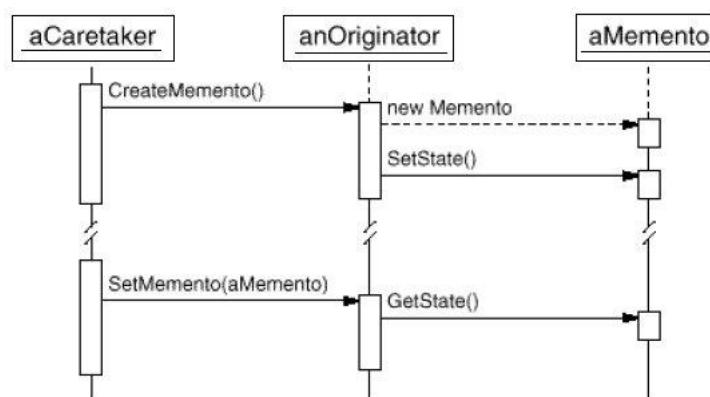
- Originator: Durum bilgisi saklanacak olan nesne
- Memento: Durum bilgisinin gerekli kısımlarını saklar
- Caretaker: Gerekli anlarda durum bilgisinin bir kopyasını alır ve elindeki kopyalardan birini geçerli durum haline getirir.

39

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

### MEMENTO:

- Kalıp bileşenlerinin etkileşimleri:



40

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### MEMENTO:

- Kalıbin uygulanabileceği anlar:
  - Nesnelerin durum bilgisinin tümü veya bir kısmının daha sonra o duruma dönebilme üzere saklanması VE
  - Nesneye doğrudan bir işaretçinin veri gizliliği ilkesini bozmasının istenmediği anlarda.
- Sorun:
  - Veri gizliliği ilkesini sağlamak için Memento'nun metodlarına sadece Originator erişebilmelidir.
  - Bu gereksinim her dilde karşılanamaz.
    - C++: Memento metodları private yapılır ve Originator sınıfı Memento sınıfının arkadaşı (friend) olarak tanımlanır.
    - Java: Memento metodları package yapılır, Memento ve Originator sınıfı aynı pakette yer alır. Böylece Caretaker sınıfı başka paketlerde yer alabilir, ya da aynı paketteki Caretaker sınıfına bir Memento veya State parametreli erişim metodları konulmaz.
    - Java diğer seçenek: Memento, Originator sınıfının iç sınıfı yapılır.
      - Ancak bu durumda State clonable yapılmalı ve Originator. createMemento metodu klon döndürmelidir.

41

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### MEMENTO:

- Dış sınıf ile çözüm: Önceki yanda anlatıldığı gibi kodlanır.
- İç sınıf ile çözüm:

```
package dp.mementoV2.innerClass;
public class State implements Cloneable {
    private String durum;
    public String getDurum() {
        return durum;
    }
    void setDurum(String durum) {
        this.durum = durum;
    }
    public State clone() {
        State s = new State();
        s.durum = new String(durum);
        return s;
    }
}
```

42

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### MEMENTO:

- İç sınıf ile çözüm (devam):

```
package dp.mementoV2.innerClass;
public class Originator {
    private State state;
    public Originator() { state = new State(); }
    public Memento createMemento() {
        Memento m = new Memento();
        m.setState(state.clone());
        return m;
    }
    public void setMemento(Memento m) { state = m.getState(); }
    public void modify(String str) { state.setDurum(str); }
    public String toString() { return state.getDurum(); }

    public class Memento {
        private State state;
        private State getState() { return state; }
        private void setState(State state) { this.state = state; }
    }
}
```

43

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### MEMENTO:

- İç sınıf ile çözüm (devam):

```
package dp.mementoV2Outer;
import dp.mementoV2.innerClass.*;
import dp.mementoV2.innerClass.Originator.Memento;

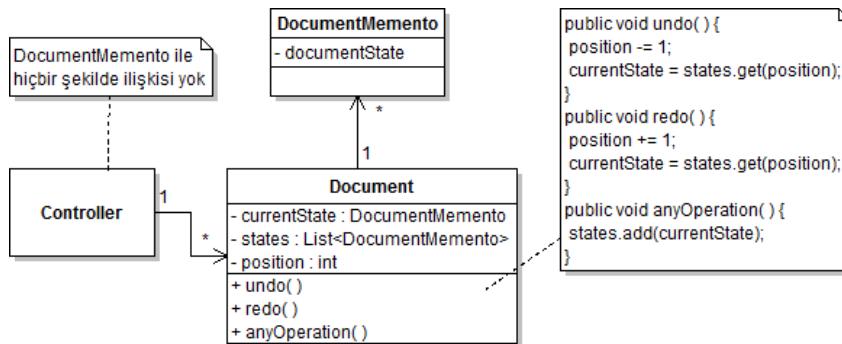
public class Caretaker {
    public void test() {
        Originator subject = new Originator();
        subject.modify("ilk durum");
        Memento mem = subject.createMemento();
        subject.modify("yeni durum");
        //bir süre sonra...
        subject.setMemento(mem);
        System.out.println(subject);
    }
}
public static void main(String[] args) {
    Caretaker ct = new Caretaker();
    ct.test();
}
```

44

## ÖRNEK KALIP GERÇEKLEMELERİ – MEMENTO

### ÇÖZÜLECEK PROBLEM:

- Bir belge işleme programında geri alma ve yineleme işlemleri

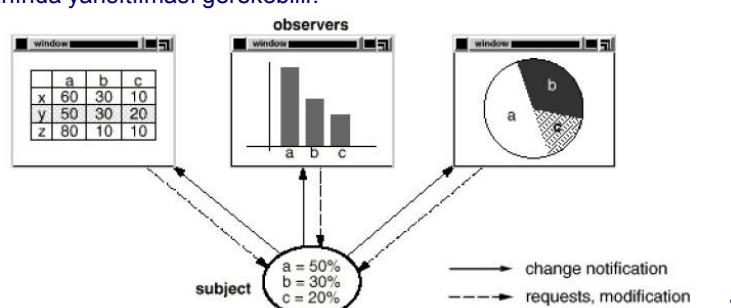


45

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### OBSERVER:

- Amaç:**
  - Bir nesnenin durumu değiştiğinde, bu nesne ile ilişkili tüm diğer nesnelerin durumdan haberdar edilmelerini ve güncellenmelerini sağlamak.
  - Aralarındaki ilişkilerin çalışma sırasında kurulması gereken nesnelerde bu iş nasıl yapılabilir?
- Örnek:**
  - Bir hesap tablosundaki bilgi değiştiğinde çeşitli çizelgelere de son durumun altında yansıtılması gerekebilir.

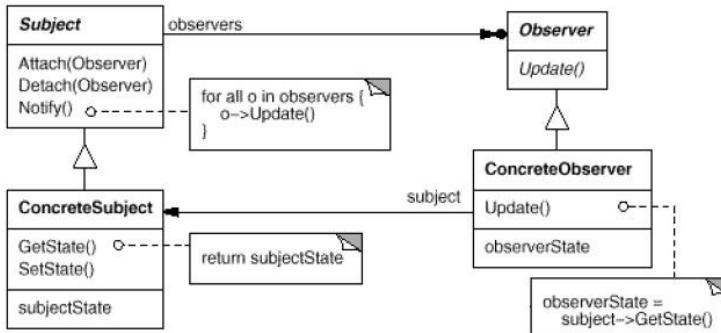


46

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

### OBSERVER:

- Kalıp yapısı:



- Kalıp bileşenleri:

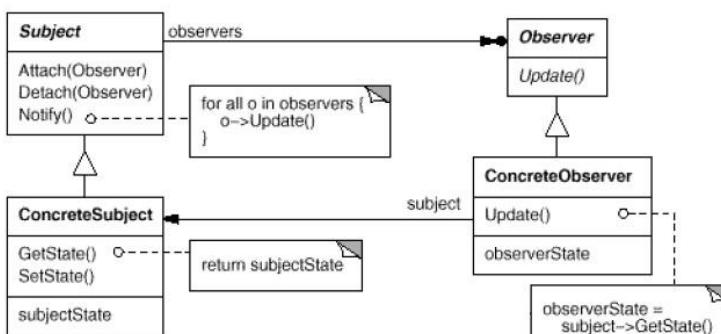
- **Subject**: Durumundaki değişikliklerin izleneceği nesnenin arayüzü.
  - Birden fazla gözleyicisi olabilir. Bu nedenle gözleyici ekle/sil metodlarını tanımlayan soyut bir sınıf olmalıdır.
- **Observer**: Gözleyici nesnelerin arayüzü.

47

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

### OBSERVER:

- Kalıp yapısı:



- Kalıp bileşenleri:

- **ConcreteSubject**: Subject gerçeklemeleri. Durumu değiştiğinde gözleyicilerini Observer arayüzüünü kullanarak bilgilendirir.
- **ConcreteObserver**: Observer gerçeklemeleri. Gözlediği nesneyi gösteren bir işaretçi tutabilir.

48

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### OBSERVER:

- Gerçekleme ayrıntıları :
  - Durum bilgisinin gözleyicilere gönderilmesi:
    - Pull model:
      - + Gözlenen sadece gözleyicileri haberdar eder, gözleyiciler ilgili durum bilgisini gözlenenden alır.
      - + Gözlenenin gözleyicilerle bağlantısını azaltan çalışma biçimidir.
      - + Gözleyici durum bilgisinin sadece istediği kısmını almayı seçebilir.
      - Gözleyici gözlenende nelerin değiştiğini kendisi çıkarmak zorunda kalır.
    - Push model: Gözlenen gözleyicilere durum değişikliği hakkında ayrıntılı bilgi gönderir.
  - Aynı türden gözlenen nesnelerin tümünün gözleyicilerinin ortak olması istenirse, gözlenen gözleyicilerini statik bir veri yapısı içerisinde tutulabilir.

49

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### OBSERVER:

- Gerçekleme ayrıntıları (devam):
  - Bir gözleyicinin birden fazla nesneyi gözlemesi istenebilir.
  - Gözlenen nesnenin durumundaki farklılıkların farklı türden gözleyicilerin izlemesi istenebilir.
  - Haber verme mesajının tetiklenmesi:
    - Gözleyiciler tetiklerse, gözlenenlerin istekçilerini kodlayan kişi tetikleme komutunu vermeyi unutabilir.
    - Gözlenen nesne tetiklerse, gözlenenlerin istekçilerinin peş peşe yapacağı durum değişiklikleri tek bir haber verme mesajı ile gönderilebileceği yerde birçok izleyiciye gidecek bir sürü mesaj gönderilmesine neden olabilir.
  - Kalıbin kullanılabileceği anlar:
    - Herhangi bir soyutlamadan birbirine bağımlı birden fazla parçası olduğunda.
    - Bir nesnedeki değişiklıkların derleme anında bilinmeyen sayıda nesneyi de etkilemesi gereken anlarda.
    - Bağlaşımı arttırmadan bir nesnenin diğer nesneleri herhangi bir durumdan haberdar etmesi istenen anlarda.

50

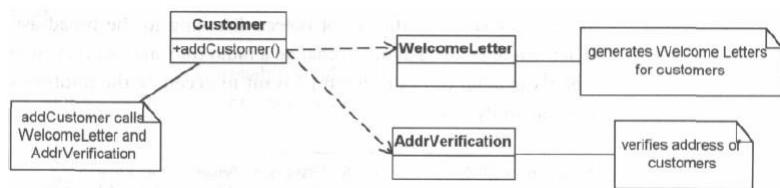
## ÖRNEK KALIP GERÇEKLEMELERİ – OBSERVER

### ÇÖZÜLECEK PROBLEM:

- Yazdığımız bir müşteri ilişkileri programına yeni müşteri eklendiğinde, müşteriyeye bir "hoş geldiniz" e-postası gönderilecek ve müşterinin ikamet adresinin doğru olup olmadığına bakılacak.

### ÇÖZÜM 1:

- Bu davranışlar sabit bir şekilde kodlanır.



- Peki ya gereksinimler değişirse?

51

## ÖRNEK KALIP GERÇEKLEMELERİ – OBSERVER

### ÇÖZÜM 2:

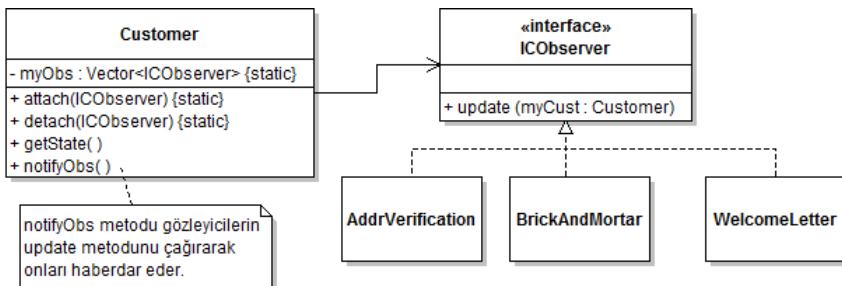
- Yeni bir gereksinim ortaya çıkıyor:
  - Müşteri bir şubemizin 10km. yakınında ise ona gönderilecek mektupta indirim kuponları da olsun.
  - Benzer başka gereksinimler de ortaya çıkabileceğini görüp, şimdiden önlemimizi alalım.
  - Sisteme her yeni müşteri eklendiğinde, bir veya birkaç farklı işlem yapmak üzere bir alt yapı oluşturalım.
  - Müşteri eklendiği zaman sistem yapması gereken işler olabileceğiinden haberdar edilsin.
  - Bu düşünce biçimini observer kalıbına götürür.
  - Brick-and-mortar: Ağırlıklı olarak e-ticaret yapanlar, fiziksel binalarda yaptıkları ticareti böyle adlandırır.

52

## ÖRNEK KALIP GERÇEKLEMELERİ – OBSERVER

### ÇÖZÜM 2:

- Observer kalibinin kullanılması:



- Sadece müşteri nesneleri izlenecekse bir soyut 'Subject' üst sınıfı hazırlamaya gerek kalmadan observer kalibi kullanılabilir.

53

## ÖRNEK KALIP GERÇEKLEMELERİ – OBSERVER

### ÇÖZÜM 2:

```
package observer.example;
public interface ICObserver {
    public void update (Customer myCust);
}
public class AddrVerification implements ICObserver {
    public void update(Customer myCust) {
        String state = myCust.getState();
        // TODO Gerekli işlemleri tamamla
    }
}
public class WelcomeLetter implements ICObserver {
    public void update(Customer myCust) {
        String state = myCust.getState();
        // TODO Gerekli işlemleri tamamla
    }
}
public class BrickAndMortar implements ICObserver {
    //diğerleri gibi kodla
}
```

54

## ÖRNEK KALIP GERÇEKLEMELERİ – OBSERVER

### ÇÖZÜM 2:

```
package observer.example;
import java.util.*;
public class Customer {
    private static Vector<ICOserver> myObs =
        new Vector<ICOserver>() ; //Gözleyiciler ortak olacak
    public static void attach(ICOserver o){
        myObs.addElement(o);
    }
    public static void detach(ICOserver o){
        myObs.remove(o);
    }
    public String getState () {
        // TODO: Burada durum bilgisini döndür
        return null;
    }
    public void notifyObs () {
        for( ICOserver obs : myObs ) {
            obs.update(this);
        }
    }
}
```

55

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### STRATEGY:

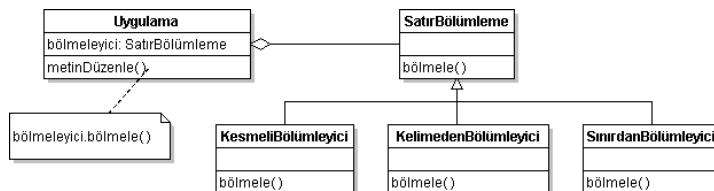
- Amaç:
  - Aynı iş için olan farklı algoritmalar ile bu işin yapılmasını isteyecek istemcileri birbirlerinden soyutlamak.
  - Böylece kullanılan algoritmayı çalışma alanında değiştirilebilir.
- Örnek:
  - Metinlerin gösteriminde kullanılabilecek alt satırda geçme (line breaking) algoritmaları çok çeşitlidir.
  - Bu algoritmaları gerçekleştirmek istiyoruz, ancak:
    - Bunları farklı metin uygulamalarında kullanabilmek istiyoruz.
    - Bunları aynı uygulamada istediğimiz anda birbirlerinin yerine kullanabilmek istiyoruz.

56

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### STRATEGY:

- Örnek çözüm:

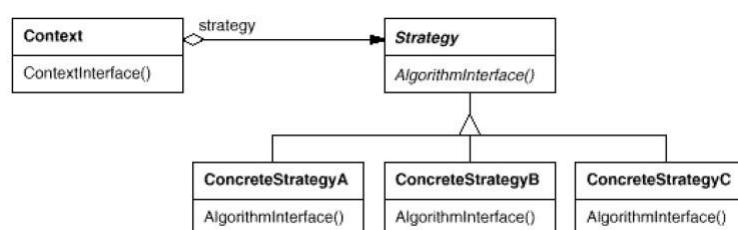


57

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### STRATEGY:

- Kalıp yapısı:



- Kalıp bileşenleri:

- Strategy: Desteklenecek tüm algoritmaların ortak arayüzüne belirler.
- ConcreteStrategyX: Algoritma gerçeklemeleri.
- Context: Algoritma ihtiyaç duyan nesne
  - Bir/birkaç ConcreteStrategy nesnesi ile ilişkilendirilir.
  - `ContextInterface()`: Gerekli hallerde algoritma gerçeklemesine durum bilgisini açan metod(lar).

