NESNEYE DAYALI TASARIM VE MODELLEME
Eylül 2020
Dr.Öğr.Üyesi Yunus Emre SELÇUK

GENEL BİLGİLER

DERS İÇERİĞİ

- Kısım 1: Gang of Four Tasarım Kalıpları
- Kısım 2: Code Smells 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 yansı)
- Kısım 2:
  - Refactoring: Improving the Design of Existing Code, Martin Fowler. Addison-Wesley, 1999

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 önereceğiniz 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

- Küçük sınav: %0, UML sınıf şemalarına ve nesneye yönelime hakimiyetinizi görmek amaçlı, 3. hafta (22 Ekim 2020)
- 1. Ara sınav: %30, Tasarım kalıpları konulu, 8. hafta (26 Kasım 2020)
- 2. Ara sınav: %30, Tasarım kalıpları konulu, 12. hafta (24 Aralık 2020)
- Vize haftalarında değişiklikler olabilir, bölüm web sayfasından ve hocanın sayfasından duyuruları izleyiniz.
- Ara sınav telafi: 14. hafta (7 Ocak 2021)
- 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 ayrılabiliyorsa, 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 kullanılabilirlik
  - İlgi alanlarının ayrılması
  - Esneklik
  - Taşınabilirlik
  - Okunabilirlik ve anlaşılabilirlik
  - Sınanabilirlik
  - 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 tasarıma götüren temel ilkeler:
  - Düşük bağlaşım (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

- Bağlaşım: 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 bağlaşım oranı artar.
- Düşük bağlaşımın 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 bağlaşıma ve yüksek uyuma sahip olması istenir.

8

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

9

TASARIM KALIPLARINA GİRİŞ

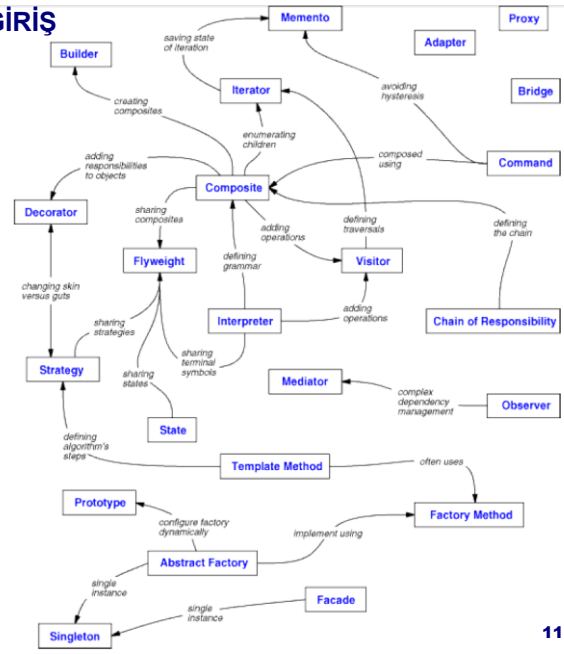
TASARIM KALIBI (DESIGN PATTERN) NEDİR?

- Yazılım tasarımında 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 arttıracak yönde çözülmesi.
- Bir tasarım kalıbının ana bileşenleri:
  - İsim: Kalıbı en güzel şekilde anlatacak bir veya birkaç kelimelik isim.
  - Sorun: Kalıbın çözdüğü 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.

10

TASARIM KALIPLARINA GİRİŞ

- Kalıplar arasında olası ilişkiler:



11

TASARIM KALIPLARINA GİRİŞ

TASARIM KALIBI (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.

12

TASARIM KALIPLARINA GİRİŞ

TASARIM KALIBI (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 dağıtılması ve algoritma seçimine yönelik kalıplardır.

13

TASARIM KALIPLARINA GİRİŞ

TASARIM KALIBI (DESIGN PATTERN) NEDİR?