58

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

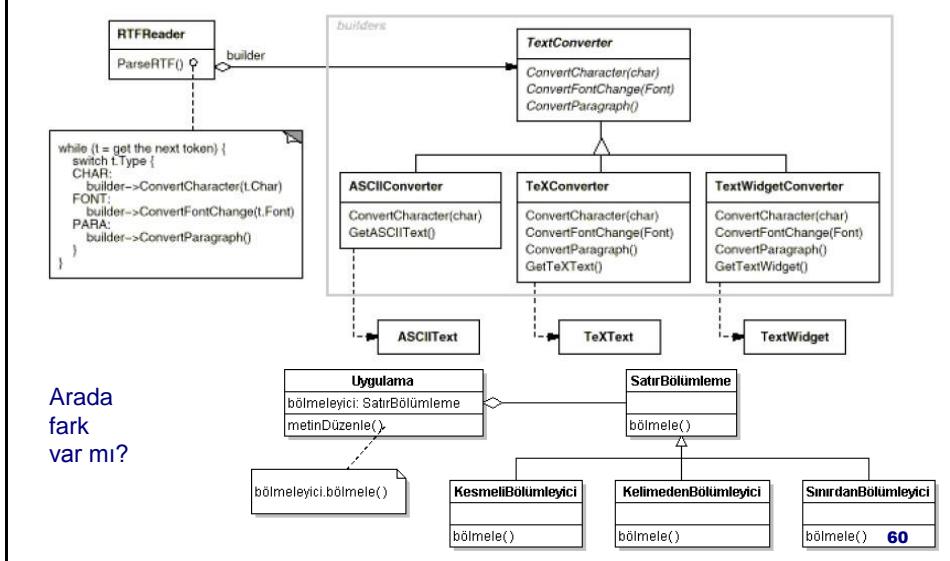
### STRATEGY:

- Gerçekleme ayrıntıları:
  - Strategy ve Context nesneleri seçilen algoritmayı gerçeklemek için etkileşimlerde bulunur.
  - Context nesnesini kullanacak istekçi nesneler Context nesnesine bir ConcreteStrategy nesnesi sağlar.
- Kalıbin uygulanabileceği anlar:
  - Bir algoritmanın değişik çeşitlemelerine ihtiyaç duyulduğunda.
  - İstekçilerin kullanılacak algoritmanın gerçekleştirmesini bilmemesi gerekiği hallerde.
  - Sadece davranışlarında değişiklik gösteren birçok ilişkili sınıf bulunduğuanda.
- Kalıbin zayıf yönleri:
  - İstekçiler değişik stratejilerin varlığından haberdar olmak zorundadırlar.
  - Context'in açtığı durum bilgisinin tümüne her tür stratejinin ihtiyacı olmayabilir (haberleşmede ek yük).

59

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

### STRATEGY ve BUILDER KALIPLARI:



## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### STRATEGY ve BUILDER KALIPLARI:

- Tasarım kalıpları benzer sınıf yapılarına sahip olabilir.
- Kalıpları farklılaştırılan amaçları, kullanım yerleri ve gerçekleme ayrıntılarıdır.
- Örneğimizde ilişkilerin elmas tarafındaki nesne:
  - Builder kalıbında parçaları nasıl oluşturup birleştireceğini bilir ve denetler.
  - Strateji kalıbında işlerin nasıl yürütüldüğünü bilmez.

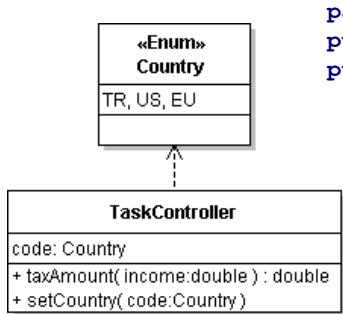
61

## ÖRNEK KALIP GERÇEKLEMELERİ – STRATEGY

### ÇÖZÜLECEK PROBLEM:

- Uluslar arası pazara hitap eden bir kişisel finans programı yazılıyor.
- Değişik ülkelerdeki değişik vergi kuralları ile işlem yapılması gerekmektedir.
- Ana programımız: TaskController

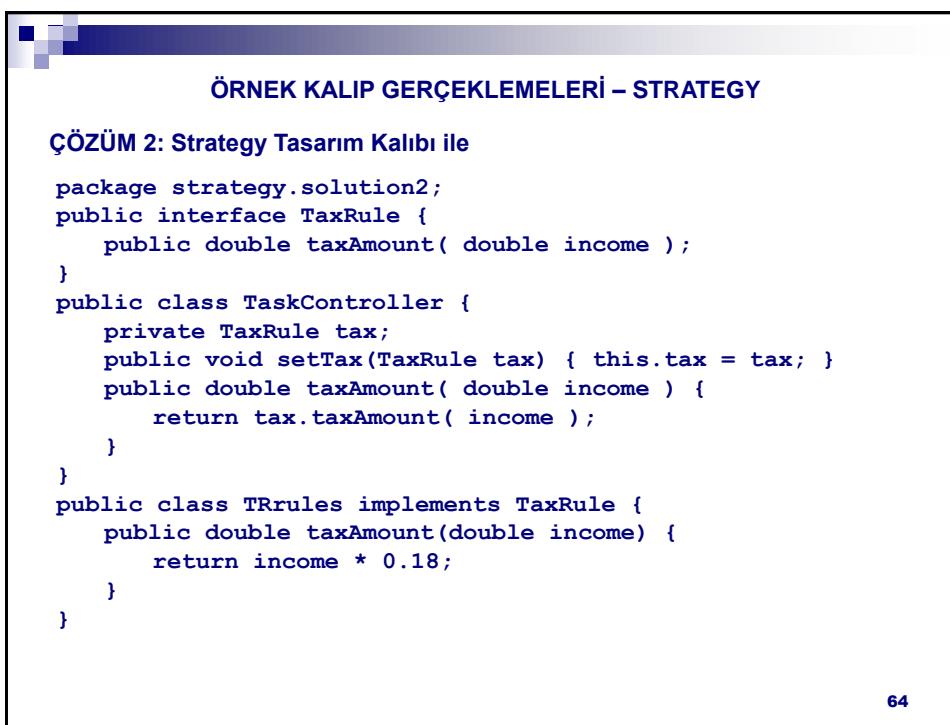
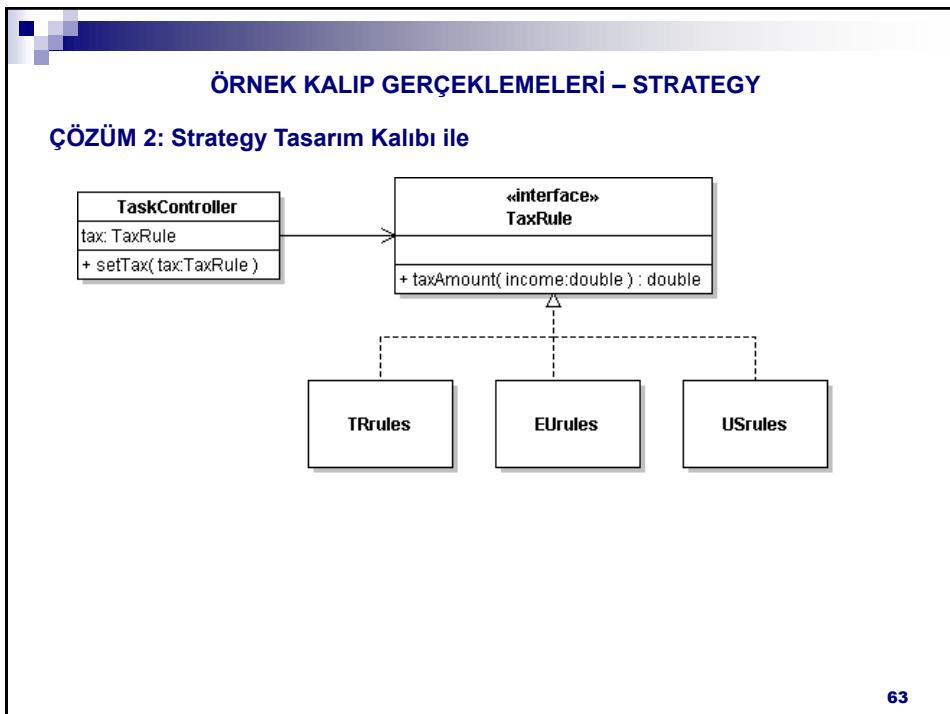
### ÇÖZÜM 1: switch-case



```
package strategy.solution1;
public enum Country { TR, US, EU }
public class TaskController {
    private Country code;
    public void setCountry(Country code) {
        this.code = code;
    }
    public double taxAmount(double income) {
        switch (code) {
            case TR: return income * 0.18;
            case EU: return income * 0.15;
            case US: return income * 0.12;
            default: return income * 0.10;
        }
    }
}
```

- Kaynak: DP Explained

62



## ÖRNEK KALIP GERÇEKLEMELERİ – STRATEGY

### ÇÖZÜM 2: Strategy Tasarım Kalıbı ile

```
public class EURules implements TaxRule {  
    public double taxAmount(double income) {  
        return income * 0.15;  
    }  
}  
  
public class USRules implements TaxRule {  
    public double taxAmount(double income) {  
        return income * 0.12;  
    }  
}
```

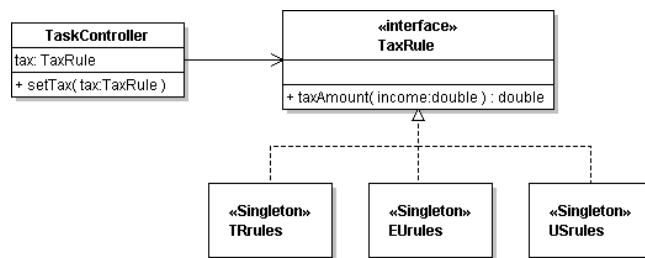
- Kalıp asıl esnekliğini vergi hesaplamanın daha karmaşık kurallara dayandığı zaman gösterir ki, gerçek hayatı da durum öyledir.
- Diğer mali işlemler için Strategy kalımı tekrarlanır.

65

## ÖRNEK KALIP GERÇEKLEMELERİ – SINGLETON

### SINGLETON ve STRATEGY BİRLİKTELİĞİ:

- Kişisel finans programında farklı ülkeler için farklı kurallar olabilir, ancak belli bir ülke için sadece tek bir kural bulunur.
- Bu gereksinimi karşılamak üzere Singleton kalımı kullanılabilir:
  - TRrules, EURules, USrules sınıfları Singleton olarak gerçekleştirilebilir.



66

## ÖRNEK KALIP GERÇEKLEMELERİ – SINGLETON

### ÇÖZÜM:

```
package singleton.example;
public class TRrules implements TaxRule {
    private static TRrules instance;
    private TRrules() {
        //gerekli ilklenirmeler
    }
    public static TRrules getInstance() {
        if( instance == null )
            instance = new TRrules();
        return instance;
    }
    public double taxAmount(double income) {
        return income * 0.18;
    }
}
```

- EUrules ve USrules sınıfları da benzer şekilde gerçeklenir.

67

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### STATE:

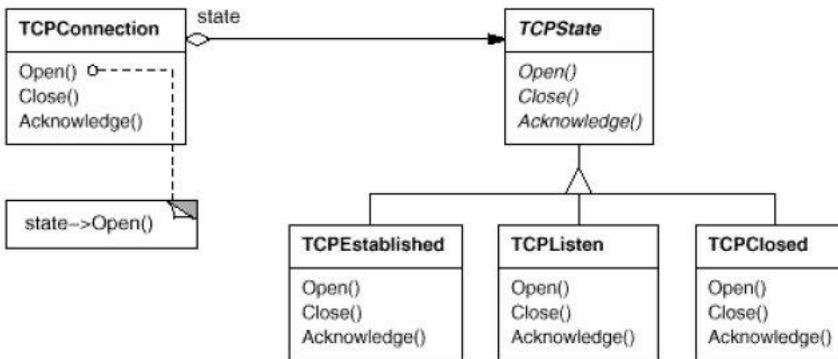
- Amaç:
  - Bir nesnenin iç durumu değiştiğinde davranışının da değişimini sağlamak.
  - Bu değişiklik büyük ölçekli olmalıdır.
    - Küçük Ölçekli: Bir üye alanın değerinin değişmesi yüzünden bir metodun hesapladığı sonucunun değişmesi
    - Büyük Ölçekli: Nesnenin önceki çalışma biçimini tamamen değiştirmesi.
- Örnek:
  - Bir ağ kütüphanesinin bir parçası olarak, TCP gerçeklemesi yapılıyor.
  - Bir TCP bağlantısı, herhangi bir anda şu üç durumdan birinde olabilir:
    - Established, Listen, Close
  - Bağlantı, o andaki durumuna göre farklı şekilde işler
- Esnek olmayan çözüm: Switch-case ile.
  - Benzerini önceden gördük!

68

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### STATE:

- Önerilen çözüm:
  - Neyin değiştiğini bularak sarmalamak: TCP durumları
  - Parça-bütün ilişkisini kullanmaya teşvik: TCP soketi sınıfı ve soket durumunu simgeleyen üye

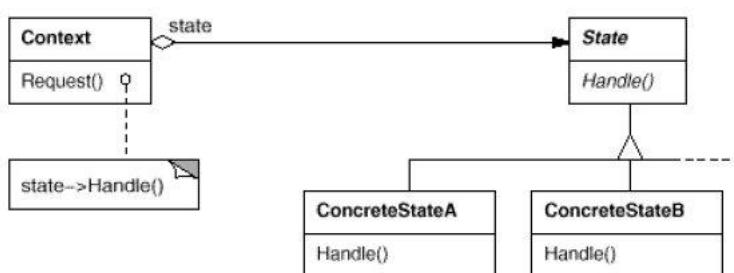


69

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### STATE:

- Kalıp yapısı:



- Kalıp bileşenleri:
  - Context**: Davranışı durum bilgisi ile birlikte değişen varlığı simgeler.
  - State/ConcreteStateX**: Değişen davranışın arayüzü/gerçeklemeleri.
- Gerçekleme ayrıntıları:
  - Durum değişiminden genellikle Context sorumlu olur,
  - Aksi halde?

70

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### STATE:

- Kalıbın kullanılabileceği anlar:
  - Bir nesnenin çalışma şeklinin çalışma anında o nesnenin durumuna göre değişimeceği yerlerde
  - Bir nesnenin durumuna bağlı, karmaşık ve uzun olan çok kısımlı karar verme komutlarının verilmesi gerektiği anlarda.

### STATE ve STRATEGY KALIPLARININ FARKI:

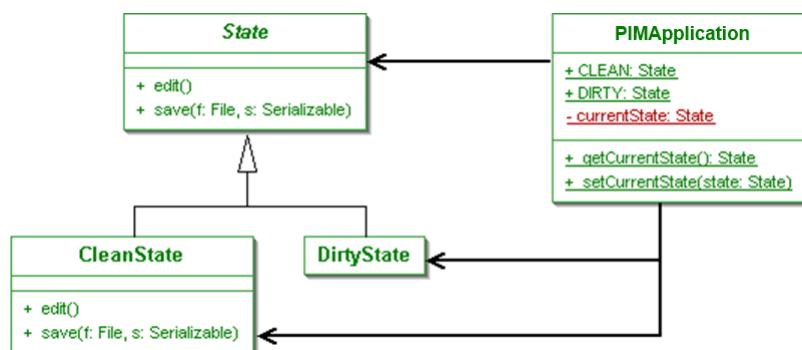
- State:
  - Context'teki durum değişikçe, Context kendi ConcreteState'ini seçer.
  - State geçişleri açıkça yapılır (Context'te verilen komutlarla veya ConcreteState'ler içerisinde)
  - Durum geçişleri daha sıkır.
- Strategy:
  - Strateji geçişleri seyrekir.
  - Strateji nesneleri Context hakkında daha az şey bilir, kendilerine gelen parametrelere göre işlem yapar.

71

## ÖRNEK KALIP GERÇEKLEMELERİ – STATE

### ÇÖZÜLECEK PROBLEM:

- Bir PIM yazılımında kayıtların iki durumu olabilir: Temiz ve Kirli
  - EN: Dirty and Clean
- Çözümün sınıf şeması:



- Kaynak: Applied Java Patterns

72

## ÖRNEK KALIP GERÇEKLEMELERİ – STATE

### ÇÖZÜM:

- Kaynak kodlar

```
package dp.state.example;
import java.io.*;
public abstract class State {
    public void save(File f, Serializable s) throws IOException {
        FileOutputStream fos = new FileOutputStream(f);
        ObjectOutputStream out = new ObjectOutputStream(fos);
        out.writeObject(s);
    }
    public void edit( ) { /*Gerekli işler*/ }
}
```

73

## ÖRNEK KALIP GERÇEKLEMELERİ – STATE

### ÇÖZÜM:

- Kaynak kodlar (devam)

```
package dp.state.example;
import java.io.*;
public class CleanState extends State {
    public void edit( ){
        StateFactory.setCurrentState(PIMApplication.DIRTY);
    }
    public void save(File f, Serializable s) throws IOException {
        StateFactory.setCurrentState(PIMApplication.CLEAN);
        super.save(f, s);
    }
}

package dp.flyweight.example;
public class DirtyState extends State { }
```

74

## ÖRNEK KALIP GERÇEKLEMELERİ – STATE

### ÇÖZÜM:

- Kaynak kodlar (devam)

```
package dp.state.example;
public class PIMApplication {
    public static final State CLEAN = new CleanState();
    public static final State DIRTY = new DirtyState();
    private static State currentState = CLEAN;

    public static State getCurrentState(){
        return currentState;
    }
    public static void setCurrentState(State state){
        currentState = state;
    }
}
```

75

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### TEMPLATE METHOD:

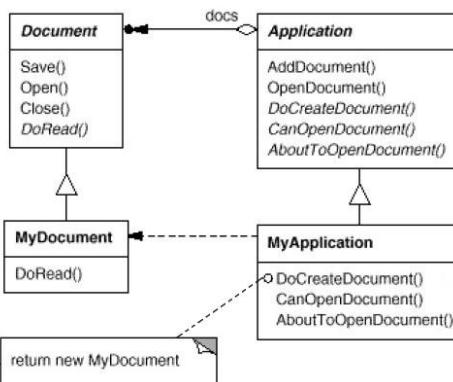
- Amaç: Bir algoritmanın bazı adımlarını belirleyip bazlarının ayrıntılarının belirlenmesini ertelemek.
- Örnek:
  - Farklı belge türleri ile çalışabilecek uygulamalar geliştirmeyi sağlayan bir çerçeve program hazırlanıyor.
  - Kullanıcı çerçevenin sunduğu Belge ve Uygulama sınıflarından kalıtım ile yeni sınıflar türeterek, kendi yazılımını hazırlıyor.
- Sorun:
  - Bir belge açar veya oluştururken her türlü ince ayrıntıyı kullanıcıya atmak istemiyoruz; bazı temel adımların uygun sıra ile yürütülmesini çerçeve programa bırakmak istiyoruz.

76

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

### TEMPLATE METHOD:

- Örnek çözüm:



```

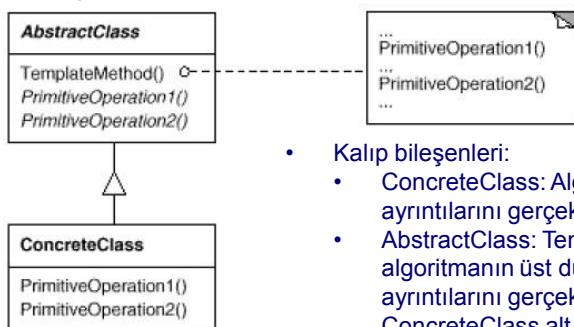
void Application::OpenDocument (const
    char* name) {
    if (!CanOpenDocument(name)) {
        // cannot handle this document
        return;
    }
    Document* doc = DoCreateDocument();
    if (doc) {
        docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
        doc->DoRead();
    }
}
  
```

77

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

### TEMPLATE METHOD:

- Kalıp yapısı:



- Kalıp bileşenleri:

- ConcreteClass: Algoritmanın alt düzey ayrıntılarını gerçekler.
- AbstractClass: TemplateMethod( ) içerisinde algoritmanın üst düzey veya değişmeyecek ayrıntılarını gerçekler, alt düzey ayrıntıları ConcreteClass alt sınıflarına aktarır.

- Gerçekleme ayrıntıları:

- Alt düzey metodlar alt sınıflarda mutlaka tanımlanmalıdır (abstract (Java) veya 'pure' virtual (C++) olmalıdır).
- Üst düzey metot (TemplateMethod) alt sınıflar tarafından yeniden tanımlanamamalıdır (final olmalıdır).

78

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### TEMPLATE METHOD:

- Kalıbin uygulanabileceği anlar:
  - Bir algoritmanın değişmez kısımlarını bir kere kodlayıp değiştirebilecek kısımları alt sınıflar üzerinden gerçekletemek istenildiğinde
  - Alt sınıfların davranışlarının denetlenmesi istenildiğinde
  - Tutarlılığı veya kod tekrarını önlemek üzere kardeş sınıfların ortak davranışlarını bir noktada (üst sınıfta) toplamak istenildiğinde

79

## ÖRNEK KALIP GERÇEKLEMELERİ – TEMPLATE METHOD

### ÇÖZÜLECEK PROBLEM:

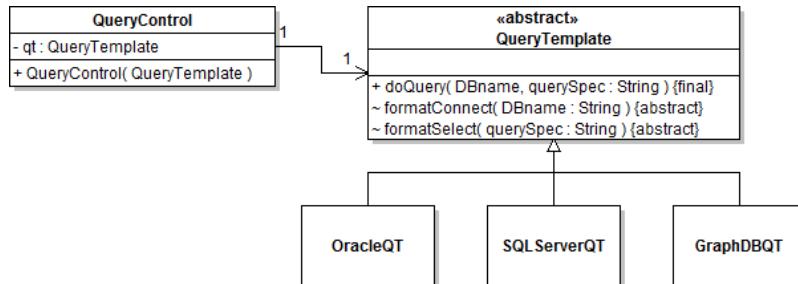
- Bir programda veritabanından kayıtlar çekilecek.
  - Bu işlem SQL ile bir SELECT komutu hazırlanarak verilebilir.
  - Programın farklı VTYS'ler ile çalışabilmesi gerekmektedir.
  - Farklı VTYS'lerde VT ile bağlantı kurma ve SQL komutlarını biçimlendirme faklı olabilir.
    - Özellikle saklı yordam (SP) dilleri VTYS'ler arasında çok farklılık gösterir.
  - Tüm farklılıklara rağmen, kayıt çekme işleminin değişmeyen temel adımları vardır:
    1. Bir bağlantı komutu biçimlendirilir (CONNECT).
    2. VT'na bağlantı komutu gönderilir.
    3. Bir kayıt seçme komutu biçimlendirilir (SELECT).
    4. VT'na seçme komutu gönderilir.
    5. Seçilen veri kümesi geri döner.
  - Ortak temel adımların olması, strateji yerine Template Method kalibinin kullanımını daha doğru bir seçim kılar.
- 
- Kaynak: DP Explained

80

## ÖRNEK KALIP GERÇEKLEMELERİ – TEMPLATE METHOD

### ÖNERİLEN ÇÖZÜM:

- Sınıf şeması:
  - Komut biçimlendirme metotları iç çalışma ile ilgili görülerek protected tanımlanmıştır.
  - Bir sınıf kütüphanesi veya çerçeve program hazırlanmadığına göre, Java için paket görünülüğü kullanılabilir.



81

## ÖRNEK KALIP GERÇEKLEMELERİ – TEMPLATE METHOD

### ÖNERİLEN ÇÖZÜM:

```
package templateMethod.example;
public abstract class QueryTemplate {
    public final void doQuery( String DBname,
                               String querySpec ) {
        String dbCommand;
        dbCommand = formatConnect( DBname );
        //dbCommand komutunu yürüterek bağlantıyi kur.
        dbCommand = formatSelect( querySpec );
        //dbCommand komutunu yürüterek kayıtları oku.
        //Oluşan veri kümesini geri döndür.
    }
    abstract String formatConnect( String DBname );
    abstract String formatSelect( String querySpec );
}
```

82

### ÖRNEK KALIP GERÇEKLEMELERİ – TEMPLATE METHOD

#### ÖNERİLEN ÇÖZÜM:

```
package templateMethod.example;
public class OracleQT extends QueryTemplate {
    String formatConnect(String DBname) {
        //Oracle'a göre özel biçimlendirme komutlarını ver.
        return DBname;
    }
    String formatSelect(String querySpec) {
        //Oracle'a göre özel biçimlendirme komutlarını ver.
        return querySpec;
    }
}
public class SQLServerQT extends QueryTemplate {
    //Komutları bu kez SQL Server'a göre biçimlendir.
    String formatConnect(String DBname) { return DBname; }
    String formatSelect(String querySpec) { return querySpec; }
}
```

83

### ÖRNEK KALIP GERÇEKLEMELERİ – TEMPLATE METHOD

#### ÖNERİLEN ÇÖZÜM:

```
package templateMethod.example;
public class QueryControl {
    private QueryTemplate qt;
    public QueryControl(QueryTemplate qt) {
        this.qt = qt;
    }
    public void aMethodThatNeedsSelect( ) {
        qt.doQuery("myDB", "*");
    }
}
```

84

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### VISITOR:

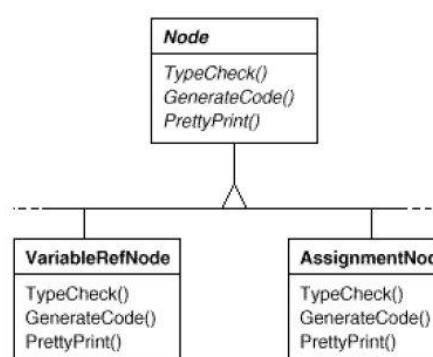
- Amaç:
  - Herhangi bir nesne yapısını oluşturan elemanlar üzerinde yapılacak bazı işlemler tanımlamak.
- Sorun:
  - Nesne yapısı her türden bağıntılar içeren sınıfların örnekleri arasında kurulmuş olabilir.
  - Üzerinde işlem yapılan elemanların sınıf yapılarını değiştirmek istemiyoruz.
- Eleştiri:
  - Madem daha yapılacak işlemler var, eylemler ve sorumluluklar tam olarak belirlenmemiş demektir. Dolayısıyla mevcut sınıf yapılarını değiştirerek farklı bir tasarım yapmak daha doğru olabilir.
- Örnek:
  - Yazılan bir derleyici, kaynak kodu farklı türlerden token'lara ayıriyor.
  - Bu parçalar (Node) farklı tiplerden oluşmaktadır.
  - Derleyici bütün parçalar üzerinde çeşitli işlemleri yapacaktır.
  - Farklı tip parçalarda işlemler farklı yürütülecektir.

85

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### VISITOR:

- Örnek parça yapısı:



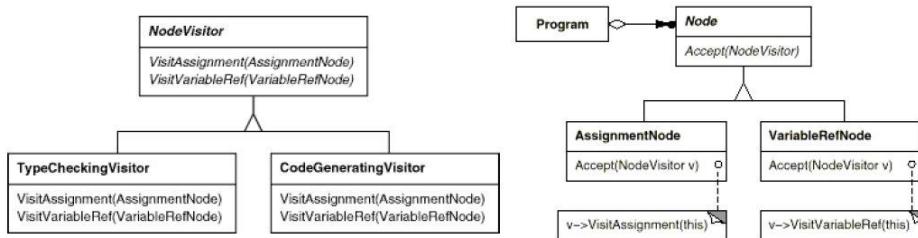
- Bu işlemlerin çeşitli parça yapılarına dağıtılması, bir bakım kabusu oluşturur.
- Her tür düğümde her tür işlemin yapılması da anlamı yoktur.

86

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

### VISITOR:

- Örnek Çözüm: Derleyici parçaları tek tek ziyaret ederek her parça üzerinde sadece gerekli işlemleri yapar.



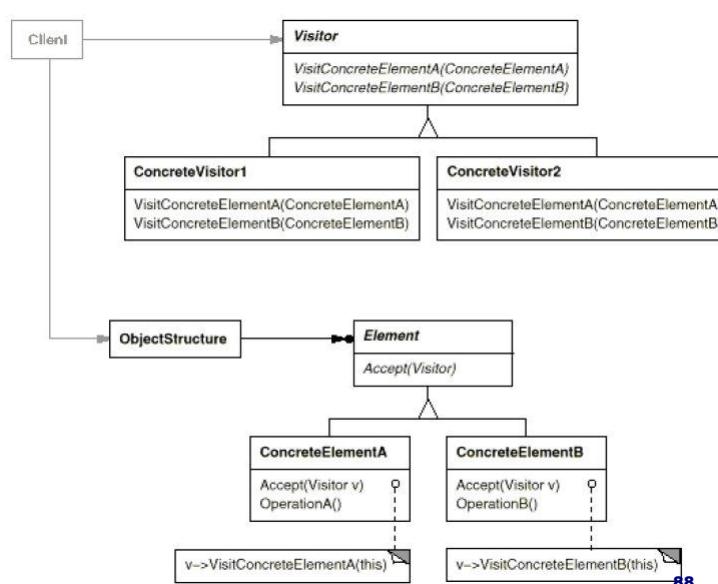
- Düğüm gerçeklemeleri kendi üzerindeki gerekli işlemleri çağıracak ziyaretçi gerçeklemelerini kabul eder (`Accept` metodu ile).
- Bu şekilde parçalarda sadece gerekli işlemler yürütülür.
- Çözümün zayıf yönü: Düğümler ziyaretçinin hangi metodunun bu düğüm türü için olduğunu bilmelidir.

87

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

### VISITOR:

- Kalıp yapısı:



88

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

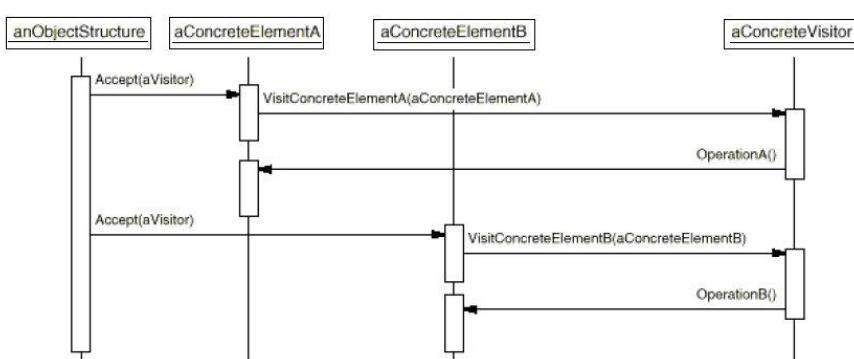
### VISITOR:

- Kalıp bileşenleri:
  - ObjectStructure: Elemanları dolaşılacak yapı
  - Element:
    - Dolaşılacak elemanların ortak arayüzü
    - Accept metodu Element arayüzünde tanımlanır ve ziyaretçiler bu metotla kabul edilir.
  - ConcreteElementX:
    - Dolaşılacak elemanların gerçeklemeleri
    - Ziyaretçilerin hangi metodunun bu tür eleman için olduğu bilgisine sahip olmalıdır.
  - Visitor: Ziyaretçilerin ortak arayüzü.
    - Her tür eleman için olan ziyaret işlemlerinin tamamı burada tanımlanmak zorundadır.
  - ConcreteVisitorX: Ziyaretçi gerçeklemeleri. İş yapılrken elemanlar ayrı ayrı ziyaret edileceğinden, ziyaretler arasında durum bilgisinin saklanması için gerekli kodlama yapılmalıdır.
- Çözümün zayıf yönleri:
  - Yeni ConcreteElement türleri oluşturmak zordur: Her yeni tip için Visitor arayüzüne ve gerçeklemelerine yeni bir metot eklemek gerekecektir. **89**

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### VISITOR:

- Kalıp etkileşimleri:



- Kalının kullanabileceği anlar:

- Değişen arayzlere sahip nesnelerden oluşan bir yapıda, nesneler üzerinde farklı işlemler yapılması gerekiği anlarda VE
- Bu nesne yapısındaki türlerin değişmediği/ender değiştiği koşullarda. **90**

## ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

### ÇÖZÜLECEK PROBLEM – Çözüm 1 :

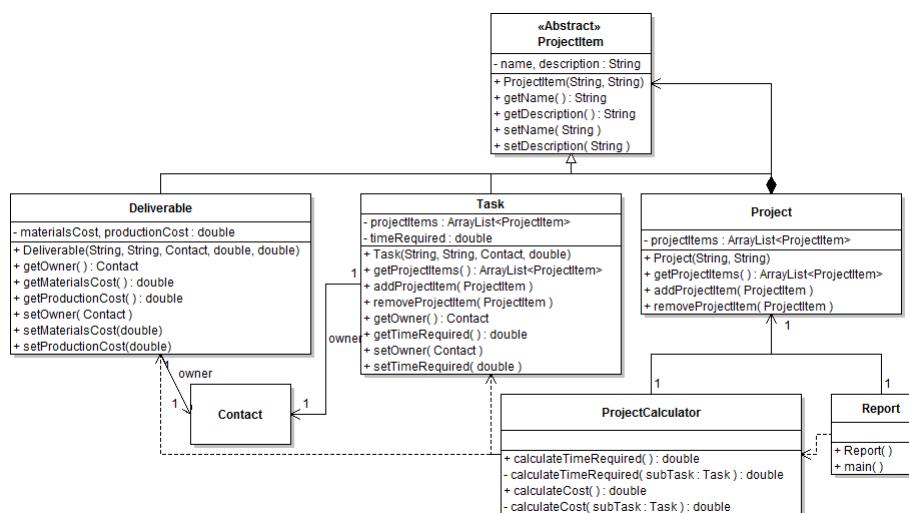
- Bir proje yönetim yazılımı hazırlıyoruz.
- Aşağıdaki temel varlıklarını modelimize alıyoruz:
  - Project: Proje hiyerarşisinin kökü
  - Task: Projede tanımlı bir görev
  - DependentTask: Tamamlanabilmesi için başka görevlere bağlı bir görev
  - Deliverable: Projenin ara çıktıları veya sonucu olarak üretilebilecek herhangi bir kod, belge, ürün, vb.

- Kaynak: Applied Java Patterns

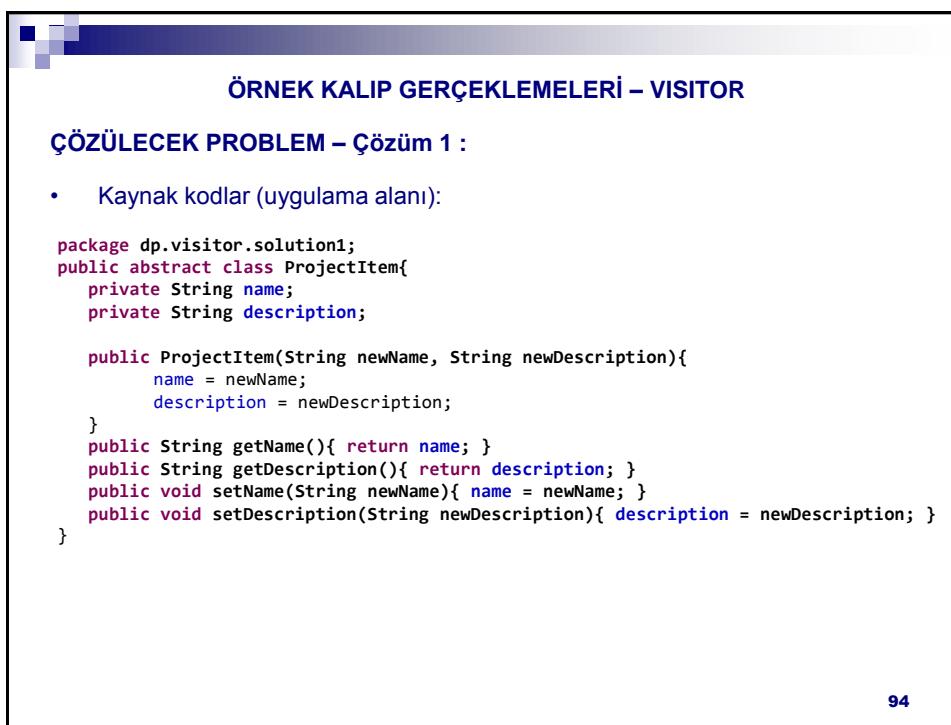
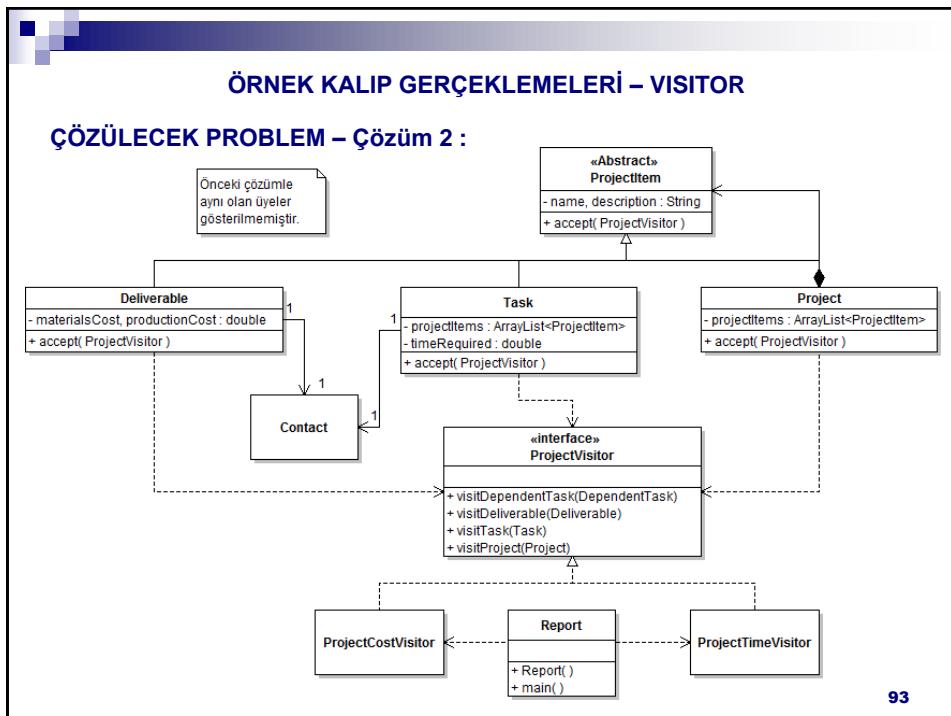
91

## ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

### ÇÖZÜLECEK PROBLEM – Çözüm 1 :



92



## ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

### ÇÖZÜLECEK PROBLEM – Çözüm 1 :

- Kaynak kodlar (uygulama alanı):

```
package dp.visitor.solution1;
import java.util.ArrayList;
public class Project extends ProjectItem{
    private ArrayList<ProjectItem> projectItems = new ArrayList<>();

    public Project(String newName, String newDescription){
        super(newName, newDescription);
    }

    public ArrayList<ProjectItem> getProjectItems(){ return projectItems; }

    public void addProjectItem(ProjectItem element){
        if (!projectItems.contains(element)){
            projectItems.add(element);
        }
    }
    public void removeProjectItem(ProjectItem element){
        projectItems.remove(element);
    }
}
```

95

## ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

```
package dp.visitor.solution1;
import java.util.ArrayList;
public class Task extends ProjectItem{
    private ArrayList<ProjectItem> projectItems = new ArrayList<>();
    private Contact owner;
    private double timeRequired;

    public Task(String newName, String newDesc, Contact newOwner, double newTimeRequired){
        super(newName, newDesc);
        owner = newOwner; timeRequired = newTimeRequired;
    }
    public ArrayList<ProjectItem> getProjectItems(){ return projectItems; }
    public Contact getOwner(){ return owner; }
    public double getTimeRequired(){ return timeRequired; }
    public void setOwner(Contact newOwner){ owner = newOwner; }
    public void setTimeRequired(double newTimeRequired){ timeRequired = newTimeRequired; }

    public void addProjectItem(ProjectItem element){
        if (!projectItems.contains(element)){
            projectItems.add(element);
        }
    }
    public void removeProjectItem(ProjectItem element){
        projectItems.remove(element);
    }
}
```

96

### ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

```
package dp.visitor.solution1;

public class Deliverable extends ProjectItem{
    private Contact owner;
    private double materialsCost;
    private double productionCost;

    public Deliverable(String newName, String newDescription,
                       Contact newOwner, double newMaterialsCost, double newProductionCost){
        super(newName, newDescription);
        owner = newOwner;
        materialsCost = newMaterialsCost;
        productionCost = newProductionCost;
    }

    public Contact getOwner(){ return owner; }
    public double getMaterialsCost(){ return materialsCost; }
    public double getProductionCost(){ return productionCost; }

    public void setMaterialsCost(double newCost){ materialsCost = newCost; }
    public void setProductionCost(double newCost){ productionCost = newCost; }
    public void setOwner(Contact newOwner){ owner = newOwner; }
}
```

97

### ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

#### Raporlama – Çözüm 1:

- Kaynak kodlar:

```
package dp.visitor.solution1;
public class Report {
    private Project project;
    public Report() {
        project = new Project("Yüksek Lisans", "Lisansüstü eğitim (Yüksek lisans)");
        Task gorev; Deliverable urun;
        gorev = new Task("2014-1 dersleri", "2014-2015 Güz yarıyılı (4 ders)", null, 15);
        urun = new Deliverable("BLM 5128", "Yazılım Kalitesi", null, 45, 30);
        gorev.addProjectItem(urun);
        //3 ders daha ekle
        project.addProjectItem(gorev);
    }
    public static void main(String[] args) {
        Report rep = new Report();
        ProjectCalculator calc = new ProjectCalculator(rep.project);
        System.out.println("Proje adı ve açıklaması: " + rep.project.getName() + ":" +
                           rep.project.getDescription());
        System.out.println("Proje süresi: " + calc.calculateTimeRequired() + " hafta.");
        System.out.println("Proje maliyeti: " + calc.calculateCost() + " saat çalışma.");
    }
}
```

98

## ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

### Raporlama – Çözüm 1:

- Kaynak kodlar (devam):

```
package dp.visitor.solution1;

public class ProjectCalculator {
    private Project project;
    public ProjectCalculator(Project project) { this.project = project; }
    public double calculateTimeRequired() {
        double time = 0.0;
        for( ProjectItem item : project.getProjectItems() )
            if( item instanceof Task )
                time += calculateTimeRequired((Task)item);
        return time;
    }
    private double calculateTimeRequired( Task subTask ) {
        double time = subTask.getTimeRequired();
        for( ProjectItem item : subTask.getProjectItems() )
            if( item instanceof Task )
                time += calculateTimeRequired((Task)item);
        return time;
    }
}
```

99

## ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

### Raporlama – Çözüm 1:

- Kaynak kodlar (devam):

```
public double calculateCost( ) {
    double cost = 0.0;
    for( ProjectItem item : project.getProjectItems() ) {
        if( item instanceof Deliverable )
            cost += ((Deliverable)item).getMaterialsCost()+
                    ((Deliverable)item).getProductionCost();
        if( item instanceof Task )
            cost += calculateCost((Task)item);
    }
    return cost;
}
private double calculateCost( Task subTask ) {
    double cost = 0.0;
    for( ProjectItem item : subTask.getProjectItems() ) {
        if( item instanceof Deliverable )
            cost += ((Deliverable)item).getMaterialsCost()+
                    ((Deliverable)item).getProductionCost();
        if( item instanceof Task )
            cost += calculateCost((Task)item);
    }
    return cost;
}
```

100

## ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

### ÇÖZÜLECEK PROBLEM – Çözüm 2 :

- Kaynak kodlar (sadece değişen kısımlar):

```
package dp.visitor.solution2;
public abstract class ProjectItem{
    abstract public void accept(ProjectVisitor v);
}

package dp.visitor.solution2;
public class Project extends ProjectItem{

    public void accept(ProjectVisitor v){ v.visitProject(this); }

}

package dp.visitor.solution2;
public class Task extends ProjectItem{

    public void accept(ProjectVisitor v){ v.visitTask(this); }

}

package dp.visitor.solution2;
public class Deliverable extends ProjectItem{

    public void accept(ProjectVisitor v){ v.visitDeliverable(this); }

}
```

101

## ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

### ÇÖZÜLECEK PROBLEM – Çözüm 2 :

- Kaynak kodlar :

```
package dp.visitor.solution2;
public interface ProjectVisitor{
    public void visitDependentTask(DependentTask p);
    public void visitDeliverable(Deliverable p);
    public void visitTask(Task p);
    public void visitProject(Project p);
}
```

102

## ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

### ÇÖZÜLECEK PROBLEM – Çözüm 2 :

- Kaynak kodlar :

```
package dp.visitor.solution2;
public class ProjectCostVisitor implements ProjectVisitor {
    private double cost = 0.0;
    public ProjectCostVisitor(Project project) {
        project.accept(this);
    }
    public void visitDependentTask(DependentTask p) { visitTask(p); }
    public void visitDeliverable(Deliverable p) {
        cost += p.getMaterialsCost() + p.getProductionCost();
    }
    public void visitTask(Task p) {
        for( ProjectItem item : p.getProjectItems() )
            item.accept(this);
    }
    public void visitProject(Project p) {
        for( ProjectItem item : p.getProjectItems() )
            item.accept(this);
    }
    public double getCost() {
        return cost;
    }
}
```

103

## ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

### ÇÖZÜLECEK PROBLEM – Çözüm 2 :

- Kaynak kodlar :

```
package dp.visitor.solution2;
public class ProjectTimeVisitor implements ProjectVisitor {
    private double time = 0.0;
    public ProjectTimeVisitor(Project project) {
        project.accept(this);
    }
    public void visitDependentTask(DependentTask p) {visitTask(p);}
    public void visitDeliverable(Deliverable p) { }
    public void visitTask(Task p) {
        time += p.getTimeRequired();
        for( ProjectItem item : p.getProjectItems() )
            item.accept(this);
    }
    public void visitProject(Project p) {
        for( ProjectItem item : p.getProjectItems() )
            item.accept(this);
    }
    public double getTime() {
        return time;
    }
}
```

104

## ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

### Raporlama – Çözüm 2:

- Kaynak kodlar:

```
package dp.visitor.solution2;
public class Report {
    private Project project;
    public Report() {
        //önceki ile aynı kurucu
    }
    public static void main(String[] args) {
        Report rep = new Report();
        ProjectCostVisitor vc = new ProjectCostVisitor(rep.project);
        System.out.println("Proje adı ve açıklaması: " + rep.project.getName() + ": " +
                           rep.project.getDescription());
        System.out.println("Proje maliyeti: " + vc.getCost() + " saat çalışma.");
        ProjectTimeVisitor vt = new ProjectTimeVisitor(rep.project);
        System.out.println("Proje süresi: " + vt.getTime() + " hafta.");
    }
}
```

105

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### MEDIATOR:

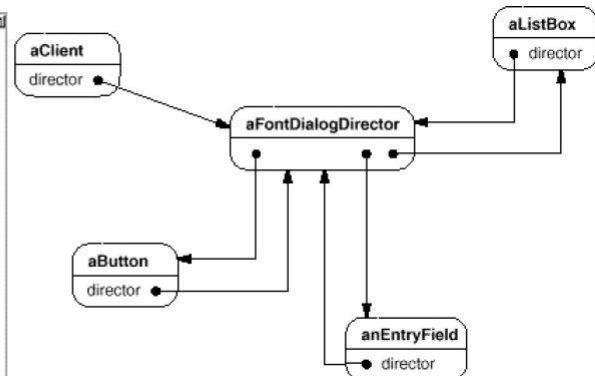
- Amaç:
  - Bir nesneler kümesinin aralarındaki etkileşimlerin sarmalanması.
  - Böylece nesneler birbirleri ile doğrudan değil de dolaylı olarak ilişki kurarlar.
  - Sonuç olarak söz konusu nesnelerin arasındaki ilişkiler bu nesneleri değiştirmeden de değiştirilebilir.
- Örnek:
  - Çeşitli GUI bileşenleri içeren bir font seçme penceresi hazırlanıyor.
  - GUI bileşenleri kullanılarak font ile ilgili yapılan her değişiklik, anında örnek metin alanına yansıtılacaktır.

106

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### MEDIATOR:

- Örnek:

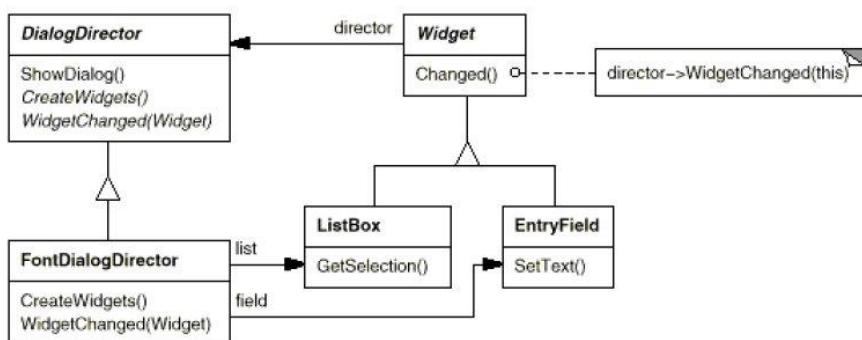


107

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### MEDIATOR:

- Örnek çözüm:
  - GUI nesnelerine ortak bir arayüz hazırlayarak, aralarındaki etkileşimleri genel olarak tanımlayacak bir başka soyut sınıf tasarlanır.
  - Bu soyut sınıfın gerçeklemeler oluşturmak için Template Method kalıbı kullanılır.

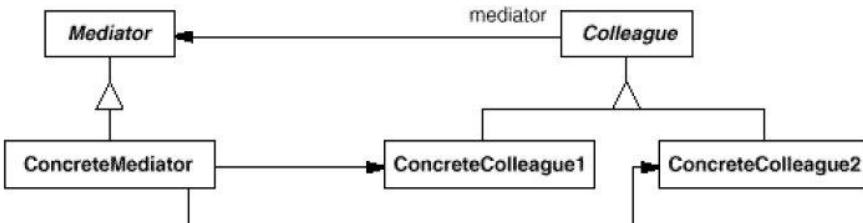


108

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

### MEDIATOR:

- Kalıp yapısı:
  - Colleague gerçeklemeleri kendi aralarında doğrudan değil, bir Mediator gerçeklemesi üzerinden yaparlar.



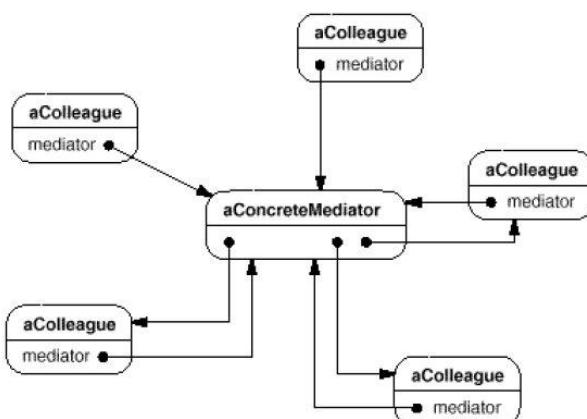
- Kalıp bileşenleri:
  - Mediator: Colleague nesneleri ile etkileşim kurabilecek bir arayüz sağlar.
  - ConcreteMediator: Colleague nesneleri arasında eşgüdüm sağlar ve bunların idamesinden sorumludur (sahiplik ilişkisi).
  - Colleague/ConcreteColleague: Etkileşimde bulunacak nesnelerin arayüzü ve gerçeklemeleri.

109

## DAVRANIŞAL (BEHAVIORAL) KALIPLAR

### MEDIATOR:

- Nesneler arasındaki etkileşimler:



110

## DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

### MEDIATOR:

- Kalıbın zayıf yönleri:
  - Mediator gerçeklemelerinin yapısı aşırı karmaşıklaşır.
  - Bu nedenle Mediator nesnelerinin yeniden kullanımı da zorlaşır.
- Kalıbın kullanılabileceği anlar:
  - Bir grup nesnenin arasında iyi tanımlanmış ancak karmaşık etkileşimlerin bulunduğu anlarda,
  - Birden fazla nesne arasına dağılmış olan bir davranışın tek bir sınıfta toplamak istediği, ancak bu davranışın fazla değişim göstermeyeceği durumlarda.

### ÖRNEK KALIP GERÇEKLEMELERİ – MEDIATOR

#### ÇÖZÜLECEK PROBLEM:

- Bu kalıp için örnek yapılmayacaktır.

111

- Bu yansız ders notlarının düzeni için boş bırakılmıştır.

112

## NESNEYE DAYALI TASARIM VE MODELLEME KISIM 2: REFACTORING

1

### REFACTORING (YENİDEN YAPILANDIRMA)

#### REFACTORING NEDİR?

- Bir kod parçasının ne yaptığını değiştirmeden nasıl yaptığını değiştirmeye refactoring adı verilmiştir.
- Yazılımın dışarıdan görününen davranışını değiştirmeden iç yapısının değiştirilmesi anlamındadır.

#### REFACTORING İLE AMAÇLANAN NEDİR?

- Mevcut kod üzerinde, kodun iç kalite özelliklerini daha iyiye götürecek şekilde değişiklikler yapılması amaçlanır.
  - Kodun anlaşılabilirliğinin artırılması
  - Koda esneklik kazandırılması
  - Yeniden kullanılabilirliğin artırılması
  - Yüksek uyum ve düşük bağımlılığın sağlanması
  - vb.

2

## REFACTORING (YENİDEN YAPILANDIRMA)

### REFACTORING NASIL YAPILIR?

- Çevik yaklaşımla yapılır:
  - Mevcut gereksinimleri karşılayacak ilk tasarım yapılır.
  - İlk tasarımın mükemmel olması ve her bilinmezi öngörmeye çalışması gerekmektedir.
  - Yeterince iyi olan tasarım gerçekleştirmeye başlanır.
  - Önceki tasarım adımlarından birinde yanlış karar vermiş olduğunuzu anlayınca refactoring yapılır.
- Sürekli sinama ile yapılır:
  - Yapılan değişikliklerin yeni hatalar oluşturmadığından emin olmak için, her değişikliğin ardından kodun yeni hali sınağmalıdır.

### NE ZAMAN REFACTORING YAPILIR?

- Kodunuz derlenip toparlanmaya gerek duyduğu zaman yapılır.
  - Kodunuza yeni özellikler eklemekte zorlanmaya başladığınızda
  - Bir hatayı bulmak için zorlanmaya başladığınız, bulduğunuzda da düzeltmek için bir çok ayrı yeri kurcalamak zorunda kaldığınızda
  - vb.
- Kodunuz kötü kokmaya başladığında yapılır!

3

## CODE SMELLS (KÖTÜ KOKAN / KUSURLU KOD)

### CODE SMELL NEDİR?

- Yazılan kodu iyileştirme gereğinin işaretleridir.
- Kod içerisinde karşılaşılan tersliklerdir.
- Kullanılan programlama yaklaşımının kusurlu kullanım örnekleridir.
- Zamanla bir yazılım yamalı bohçaya dönüşebilir.
  - Değişik zamanlarda farklı kişilerce kodun değişik kısımlarında değişiklikler yapılabilir.
  - Bu değişiklikler birlikte düşünülerek planlanabilseydi, daha iyi bir çözüme ulaşılması mümkün olabilirdi.
- Yazılım elimizden kayıp gitmeden, tehlike işaretlerini sezip önlem almak yerinde olacaktır.