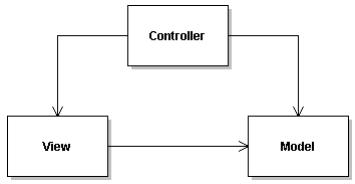
- Ders notlarındaki kalıplar ve anlatım sırası hakkında (devam):
  - GoF kalıplarının bazılarının odağı sınıflar, bazılarının odağı ise nesnelerdir.
    - Sınıfsal kalıplar: Sınıflar ve bunların alt sınıfları arasındaki ilişkilere yönelik kalıplardır. Sınıf yapısı ve kalıtım ilişkisinin doğası gereği derleme zamanında belirlenen ve daha durağan ilişkilerdir.
    - Nesnesel kalıplar: Çeşitli türden nesneler arasındaki ilişkilere yönelik kalıplardır. Kodun çalışması süresince rahatlıkla değiştirilebilen 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.

14

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.

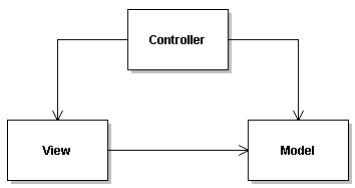
- Benzetim:
  - Bir ressam ve çizdiği insan.
  - Başka?

15

TASARIM KALIPLARINA GİRİŞ

MVC TASARIM KALIBI

- Yararlar:
  - Gösterim, kontrol ve veriyi birbirlerinden ayırarak;
    - Bunları çalışma anında değiştirebilmek.
    - İlgi alanlarının ayrılmasını sağlamak.
    - Yeniden kullanımı arttırmak.



- Çeşitlemeler:
  - Model'in geçirdiği değişiklikleri View'a bildirmesi
  - Birden fazla view:
    - İç içe (nested).
    - Ayrı/tek görevcikte (thread)
    - Örtüşen/ayrık
  - Birden fazla kontrolcü:
    - Farklı denetim biçimleri (klavye, fare, ses, …)

16

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

17

YARATIMSAL (CREATIONAL) KALIPLAR

ABSTRACT FACTORY

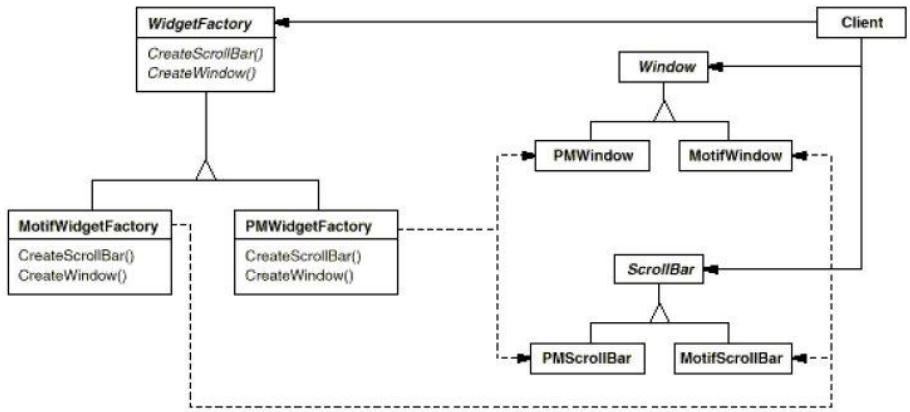
- Amaç:
  - Birbirleri ile ilişkili ya da aynı aileden olan nesneleri, sınıflarını belirtmeden oluşturabilmek.
- Örnek:
  - Java'da çeşitli GUI bileşenlerinin görünüşünün (look-and-feel) çalışma anında değiştirilebilmesi
- Sorun:
  - Taşınabilirliği sağlamak için, istekçinin hangi tür görünüşün hangi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

18

YARATIMSAL (CREATIONAL) KALIPLAR

ABSTRACT FACTORY:

- Örnek Çözüm:



19

YARATIMSAL (CREATIONAL) KALIPLAR

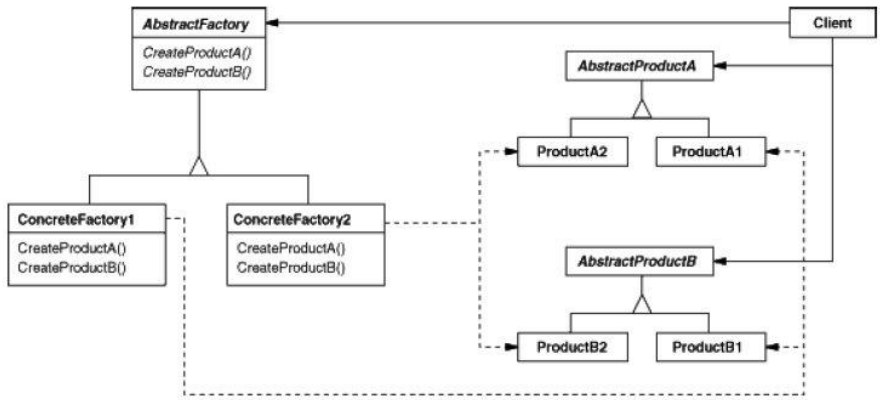
ABSTRACT FACTORY:

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

20

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.

21

YARATIMSAL (CREATIONAL) KALIPLAR

ABSTRACT FACTORY:

- Kalıbın uygulanabileceği anlar:
  - Bir sistemin, kendisine ait ürünlerin nasıl oluşturulduğu, birleştirildiği ve sunulduğundan bağımsız olmasının istendiği 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

22

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

23

ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

ÇÖZÜM 1: Switch – Case Kullanımı

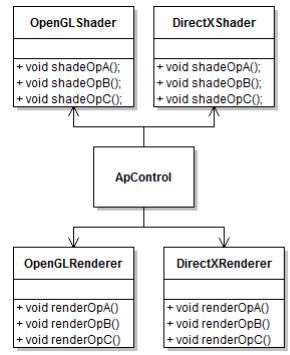
```
package dp.abstractFactory.solution1;
public class ApControl {
   private LibraryType library;
   private OpenGLRenderer or;
   private OpenGLShader os;
   private DirectXRenderer dr;
   private DirectXShader ds;
   public ApControl( ) { /*library nesnesini ilklendirerek kütüphaneyi belirle*/ }
   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;
      }
   }
}
```

24

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() { }
}
```



25

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:

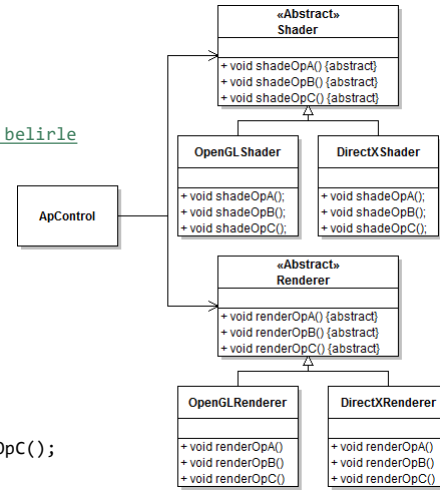
- Switch-case kullanımı çok biçimliliğe gereksinim duyulduğunun ve/veya sorumlulukların atanmasında yanlışlık olduğunun uyarı işareti olabilir.

26

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();
   }
}
```