### CODE SMELL VE REFACTORING

- Kodda karşılaşılan terslikleri düzeltmek üzere yeniden yapılandırma eylemleri yürütülür.
- Bir yapılandırma eylemi yeni kokular çıkartabilir, bu durumda yeni yapılandırma eylemleri yürütülür.

4

## JUNIT İLE BİRİM SINAMALARI

### BİRİM SINAMALARI (Unit Testing)

- En küçük yazılım bileşeninin sınanmasıdır.
- NYP'de bireysel sınıfların sınanmasıdır.
- Ne zaman tasarılanır?
  - Kodlamadan önce (TDD), kodlama sırasında veya kodlamadan ardından.
  - Bir sınıfın tek başına yürütemediği sorumlulukların sınanması için, bu sınıfın ihtiyaç duyduğu diğer sınıfların yerine geçecek kod gerekebilir.
  - Vekil, sahte, yalancı kod/sınıf, vb. (Stub, dummy, surrogate, proxy)
  - Vekil, sadece ihtiyaç duyulan sınıflar gerçeklenene dek kullanılır.

### JUnit HAKKINDA

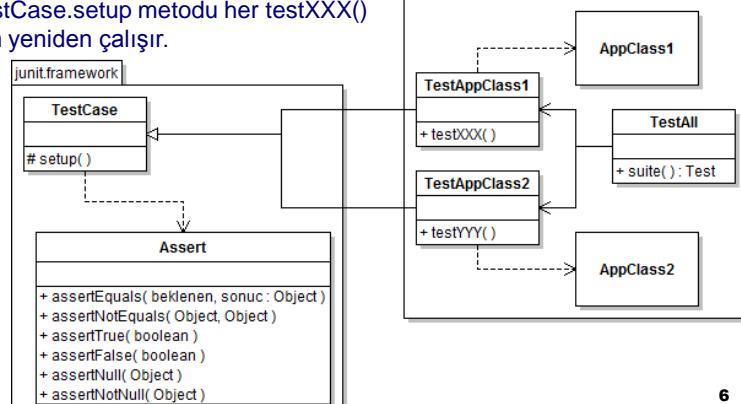
- Java ile yazılmış kodun birim sınavaları için bir çerçeve programdır (framework)
- Diğer diller için de sürümleri bulunmaktadır.
  - Ör. C# için csUnit
- IDE desteği:
  - Eclipse: IDE ile hazır geliyor (build path → add libraries)
  - NetBeans: IDE ile hazır geliyor

5

## JUNIT İLE BİRİM SINAMALARI

### jUnit Sürüm 3.X ile Test Case Hazırlamak

- Her sınıfın birim testi için ayrı sınıfta ayrı test case'ler hazırlamak, ardından test case'leri bir test suite altında toplamak tercih edilmiştir.
  - TestAll sınıfını yazmak zorunlu değildir, TestAppClass sınıfları bireysel olarak da çalıştırılabilir.
  - TestAll.suite() metodu statiktir.
  - TestCase.setup metodu her testXXX() için yeniden çalışır.



6

## JUNIT İLE BİRİM SINAMALARI

### jUnit Sürüm 4.X Gelişmeleri

- Geriye doğru uyumluluk korunmakla birlikte, jUnit 4 sürümü ile annotation desteği gelmiştir (Büyük/küçük harf duyarlılığı vardır).
  - Artık sinama sınıflarının TestCase sınıfından kalıtmala türetilmesi gerekmektedir ancak şu eklemeler yapılmalıdır:

```
import static org.junit.Assert.*;  
import org.junit.*;
```
  - Artık sinama metodlarının adlarını test kelimesi ile başlatmak gereklidir, ilgili metodların başına @Test annotation koymak yeterlidir.
    - setup adlı metodun yerine ise @Before annotation gelmiştir.

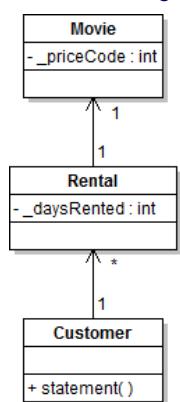
```
@Before  
public void setUp() { /*Preparations*/ }  
  
@Test  
public void testSomething() { /*Do test*/ }
```
- Exception tanımlanan yazılımlarda atılması gereken exception'ların gerçekten ortaya çıkıp çıkmadığının sınanması da mümkün olmuştur.

```
@Test(expected=SomeException.class)  
public void testTheException() throws SomeException {  
    doSomethingThatCreatesTheException();  
}
```

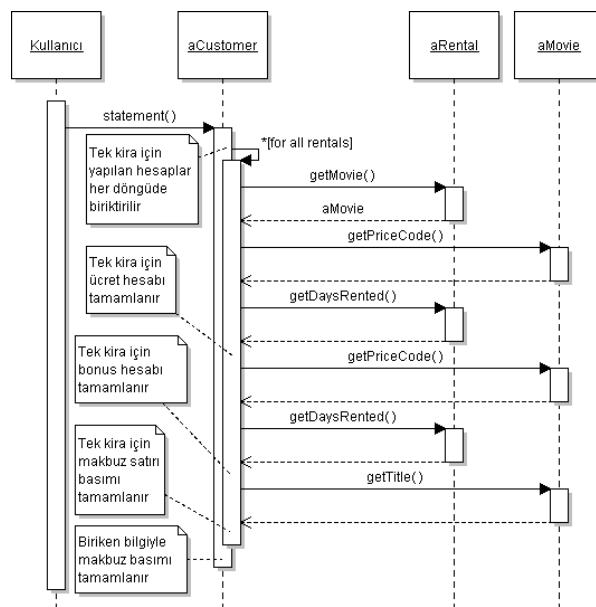
7

## JUNIT İLE BİRİM SINAMALARI

- Basit bir video kiralama örneği:



- Örneğin kaynak kodu:  
refactoring/fowler00



## JUNIT İLE BİRİM SINAMALARI

- Test cases:
  - kaynak kodu: testing/fowler00

```
package fowler_00;
import junit.framework.TestCase;
public class TestCustomer extends TestCase {
    private Customer yunus;
    private Movie matrix, monster, surrogate, terminator;
    protected void setUp() {
        yunus = new Customer("Yunus Emre Selçuk");
        matrix = new Movie("The Matrix",Movie.REGULAR);
        monster = new Movie("Monsters, Inc.",Movie.CHILDRENS);
        surrogate = new Movie("Surrogates", Movie.NEW_RELEASE);
        terminator = new Movie("Terminator Salvation",Movie.NEW_RELEASE);
    }
    public void testGetName() {
        String sonuc = yunus.getName();
        assertEquals("Yunus Emre Selçuk",sonuc);
    }
    public void testStatementWhenEmpty() {
        String sonuc = yunus.statement();
        String beklenen = "Rental Record for Yunus Emre Selçuk\n";
        beklenen += "Amount owed is 0.0\n";
        beklenen += "You earned 0 frequent renter points";
        assertEquals(beklenen, sonuc);
    }
}
```

9

## JUNIT İLE BİRİM SINAMALARI

- Test cases (devam):

```
public void testStatementWithMoviesLongRent() {
    yunus.addRental( new Rental(matrix, 3) );
    yunus.addRental( new Rental(monster, 4) );
    yunus.addRental( new Rental(surrogate, 2) );
    String sonuc = yunus.statement();
    String beklenen = "Rental Record for Yunus Emre Selçuk\n";
    beklenen += "\tThe Matrix\t3.5\n";
    beklenen += "\tMonsters, Inc.\t3.0\n";
    beklenen += "\tSurrogates\t6.0\n";
    beklenen += "Amount owed is 12.5\n";
    beklenen += "You earned 4 frequent renter points";
    assertEquals(beklenen, sonuc);
}
public void testStatementWithMoviesShortRent() {
    yunus.addRental( new Rental(matrix, 2) );
    yunus.addRental( new Rental(monster, 3) );
    yunus.addRental( new Rental(surrogate, 1) );
    String sonuc = yunus.statement();
    String beklenen = "Rental Record for Yunus Emre Selçuk\n";
    beklenen += "\tThe Matrix\t2.0\n";
    beklenen += "\tMonsters, Inc.\t1.5\n";
    beklenen += "\tSurrogates\t3.0\n";
    beklenen += "Amount owed is 6.5\n";
    beklenen += "You earned 3 frequent renter points";
    assertEquals(beklenen, sonuc);
}
```

10

## JUNIT İLE BİRİM SINAMALARI

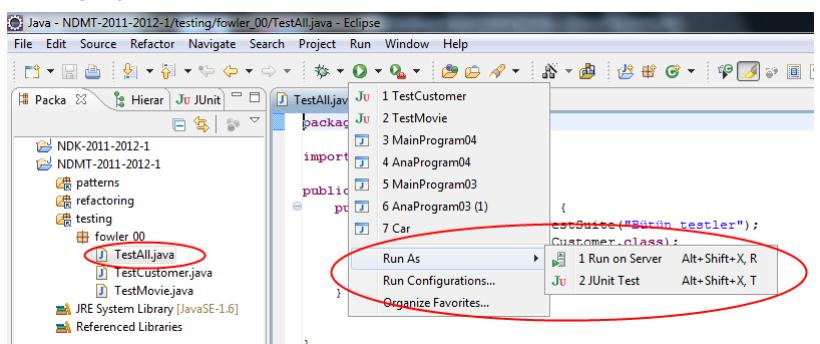
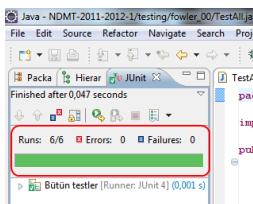
- Test cases (devam):

```
public void testNewReleaseRentalBonus() {  
    yunus.addRental( new Rental(surrogate, 2) );  
    yunus.addRental( new Rental(terminator, 1) );  
    String sonuc = yunus.statement();  
    String beklenen = "Rental Record for Yunus Emre Selçuk\n";  
    beklenen += "\tSurrogates\t6.0\n";  
    beklenen += "\tTerminator Salvation\t3.0\n";  
    beklenen += "Amount owed is 9.0\n";  
    beklenen += "You earned 3 frequent renter points";  
    assertEquals(beklenen, sonuc);  
}
```
- Test suite:

```
package fowler_00;  
import junit.framework.*;  
public class TestAll {  
    public static Test suite() {  
        TestSuite suite = new TestSuite("Bütün testler");  
        suite.addTestSuite(TestCustomer.class);  
        suite.addTestSuite(TestMovie.class);  
        return suite;  
    }  
}
```

11

## JUNIT İLE BİRİM SINAMALARI

- Testi çalışıralım ...  

- ... ve sonuç:  


12

## CODE SMELLS – DUPLICATED CODE

### YİNELENEN KOD

- Aynı kodun veya çok benzer kod yapısının birden fazla yerde tekrarlanması durumudur.
- Bu kodun işlevselliginde değişiklik yapılması gerekiğinde, kodun tekrarlandığı her yerde aynı işlemlerin tekrarlanması gerekecektir.
- Ortaya çıkan sonuç bir bakım kabusudur.
- Çözüme götürecek düzenlemeler (refactorings):
  - Extract Method: YineLENEN kodu bir metot altında toplamak amacı ile kullanılır.
  - YineLENEN kodu toplamaya yönelik diğer düzenlemelere ileride deGİNilecektir.

### ÜYE ALAN ADLANDIRMA

- Kod örneklerinde alt çizgi ile başlayan değişkenler, nesnenin üye alanlarıdır.
- IDE'lerde ise üye alanlar farklı renk ile vurgulanır.

13

## REFACTORING – EXTRACT METHOD

### METOT ÇIKARTMA

- Bir metot içindeki kodun bir kısmını yeni bir metot olarak belirlemeye dayanır.
- Bu yapılandırma bir çok kod kusuru (smell) düzeltir.
- Oluşturulan yeni metoda, amacını açıkça belirten, isabetli bir ad verilmelidir.
  - Böylece alt düzey ayrıntılarla uğraşmadan, kodun anlaşılabilirliği artar.
- Örnek:

```
/** ÖNCE: ***/
void printOwing(double
                amount) {
    printBanner();
    //print details
    System.out.println
        ("name:" + _name);
    System.out.println
        ("amount" + amount);
}
```

```
/** SONRA: ***/
void printOwing(double
                amount) {
    printBanner();
    printDetails(amount);
}
void printDetails (double
                  amount) {
    System.out.println
        ("name:" + _name);
    System.out.println
        ("amount" + amount);
}
```

14

## CODE SMELLS – LONG METHOD

### AŞIRI UZUN KOD

- Bir metodun uzunluğu arttıkça:
  - Anlaşılmazı zorlaşır,
  - Değişikliklerden etkilenme olasılığı artar,
  - Birden fazla ilgi alanını kapsama olasılığı artar (uyumun düşmesi).
- Çözüme götürecek düzenlemeler (refactorings):
  - Extract Method: Uyumlu (belli bir ortak amaca yönelik) komutlar seçilerek bir metot altında toplanır.
  - Aşırı uzun kodu kısaltmaya yönelik diğer düzenlemelere ileride değinilecektir.

15

## REFACTORING – EXTRACT METHOD

### METOT ÇIKARTMA

```
/** ÖNCE: ***/
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // print banner
    System.out.println("*****");
    System.out.println("* Customer Owes *");
    System.out.println("*****");

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" +
        outstanding);
}
```

```
/** SONRA: ***/
void printOwing() {
    printBanner();
    printDetails(getOutstanding());
}

void printBanner() {
    System.out.println("*****");
    System.out.println("* Customer Owes *");
    System.out.println("*****");
}

double getOutstanding() {
    double outstanding = 0.0;
    for (Order each : _orders)
        outstanding += each.getAmount();
    /* for-each ile döngü kısaldı! */
    return outstanding;
}

void printDetails (double outstanding) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" +
        outstanding);
}
```

16

## REFACTORING

### TASARIM İLKELERİ

- Büyük sınıflar ve az sayıda nesne yerine daha çok sayıda küçük sınıflar ve nesneler kullanılması önerilir.
  - Böylece uygulama mantığının programın çeşitli yerlerinde paylaşımı kolaylaşır.
  - Sınıfların anlaşılması kolaylaşır.
  - Yeniden kullanım olasılığı artar.
  - Uyum yükselir.
- Uzun ve çok iş yapan metodlar yerine kısa ve adı ile kendi kendini açıklayan metodların kullanılması önerilir.
  - Öyle ki, kodda yorum satırı bulunmasına hiç gerek kalmasın.
  - Kısa sınıflar için verilen yararlar kısa metodlar için de geçerlidir.
- Büyük bir kodu daha küçük parçalara bölünce, bu kez de yönetilmesi gereken parça sayısı artacaktır.
  - Ancak bu zarar, elde edilen yararlar karşısında daha küçük kalmaktadır.
  - Güncel derleyici ve yorumlayıcıların çoğu, bağlam değiştirme (context switching) ile gelen yükü büyük oranda azaltmıştır.
  - Böylece küçük nesne ve metodların kullanımı yüzünden başarım olumsuz etkilenmez.

17

## CODE SMELLS – DUPLICATED CODE

### YİNELENEN KOD (devam)

- Çözüme götürecek diğer düzenlemeler:
  - Pull Up Method: Yinelenen kod kalitim ağacındaki kardeş veya kuzen sınıflara yayılmışsa, oluşturulan metod ortak üst sınıfa çekilir.
  - Form Template Method: Kod aynı değil ancak benzerse, GoF Template Method tasarım kalıbı kullanılarak kardeş/kuzen sınıflarda düzenleme yapılır.
  - Eğer yinelenen veya benzer kod ilişkisiz sınıflarda yer alıyorsa:
    - Sorumluluklar doğru dağıtılmamış olabilir.
    - Metot bir sınıfa atanır, diğerleri bu sınıfı kullanır.
    - Ya da bu metod, oluşturulacak yeni bir sınıfa atanır ve diğerleri bu sınıfı kullanır.
  - Özette, Extract Method ile oluşturulan yeni metoda en uygun yer aranır.

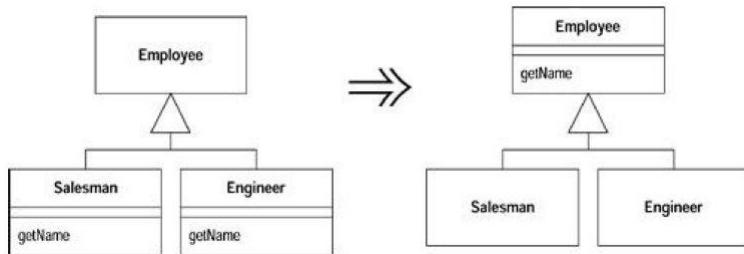
18

## REFACTORING – PULL UP METHOD

### METODU YUKARI ÇEKME

- Kardeş sınıflarda yinelenen metodlar varsa, bunlar üst sınıfı çekilerek bir tek yerde kodlanır.
- Örnek:

/\*\*\* ÖNCE: \*\*\*/      /\*\*\* SONRA: \*\*\*/



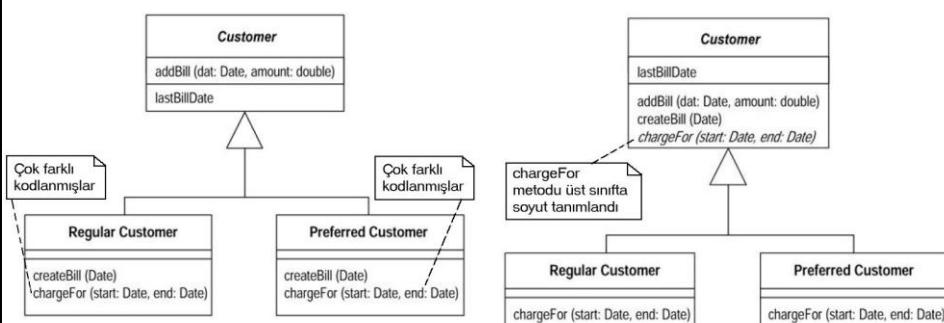
19

## REFACTORING – PULL UP METHOD

### METODU YUKARI ÇEKME

- Özel durumlar:

/\*\*\* ÖNCE: \*\*\*/      /\*\*\* SONRA: \*\*\*/



- Özel durumlar:

- Template Method tasarım kalıbı kullanılabilir.