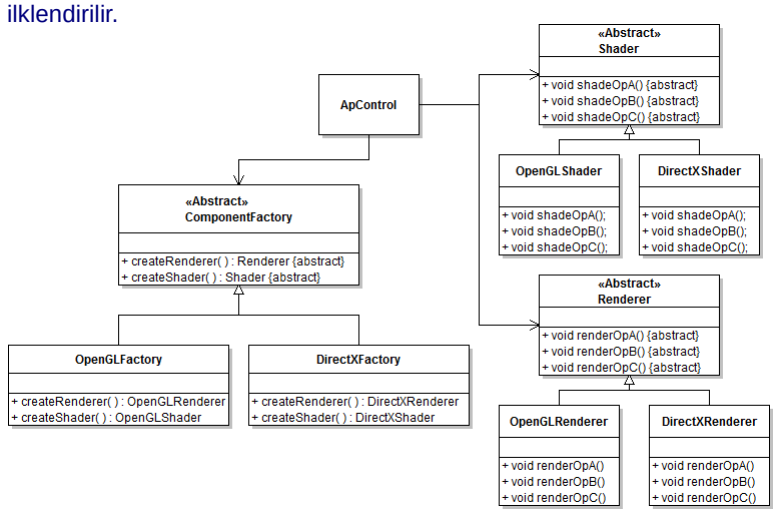


27

ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

ÇÖZÜM 3: Abstract Factory Kalıbının Kullanımı

- Ana program kurucusunda uygun bir ComponentFactory gerçeklemesi ilklendirilir.



28

ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

ÇÖZÜM 3: Abstract Factory Kalıbının Kullanımı

```
package dp.abstractFactory.solution3;
public class ApControl {
   private ComponentFactory library;
   private Renderer r;
   private Shader s;
   public ApControl( ComponentFactory library ) {
         this.library = library;
         /* Seçenek A: library nesnesini ilklendirerek kütüphane türünü
          * switch-case ile önceki örnekteki gibi belirle
          * Seçenek B: buradaki gibi kurucu metoda parametre olarak ver.
          * Seçenek C: Farklı tür hedef bilgisayarlar için
          * farklı dağıtımlar yap ve alttaki uygun satırın yorumunu kaldır.
         // library = new OpenGLFactory();
         // library = new DirectXFactory();
         */
         r = this.library.createRenderer();
         s = this.library.createShader();
   }
   void doRendering ( ) {
      r.renderOpA(); r.renderOpB(); r.renderOpC();
   }
   void doShading ( ) {
      s.shadeOpA(); s.shadeOpB(); s.shadeOpC();
   }
}
```

29

ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

ÇÖZÜM 3: Abstract Factory Kalıbının 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();
   }
}
```

30

ÖRNEK KALIP GERÇEKLEMELERİ – ABSTRACT FACTORY

ÇÖZÜM 3: Abstract Factory Kalıbının Kullanımı

- Refactoring konusunu gördükten sonra, Preserve Whole Object eyleminin yürütülmüş hali tercih edilir:
  - Neden? Bize ne kazandırır?

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

31

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
  - Kalıtım farklı bir sakıncalı biçimde kurulabilirdi: ApControl üst sınıfından OpenGLApControl ve DirectXApControl sınıfları türetilebilirdi.
    - Sonuçta ortaya benzer kodun birden fazla yerde tekrarlanması nedeniyle kusurlu bir tasarım ortaya çıkacaktı.
- Gerçeklemelere göre değil, arayüzlere 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.

32

YARATIMSAL (CREATIONAL) KALIPLAR

FACTORY METHOD:

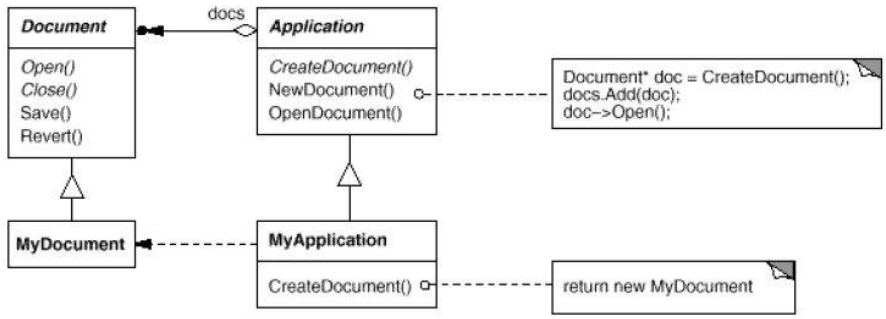
- Amaç:
  - Bir nesne oluşturmak için öyle bir ara yüz sunmak ki, oluşturulacak nesnenin sınıfına bu ara yüzü sağlayan sınıfın alt sınıfları karar verebilsin.
- Örnek:
  - Farklı belge türleri ile çalışabilecek uygulamalar geliştirmeyi sağlayan bir çerçeve program (framework) hazırlanıyor.
  - Temel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

33

YARATIMSAL (CREATIONAL) KALIPLAR

FACTORY METHOD:

- Örnek Çözüm:



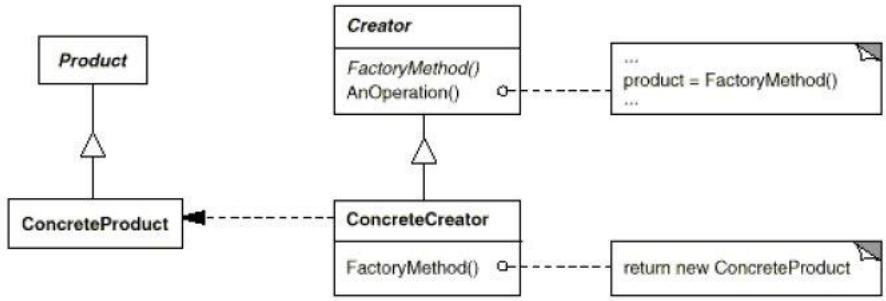
- Diğer seçenekler:
  - Java'da Generic sınıflar:
    - class Document<DocType>
    - Zayıf yönü: Dile özel
  - Başka?

34

YARATIMSAL (CREATIONAL) KALIPLAR

FACTORY METHOD:

- Kalıp yapısı:



35

YARATIMSAL (CREATIONAL) KALIPLAR

FACTORY METHOD:

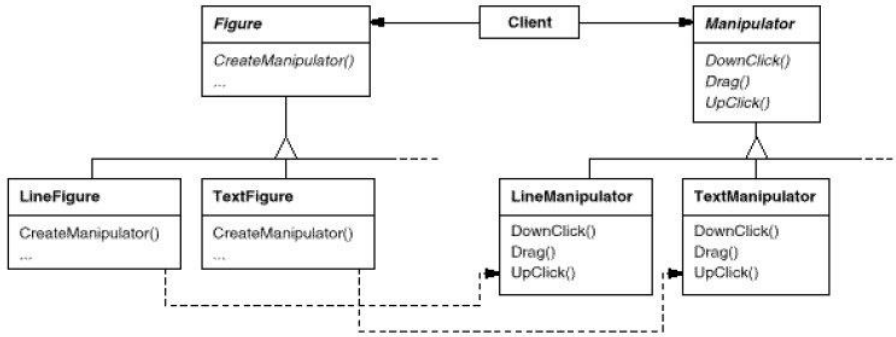
- Kalıbın uygulanabileceği anlar:
  - Yeni nesneler oluşturması gereken bir sınıfın, oluşturacağı nesnenin türünü bilemediği yerlerde,
  - Bir sınıfın kendi üzerindeki bir sorumluluğu çeşitli yardımcı alt sınıflarına ileteceği ve hangi tür yardımcı alt sınıfın oluşturulması gerektiği bilgisinin ise yerelleştirilmesi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

36

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.

37

ÖRNEK KALIP GERÇEKLEMELERİ – FACTORY METHOD

ÇÖZÜLECEK PROBLEM:

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

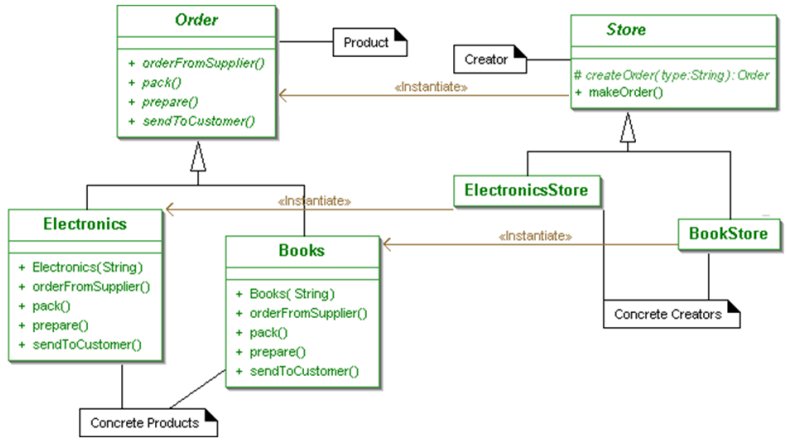
```
package factoryMethod.example;
public class MainApp {
    public static void main(String[] args) {
        Store store;
        //Elektronik ürün siparişi geldi
        store = new ElectronicsStore( );
        store.createOrder("5524345678");
        //Kitap siparişi geldi
        store = new BookStore(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