20

## CODE SMELLS – LONG METHOD

### AŞIRI UZUN KOD

- Çözüme götürecek diğer düzenlemeler:
  - Decompose Conditional: Karar verme (if-switch-case) ve döngülerin yoğun olması çoğu kez kodu uzatır.
  - Replace Temp with Query: Geçici yerel değişkenlerin kullanımı genelde aynı geçici değişkenin kullanımı bahanesi ile kodun uzamasını teşvik eder.
  - Replace Method with Method Object: Üstteki düzenlemeler geçici değişkenleri ve uzun parametre listesini yok edememişse kullanılabilecek daha karmaşık bir düzenleme.

21

## CODE SMELLS – LONG PARAMETER LIST

### UZUN PARAMETRE LİSTESİ

- Bir metodun parametre sayısının fazla olması, hatta bu parametrelerden bazlarının başka metotlarda da birlikte geçmesi durumudur.
  - Demek ki aslında bu parametreler birlikte bir varlığın durum bilgisinin bir kısmını (1) veya tamamını (2) oluşturmaktadır.
- Çözüme götürecek düzenleme
  1. Introduce Parameter Object: Metot parametrelerinin sayısı çok ise azaltmaya yönelik bir düzenlemeyidir.
  2. Preserve Whole Object: Metot parametrelerinin sayısı çok ise azaltmaya yönelik diğer bir düzenlemeyidir.

22

## REFACTORING – DECOMPOSE CONDITIONAL

### KARAR VERME KOMUTUNU PARÇALAMA

- Aşırı karmaşık if komutlarını sadeleştirmek, anlaşılabilirliği artttır:

```
/** ÖNCE: ***/
if (date.before (SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;

/** SONRA: ***/
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge (quantity);
private boolean notSummer(Date date) {
    return date.before (SUMMER_START) ||
    date.after(SUMMER_END);
}
private double summerCharge(int quantity) {
    return quantity * _summerRate;
}
private double winterCharge(int quantity) {
    return quantity * _winterRate + _winterServiceCharge;
}
```

23

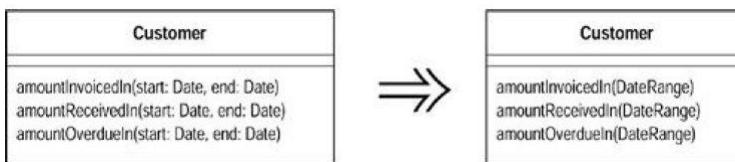
## REFACTORING – INTRODUCE PARAMETER OBJECT

### PARAMETRE NESNESİ OLUŞTURMA

- Bir metot çok fazla parametre alıyorsa, uygun parametreleri üye alan olarak yeni bir sınıfı toplayın ve o türden bir nesneyi parametre olarak kullanın.

/\*\* ÖNCE: \*\*/

/\*\* SONRA: \*\*/



- Bunu sadece parametre listesini kısaltmak amacıyla yapmış olmazsınız:
  - Oluşturulan DateRange sınıfı sadece iki veriyi birleştiren bir kayıt yapısı şeklinde kalmaz,
  - `isWithinRange(Date)` gibi uygun metodlar da içерerek, "decompose conditional" veya "extract method" düzenlemesi de yapılmış olunur.

24

## REFACTORING – PRESERVE WHOLE OBJECT

### TÜM NESNEYİ GÖNDER

- Bir nesnenin çeşitli üyelerini aynı metodun parametreleri olarak göndermek yerine, nesnenin kendisini gönderin.

```
**** ÖNCE: ****/  
int low = daysTempRange().getLow();  
int high = daysTempRange().getHigh();  
withinPlan = plan.withinRange(low, high);
```

```
**** SONRA: ****/  
withinPlan = plan.withinRange(daysTempRange());  
//withinRange metodu da gereki̇gi gibi dėstirilir.
```

- Bu düzenlemenin zayıf yönü: Bağımlılık artabilir
  - Parametre olarak gönderilen nesnenin sınıfı ile metodun sahibi olan sınıf arasında bir bağımlılık oluşur.
  - Bu durum çok sakıncalı olacaksa bu düzenlemeyi kullanmayın.

25

## REFACTORING – REPLACE TEMP WITH QUERY

### GEÇİCİ DEĞİŞKEN YERİNE ERIŞİM METODU KULLANIMI

- Bir ifadeyi bir geçici değişkende saklayarak kullanmak yerine, ifadeden metod(lar) çıkartın.

```
**** ÖNCE: ****/  
double getPrice() {  
    int basePrice = _quantity * _itemPrice;  
    double discountFactor;  
    if (basePrice > 1000) discountFactor = 0.95;  
    else discountFactor = 0.98;  
    return basePrice * discountFactor;  
}
```

```
**** SONRA: ****/  
double getPrice() { return basePrice() * discountFactor(); }  
private int basePrice() { return _quantity * _itemPrice; }  
private double discountFactor() {  
    if (basePrice() > 1000) return 0.95;  
    else return 0.98;  
}
```

26

## REFACTORING – REPLACE TEMP WITH QUERY

### GEÇİCİ DEĞİŞKEN YERİNE ERIŞİM METODU KULLANIMI

- Avantaj: Kopyala-yapıştır kodlamayı daha ortaya çıkmadan azaltmak.
  - Geçici değişkeni kullanan metodunkine benzer bir işlem eklenmek istendiğinde, mevcut metottan yeni metoda kopyala-yapıştır yapılacak.
  - Bu kod parçacığı ile ilgili bir hata bulunursa tüm sınıfı buradan kopyala-yapıştır yapılan yerler aranıp her yerde düzeltme yapmak gerekecek.
  - Halbuki geçici değişkenlerle yapılan işlemler sorgu metodlarında toplanırsa, düzeltmeler de sadece sorgu metodlarında yapılacaktır.
- Tartışma: Bu düzenleme başarımı düşürür.
  - Sadece işlem yapmak yerine bir de fazladan metot çağrıma yükü geliyor.
- Şerh:
  - Modern derleyiciler bu ek yükü ortadan kaldırılmış veya önemsizleştirmiştir.

27

## REFACTORING – REPLACE TEMP WITH QUERY

### GEÇİCİ DEĞİŞKEN YERİNE ERIŞİM METODU KULLANIMI

- Tartışma:
  - Geçici değişken metot içerisinde birden fazla yerde kullanılıyorsa aynı işlem tekrar tekrar yapılacak.
  - Bu da başarımı daha da fazla düşürecek

```
void printReceipt() { /** ÖNCE: ***/
    int totalAmount = 0, thisAmount;
    for( Item item : items ) {
        thisAmount = item.getCharge();
        System.out.println(item.getName()+" "+thisAmount);
        totalAmount += thisAmount;
    }
}
```

```
void printReceipt() { /** SONRA: ***/
    int totalAmount = 0;
    for( Item item : items ) {
        System.out.println(item.getName()+" "+item.getCharge());
        totalAmount += item.getCharge();
    }
}
```

28

## REFACTORING – REPLACE TEMP WITH QUERY

### GEÇİCİ DEĞİŞKEN YERİNE ERİŞİM METODU KULLANIMI

- Tartışma:
  - Az önceki durumda başarıım daha da düşecek.
- Şerh:
  - Başarıım, sistemin tüm özellikleri kodlandıktan sonra bir “profiler” ile değerlendirilmelidir.
  - Belki de sistemin darboğazı bu düzenlemenin sonucunda oluşmamıştır.
    - Hele ki geçici değişkenin sakladığı ifade çok karmaşık değilse.
  - Aksi halde geçici değişkeni tekrar ortaya çıkartırsınız, olur biter.

29

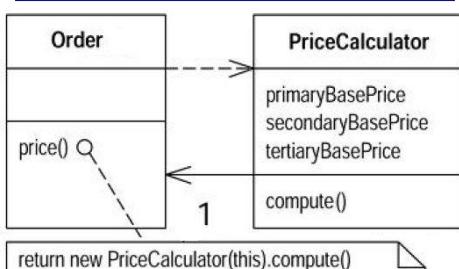
## REFACTORING – REPLACE METHOD WITH METHOD OBJECT

### METOD YERİNE YENİ TÜRDEN NESNE KULLANIMI

- Metot öyle uzun ve geçici yerel değişkenleri öyle arap saçılı gibi kullanıyor ki, “extract method” düzenlemesini kullanamıyorsunuz.
- O zaman yeni bir sınıf oluşturup bütün metodu o sınıfa taşıyın, yerel değişkenleri de o sınıfın üyeleri yapın.
- Ardından bildığınız düzenleme eylemlerini daha rahat yürütübilirsiniz.
- Yeni sınıfın adını verirken, taşınan metodun adına benzer bir ad verin.

```
/** ÖNCE: ***/  
class Order {  
    double price() {  
        double primaryBasePrice;  
        double  
        secondaryBasePrice;  
        double tertiaryBasePrice;  
        // long computation;  
        ...  
    }  
}
```

```
/** SONRA: ***/
```



30

## CODE SMELLS – LARGE CLASS

### AŞIRI BÜYÜK SINIF

- Sınıfın büyük olması, sınıfı ilgisiz sorumluluklar atandığının göstergesidir.
  - Böylece uyum düşecektir.
- Üye alan sayısının fazlalığı, aşırı büyük sınıfın bir göstergesidir.
- Çözüme götürecek düzenlemeler:
  - Extract Class: Birlikte anlam ifade eden üye alanlardan bir veya daha fazla yeni sınıf oluşturulur.
  - Extract Subclass: Bazen oluşturulacak yeni sınıfın, mevcut sınıfın bir alt sınıfı olması daha uygun olabilir. Örneğin mevcut sınıf bazı üyelerini her zaman kullanmıyorsa, bunlar yeni bir alt sınıfa aktarılabilir.
  - Duplicate Observed Data: Eğer iş mantığı ile kullanıcı arayüzü arasında yüksek bağımlılık varsa, GUI sınıfları çok büyüyecektir. Bu durumda veri yeni bir uygulama alanı sınıfına kopyalanır ve aradaki eşgündüm Observer tasarım kalibine göre sağlanır.

31

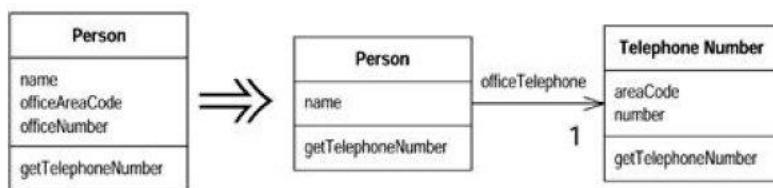
## REFACTORING – EXTRACT CLASS

### SINIF ÇIKARTMA

- Birbiri ile ilgisiz iki iş yapan bir sınıfı, bu işlere göre ikiye ayırmaktır.
- İkinci iş ile ilgili üyelerin seçilerek yeni bir sınıfa taşınmasına dayanır.
- Bu yapılandırma da bir çok kod kusurunu düzeltir.
- Örnek:

/\*\*\* ÖNCE: \*\*\*/

/\*\*\* SONRA: \*\*\*/

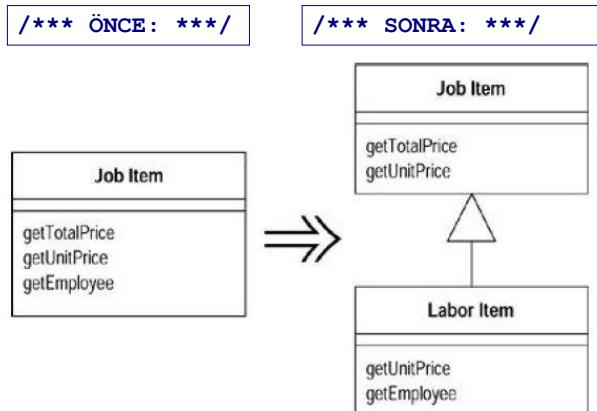


32

## REFACTORING – EXTRACT SUBCLASS

### ALT SINIF ÇIKARTMA

- Bir sınıfın bazı üyeleri her zaman kullanılmıyor, bu üyelerden yeni bir alt sınıf oluşturmak daha doğru olacaktır.
- Örnek:

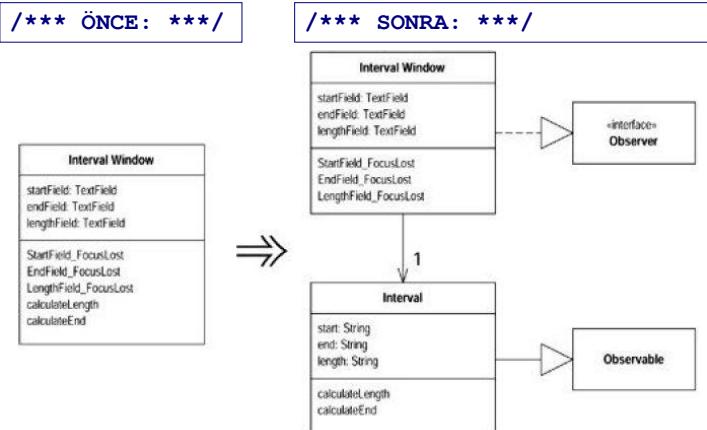


33

## REFACTORING – DUPLICATE OBSERVED DATA

### GÖZLENEN VERİNİN ÇOĞULLANMASI

- Uygulama alanı ile ilgili bir verinin sadece bir GUI sınıfında saklanması doğru değildir.
- Bu veri uygulama alanı/bilgi düzeyi katmanında oluşturulacak yeni bir sınıfa taşınmalıdır ve, GUI nesnesi ile yeni bilgi nesnesi arasında Observer tasarım kalıbı kurulmalıdır.



34

## CODE SMELLS – DIVERGENT CHANGE & SHOTGUN SURGERY

### DEĞİŞİKLİKLERİN FARKLI NOKTALARDA YAPILMASININ GEREKMESİ

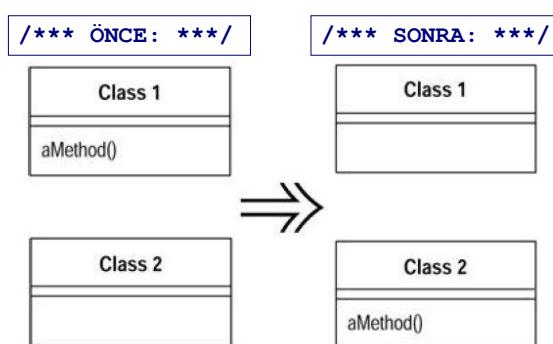
- Gereksinimlerde bir düzenleme olduğunda veya bir hatayı düzeltmek için kodda değişiklikler yapılması gerekecektir.
- Bu değişikliklerin mümkün olduğunda az yerde yapılması tercih edilir.
- Halbuki bir gereksinim değişikliğini karşılamak için gerekli kod değişiklikleri aynı sınıfın farklı metodlarında yapılıyorsa (divergent change) veya birden fazla sınıfıda yapıliyorsa (shotgun surgery), bu işte bir terslik var demektir.
- Divergent change için kötü koku örneği:
  - Yeni bir veritabanı ile çalışmak gerekirse şu sınıfın şu 3 metodu değişir
- Shotgun surgery için kötü koku örneği:
  - Yeni bir tür kredi söz konusu olunca şu 2 sınıfındaki şu 5 metod değişmeli
- Çözüme götürecek düzenlemeler:
  - Extract Class: Söz konusu metodlar ve ilgili üyelerden yeni bir sınıf çıkartılır.
  - Move Method / Move Field: Söz konusu üyeler daha uygun bir mevcut sınıfına taşınır.

35

## REFACTORING – MOVE METHOD

### METOT TAŞINMASI

- Bir metot tanımlandığı sınıfından çok, başka bir sınıflardan çağrılıyorsa; en çok hangi sınıfın çağrılıyorsa o sınıfa taşınmalıdır.
- Bu metot ya eski sınıfından tamamen silinir, ya da eski sınıfındaki hali sadece yeni sınıfındaki halini çağıracak şekilde değiştirilir (delegation).
- Bu düzenleme de pek çok kokuyu giderdiğinde çok sık kullanılır ve bazı düzenlemelerin alt adımıdır (Ör: Sınıf çıkartma/Extract class).
- Örnek:



36

## CODE SMELLS – PRIMITIVE OBSESSION

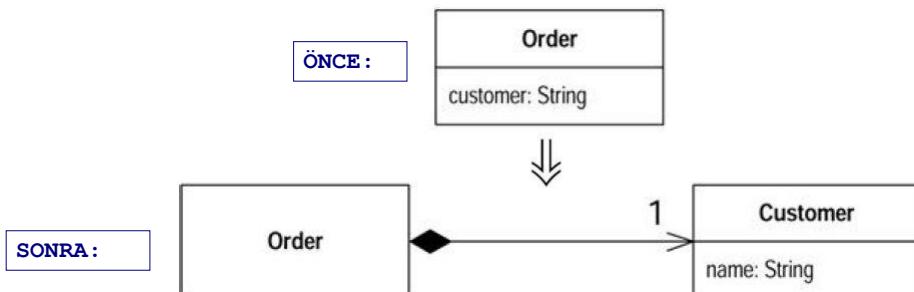
### İLKE TİPLERİN AŞIRI KULLANIMI

- Yapısal programlamadan gelenler, ilkeleri aşırı yoğun olarak kullanır.
- Bu yoğun kullanımın farklı biçimlerini kelimeler ile tarif etmektense, her birine yönelik yeni düzenlemeleri incelemek tercih edilmiştir.
- Çözüme götürecek düzenlemeler:
  - Replace Data Value with Object
  - Replace Type Code with Class
  - Replace Type Code with Subclasses
  - Replace Type Code with State/Strategy
  - Replace Array with Object
  - Extract Class ve Introduce Parameter Object düzenlemeleri de daha nesneye yönelik bir program hazırlamak için kullanılabilir.

37

## REFACTORING – REPLACE DATA VALUE WITH OBJECT

- Bir üye alan, hele ki bu alan ile ilgili çeşitli davranışlar söz konusu ise, bir sınıf ile simgelenmelidir.
- Örnek:

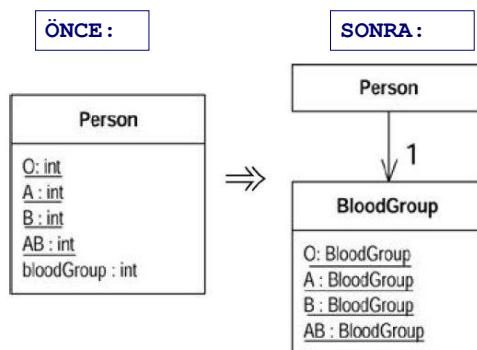


- Programın tümü incelendiğinde, belli bir müşterinin tüm siparişlerinin aranması gibi işlemler yapıldığı görülecektir.
- Bu işlemler çoğaldıkça, Müşteri sınıfı ayrılmadığı sürece, Sipariş sınıfı aşırı büyüyecektir.
- İyisi mi şimdiden bu sınıfları ayıralım.