- Esinlenme: Head First DP

38

ÖRNEK KALIP GERÇEKLEMELERİ – FACTORY METHOD

ÇÖZÜM:



39

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*/ }
```

40

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;
    }
}
```

41

YARATIMSAL (CREATIONAL) KALIPLAR

BUILDER:

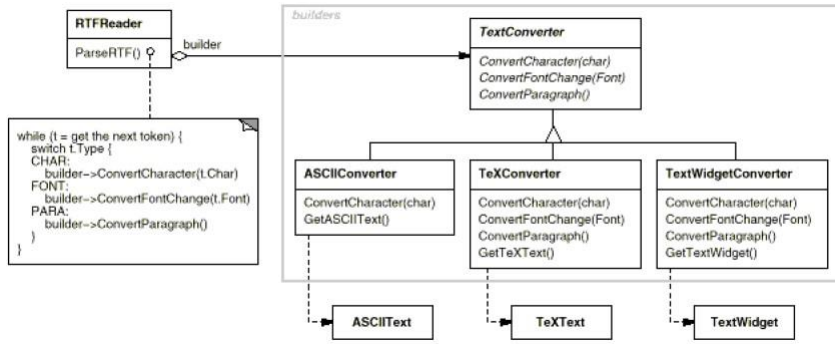
- Amaç:
  - Karmaşık bir nesnenin oluşturulması ile gösterimini birbirinden ayırmak. Böylece aynı oluşturma işlemi farklı gösterimler ortaya çıkarabilir.
- Örnek:
  - Bir RTF metin okuyucu programı, bu belgeyi çeşitli diğer metin türlerine, hatta metni düzenleyebilecek bir GUI bileşenine çevirebilsin.
- 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 çevirici nesnesine çeviri isteğini göndersin.
  - Çevirici nesne hem çevrim işleminden, hem de çevrilen metni belli bir biçimde simgelemekle yükümlü olsun.
  - Bu durumda yeni metin türleri için çevirici sınıfının alt sınıfları oluşturulabilir.

42

YARATIMSAL (CREATIONAL) KALIPLAR

BUILDER:

- Çözüm:



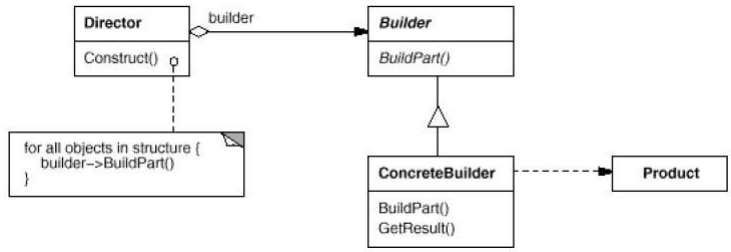
- Çözümün eleştirileri:
  - Çokbiçimliliği tam kullanamadık, switch-case hala duruyor.
  - Alt sınıflar üst sınıfın tüm metotlarını gerçeklemelidir. Bu yüzden alt sınıfta anlamı olmayan metotlar boş gövdeli olarak kalmak zorunda.