38

### REFACTORING – REPLACE TYPE CODE WITH CLASS

- Yapısal programlamada sabit değişkenlerin kullanılması alışkanlığı ile ilgilidir.
- Örnek:



- Ya da kısaca enum kullanabilirsiniz.
  - Enum'u bilmeyen yoksa sonraki 2 yansımı atlatabiliriz.

39

### REFACTORING – REPLACE TYPE CODE WITH CLASS

```
/** ÖNCE: ***/
class Person {
    public static final int O = 0;
    public static final int A = 1;
    public static final int B = 2;
    public static final int AB = 3;
    private int _bloodGroup;
    public Person (int bloodGroup) {
        _bloodGroup = bloodGroup;
    }
    public void setBloodGroup(int arg) { _bloodGroup = arg; }
    public int getBloodGroup() {
        return _bloodGroup;
    }
}
```

```
/** SONRA: ***/
class BloodGroup {
    public static final BloodGroup O
        = new BloodGroup(0);
    public static final BloodGroup A
        = new BloodGroup(1);
    public static final BloodGroup B
        = new BloodGroup(2);
    public static final BloodGroup
        AB = new BloodGroup(3);
    private static final
        BloodGroup[] _values = {O, A,
        B, AB};
    private final int _code;
    private BloodGroup (int code ) {
        _code = code; }
    private int getCode() {
        return _code; }
    private static BloodGroup code
        (int arg) {
        return _values[arg]; }
}
```

- Dikkat: BloodGroup sınıfının tüm metotları private.

40

### REFACTORING – REPLACE TYPE CODE WITH CLASS

- Person sınıfı da yeni BloodGroup tipini kullanacak şekilde değiştirilir:

```
/** ÖNCE: ***/
class Person {
    public static final int O = 0;
    public static final int A = 1;
    public static final int B = 2;
    public static final int AB = 3;
    private int _bloodGroup;
    public Person (int bloodGroup)
        { _bloodGroup = bloodGroup; }
    public void setBloodGroup(int arg) { _bloodGroup = arg; }
    public int getBloodGroup() {
        return _bloodGroup;
    }
}
```

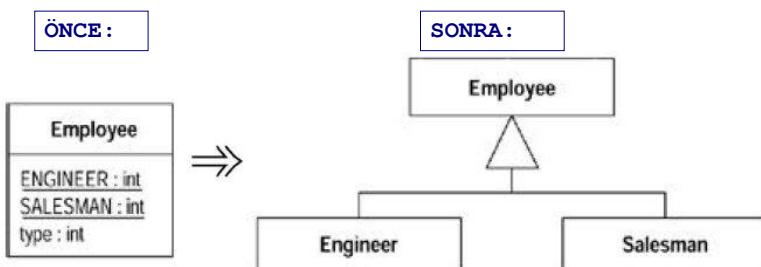
```
/** SONRA: ***/
class Person {
    private BloodGroup _bloodGroup;
    public Person (BloodGroup bg) {
        _bloodGroup = bg; }
    public BloodGroup
        getBloodGroup()
    { return _bloodGroup; }
    public void setBloodGroup (
        BloodGroup bg ) {
        _bloodGroup = bg; }
}
```

- Not: Java'da enum sınıfları da tamamen bu şekilde çalışır ve daha sadedir. Java'da ilkel enum tanımı bile aslında bir enum sınıfı tanımıdır.

41

### REFACTORING – REPLACE TYPE CODE WITH SUBCLASSES

- Bir sınıfın davranışını belirleyen bir ilkel tip kodu yerine kalıtımın kullanımı daha uygun olacaktır.
- Örnek:



42

### REFACTORING – REPLACE TYPE CODE WITH SUBCLASSES

```
/** ÖNCE: */
class Employee {
    private int _type;
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;
    Employee (int type) {
        _type = type;
    }
    public void work( ) {
        switch(_type) {
            case ENGINEER:
                //do something
                break;
            case SALESMAN:
                //do something
                break;
            case MANAGER:
                //do something
                break;
        }
    }
}
```

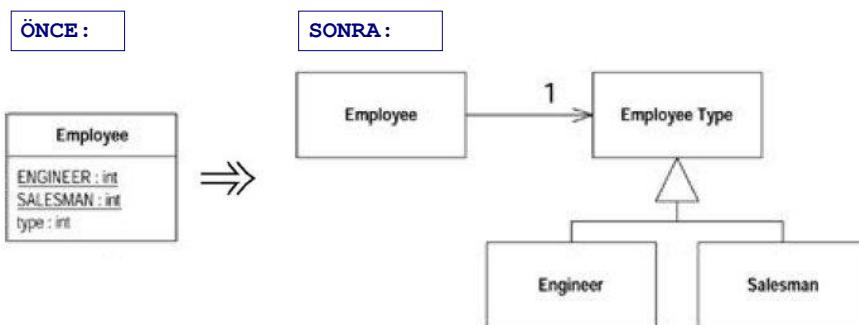
```
/** SONRA: */
abstract class Employee {
    public abstract void work();
}
public class Engineer extends Employee {
    public void work( ) {
        //do something as an engineer
    }
}
public class Salesman...
```

- Not: Employee'nin önceki halinde case default varsa, sonraki halde Employee sınıfı abstract yapılmaz.
- Not: Bu örnek Replace Conditional with Polymorphism düzenlemesine de benzemektedir.

43

### REFACTORING – REPLACE TYPE CODE WITH STATE/STRATEGY

- Önceki örnekte kalıtım kullanımı bir nedenle mümkün değilse, State veya Strategy tasarım kalığı da kullanılabilir.
- Örnek:



44

## REFACTORING – REPLACE ARRAY WITH OBJECT

- Çok boyutlu dizilerde kayıt tutmak yerine, tek bir nesne kullanın.
- Örnek:

```
/** ÖNCE: ***/
String[][] scores = new
    String[3][teamCount];
scores[0][0] = "Liverpool";
scores[1][0] = "15";
scores[2][0] = "67";
```

```
/** SONRA: ***/
Score[teamCount] scores;
scores[0] = new
    Score("Liverpool");
scores[0].setWins("15");
scores[0].setPoints("67");
```

45

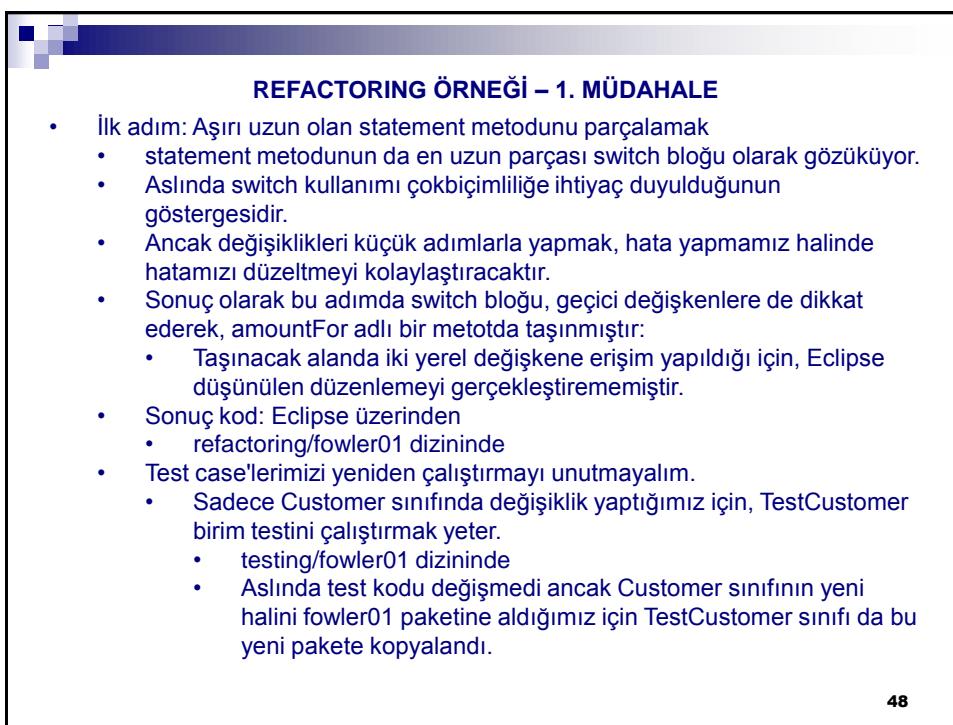
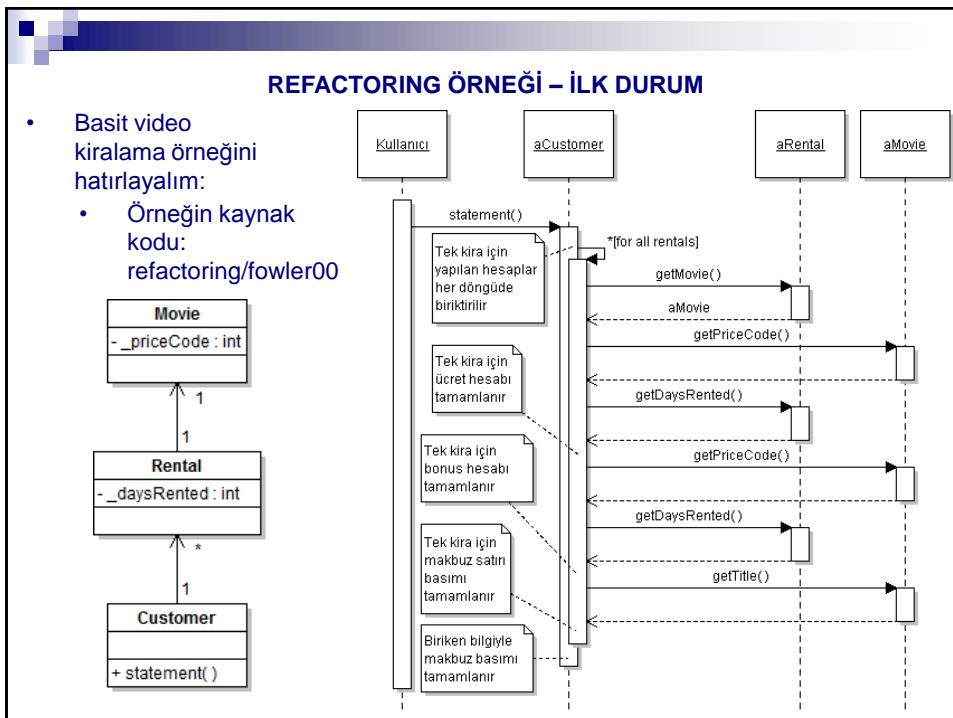
## CODE SMELLS – FEATURE ENVY

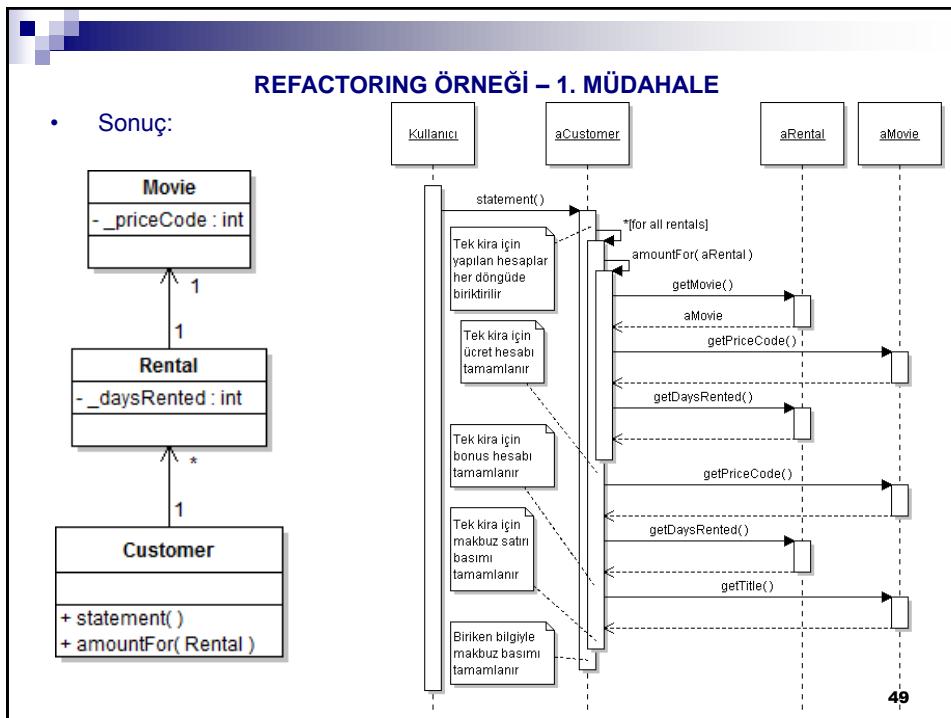
- Bir metot ait olduğu sınıfın üyelerinden çok başka sınıfların üyelerine erişiyorsa, bu duruma 'Özellik kıskançlığı' adı verilir.
  - Sorumlulukların doğru atanmadığının göstergesi olabilir.
  - Elbette bağlaşım ilkesi nedeniyle başka sınıflardan nesnelere erişmek gerekecektir, bu nedenle hiçbir sınıf sadece kendi metodlarını kullanmaz. Öyle olsaydı bu kez de uyum ilkesini olumsuz yönde etkilemiş olurduk.
  - Strategy ve Visitor kalıpları bir özellik kıskançlığı durumu olarak yorumlanmamalıdır.
- Çözüm için olası refactoring eylemleri:
  - Move method
  - Extract method: Eğer kıskançlık metodun sadece bir kısmında ise o kısmı çıkartıp kıskanılan üyelerin bulunduğu sınıfa taşınabilir.

## BİR TASARIMIN ARDİŞİL DÜZENLEMELER İLE DEĞİŞTİRİLMESİ

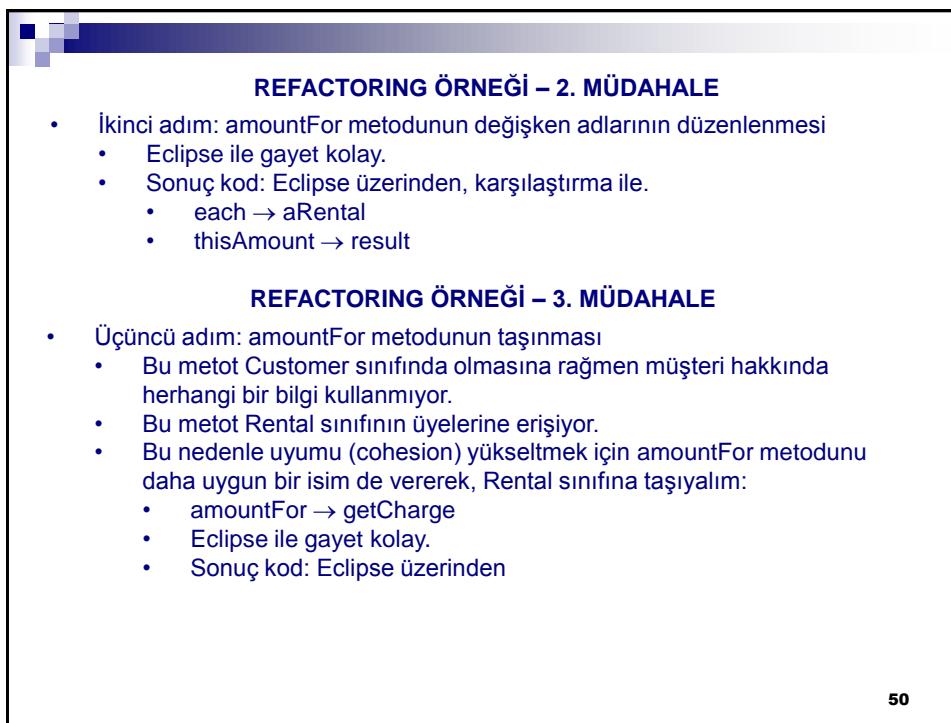
- Şimdiye kadar incelenen düzenlemelerden bazlarının birlikte kullanılarak, yapısal programlama yaklaşımı izleri taşıyan bir tasarımın nasıl değiştirildiğinin bir örneği için Fowler'in Refactoring kitabının 1. bölümü incelenebilir.

46

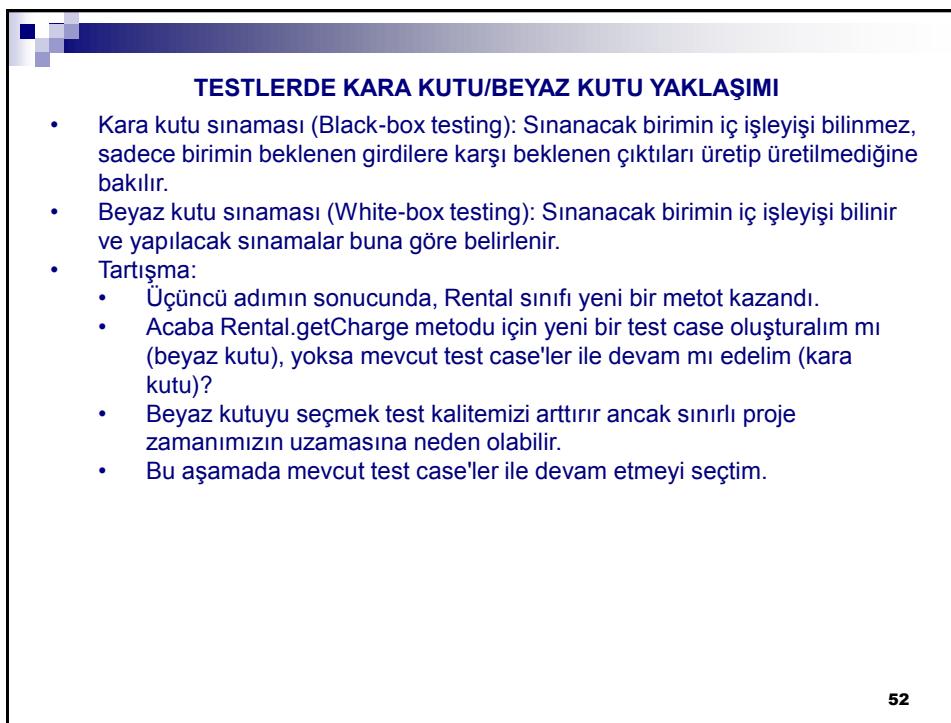
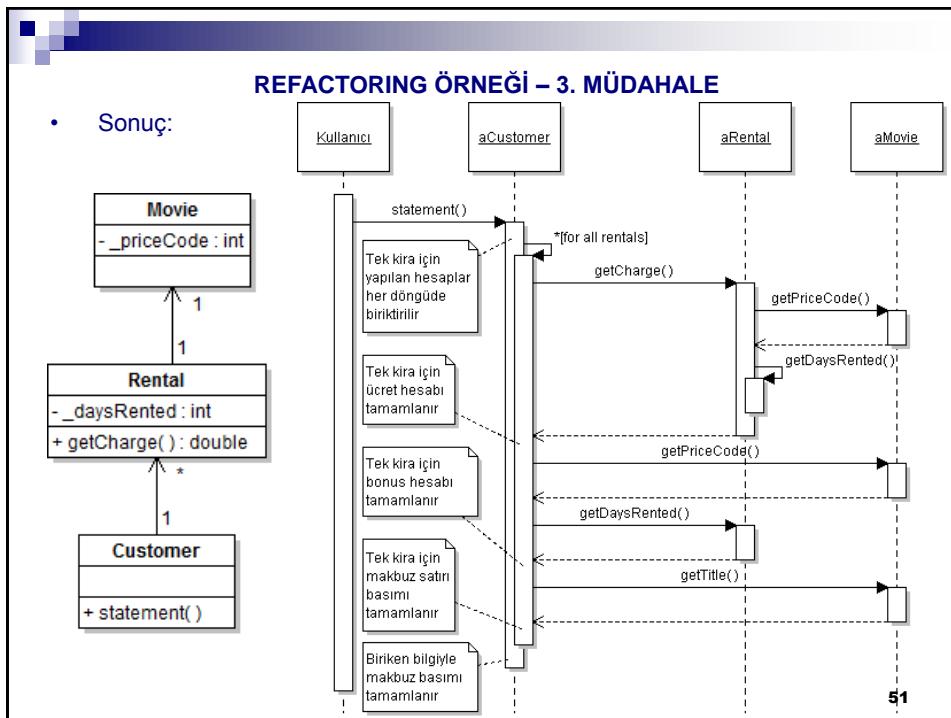




49



50



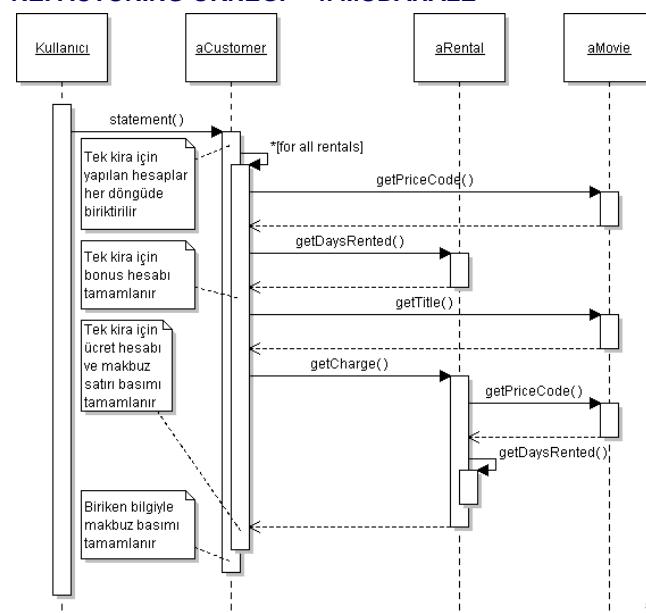
### REFACTORING ÖRNEĞİ – 4. MÜDAHALE

- Dördüncü adım: statement metodundaki yerel thisAmount değişkeni gereksizdir.
  - Bu değişkene getCharge metodunun döndürdüğü değer atanmakta ve değişkenin geçerlilik süresince bu değişkenin değeri değişimmemektedir.
  - Bu nedenle thisAmount yerine getCharge kullanılmıştır.
  - Bedel hesaplama daha ileride gerçekleştiği için etkileşim şeması değişecektir.
  - Sonuç kod: Eclipse üzerinden, karşılaştırma ile.

53

### REFACTORING ÖRNEĞİ – 4. MÜDAHALE

- Sonuç:



54

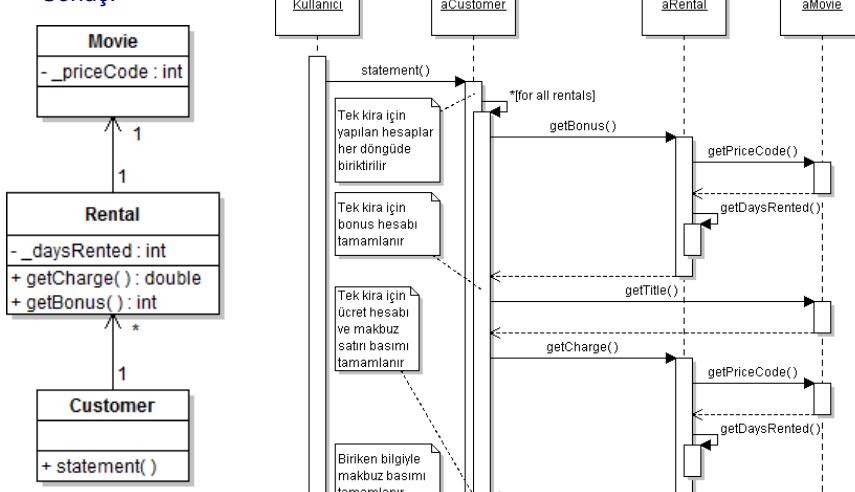
### REFACTORING ÖRNEĞİ – 5. MÜDAHALE

- Beşinci adım: Bonus hesaplaması için metod çıkartma işlemi.
  - Bonus (frequentRenterPoints) kiralama işlemi ile ilgili olduğu için, bu işlemin Rental sınıfına alınması yerinde olacaktır.
  - FrequentRenterPoints çok uzun → Bonus kullanıldı.
  - Sonuç kod: Eclipse üzerinden, karşılaştırma ile.
- UML şemaları hakkında:
  - Amacımız refactoring'i vurgulamak. Bu yüzden her getter/setter metodunu göstermedik.

55

### REFACTORING ÖRNEĞİ – 5. MÜDAHALE

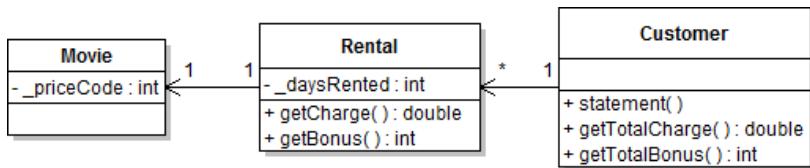
- Sonuç:



56

### REFACTORING ÖRNEĞİ – 6. MÜDAHALE

- Altıncı adım: statement metodundaki diğer yerel değişkenler olan totalAmount ve bonus'un kaldırılması
  - totalAmount → getTotalCharge( ) ve bonus → getTotalBonus( ) dönüştü.
  - Sonuç sınıf şeması:



57

### REFACTORING ÖRNEĞİ – 6. MÜDAHALE

- Sonuç etkileşim şeması: Dosya üstünden.
  - Şema fazla uzamaya başladı, ancak bunun nedeni kod fazlalığından ziyade, eski durumlarda ilkeller ve geçici değişkenler üzerinden yapılan işlemlerin artık metotlar üzerinden yapılmaya başlanmasıdır.
  - Anlamlı metot adları ile açıklama kutularının azalmasına dikkat.
    - Tasarımın anlaşılabilirliği artmıştır.
- Sonuç kod: Eclipse üzerinden, karşılaştırma ile.

58

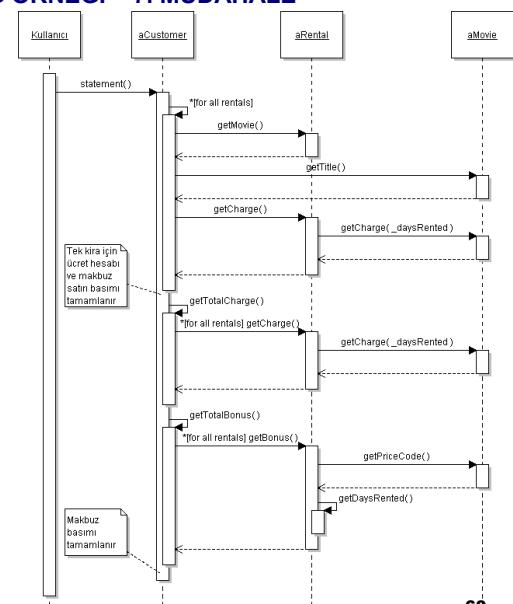
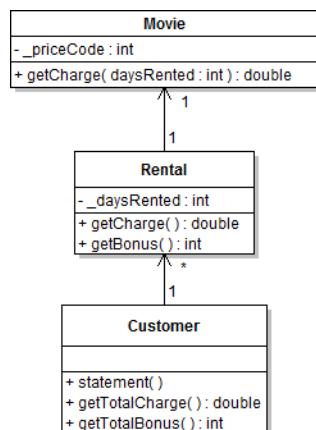
## REFACTORING ÖRNEĞİ – 7. MÜDAHALE

- Yedinci değişiklik: getCharge metodunun bölünmesi
  - getCharge metodu Rental üyelerine erişiyor diye 3. adımda bu sınıfı taşıdı.
  - Ancak bu metot aynı zamanda Movie üyelerine de erişiyor.
  - Üstelik bu erişim switch-case bloğunda kullanılıyor.
  - Bir başka sınıfın üyelerinin durumlarına switch-case bloğunda erişmek akılçılca olmaz:
    - Bu başka sınıfa yeni durumlar eklenince bu durumun işlendiği tüm sınıflarda değişiklik gerekecektir.
- Çözüm: getCharge metodunun bir kısmının Movie sınıfına taşınabilecek şekilde bölünmesi.
  - Öyle ki, her sınıfta sadece o sınıfın üyelerine doğrudan erişilsin.
  - Filmin kaç gün kiralandığı bilgisi Rental sınıfında, film türü ise Movie sınıfında saklanmaktadır.
  - Demek ki, bölme işlemi de bu şekilde yapılmalıdır.

59

## REFACTORING ÖRNEĞİ – 7. MÜDAHALE

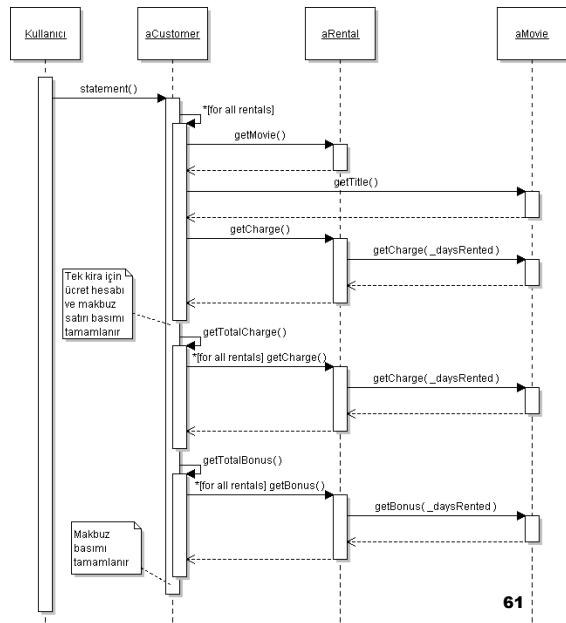
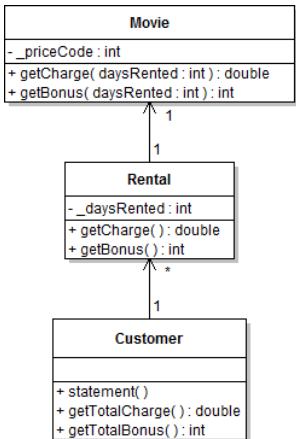
- Sonuç:



60

### REFACTORING ÖRNEĞİ – 8. MÜDAHALE

- Sekizinci değişiklik:  
getBonus metodunun da önceki gibi bölünmesi:

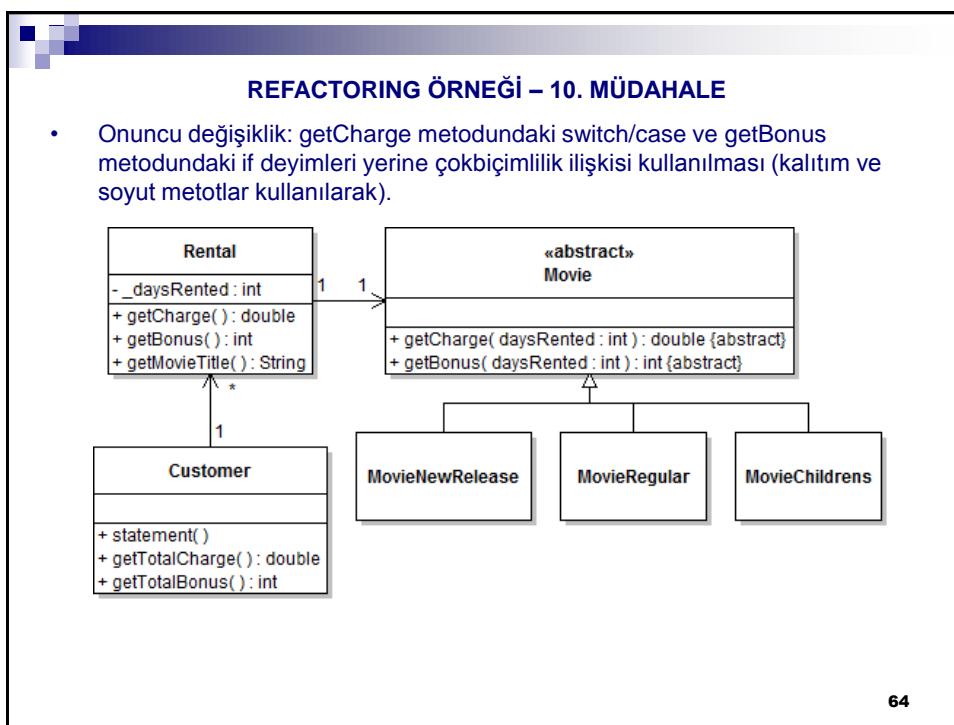
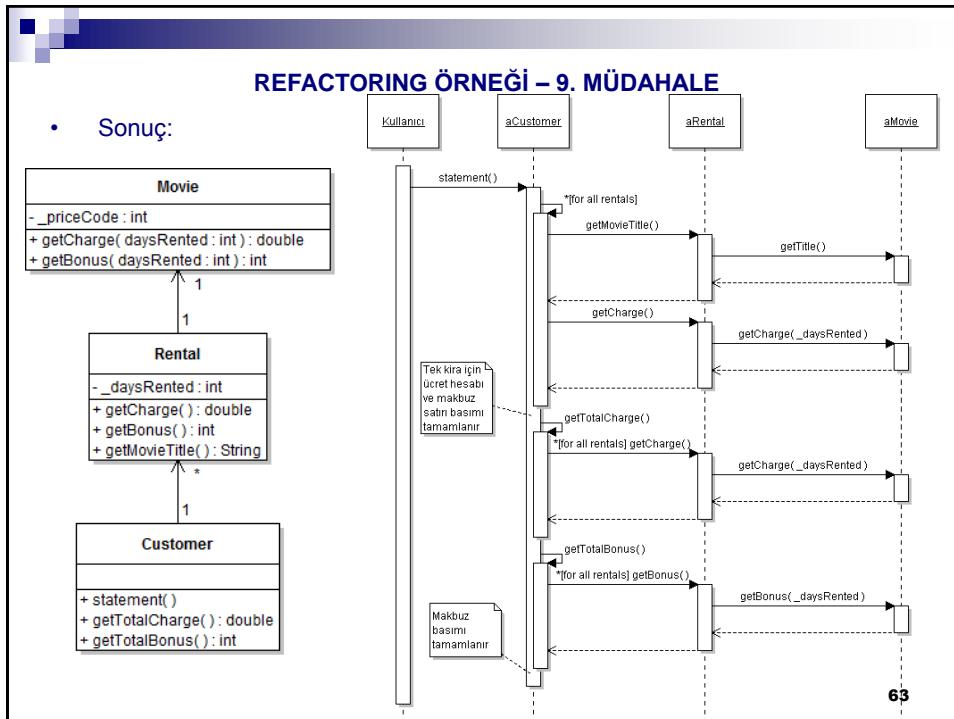


61

### REFACTORING ÖRNEĞİ – 9. MÜDAHALE

- Dokuzuncu değişiklik: Customer ile Movie arasındaki doğrudan bağlaşımın dolaylı bağlaşım haline getirilmesi.
  - 8. adıma kadar yapılan değişiklikler sonucunda, Customer ile Movie arasındaki tek doğrudan ilişki `getTitle` mesajı oldu.
  - Daha düşük bağlaşım sağlamak amacıyla, bu erişim Rental üzerinden sağlanabilir.

62



## ANTI PATTERNS (KARŞIT KALIPLAR)

### ANTIPATTERN NEDİR?

- Karşit kalıp çözüm bir tasarım kalıbı gibi görünür ancak öyle değildir.
- Karşit kalıplar aslında tasarım kalıplarının karşıtları olarak yazılım geliştirme sürecinde tekrar eden bazı yanlışları ifade ederler.
- Karşit kalıpları bilmek, yazılım geliştirme sürecinde karşılaşılabilen ciddi problemleri önceden tahmin edebilmeyi ve tedbir almayı kolaylaştırır.

### KARŞIT KALIP İLE KOD KUSURU FARKI:

- Karşit kalıplar tasarım düzeyine, kod kusurları gerçekleme düzeyine daha yakındır.
- Karşit kalıplar yazılım yaşam döngüsünün diğer evreleri ile de ilişkilidir.

### KARŞIT KALIP TÜRLERİ

- Karşit kalıplar üç ayrı grupta incelenmektedir: (2017-1'de işlemeyecek)
  - Yazılım geliştirme karşıt kalıpları
  - Yazılım mimarisi karşıt kalıpları
  - Yazılım proje yönetimi karşıt kalıpları