43

YARATIMSAL (CREATIONAL) KALIPLAR

BUILDER:

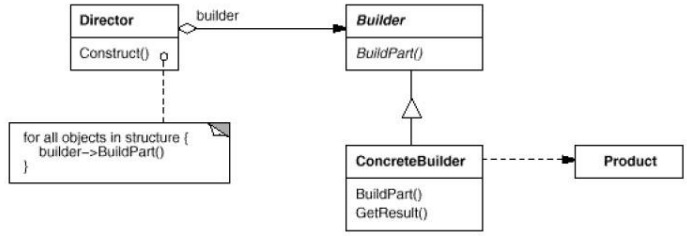
- Kalıp Yapısı:



- 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 aktarılmak zorunda kalınacaktır ve “parça şu ise şunu yap, bu ise bunu yap” gibi komutlar verilmek zorunda kalınacaktır.

44

BUILDER:



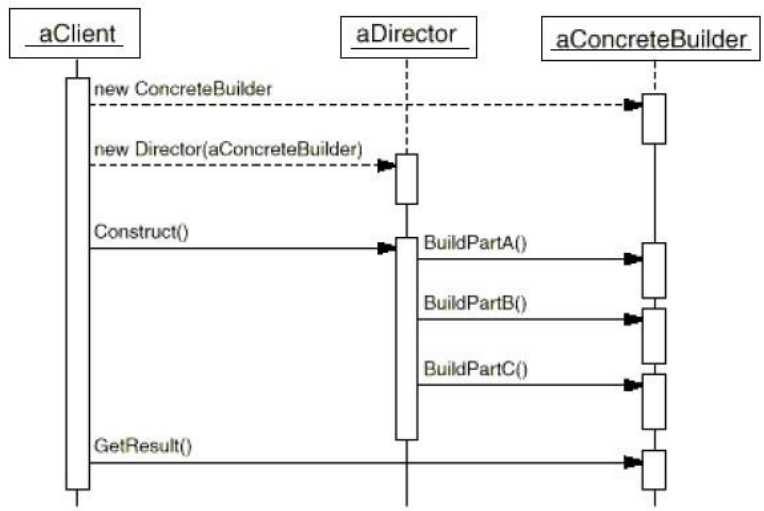
- Kalıp Bileşenleri:
  - Builder: Ürün nesnesinin parçalarını oluşturmak için bir arayüz sunar.
  - Product: Ürün.
    - Kendisini oluşturan parçaları tanımlayan sınıfları içerir.
    - Parçaları bir bütün haline getirmek için gerekli arayüzleri içerir.
  - Director: Ürünü, Builder arayüzünü kullanarak 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.

45

YARATIMSAL (CREATIONAL) KALIPLAR

BUILDER:

- Kalıp bileşenlerinin etkileşimleri:



46

YARATIMSAL (CREATIONAL) KALIPLAR

BUILDER:

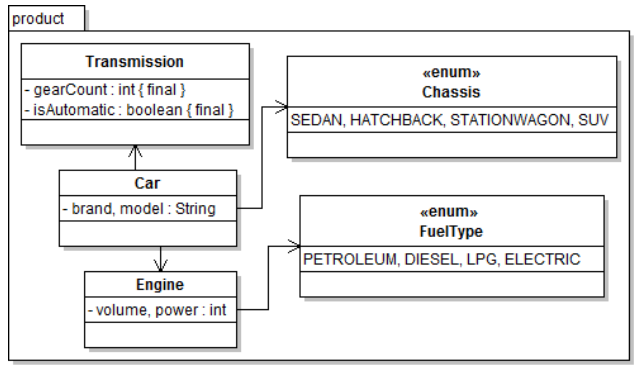
- Kalıbın uygulanabileceği anlar:
  - Karmaşık bir nesneyi oluşturma algoritmasının, nesneyi oluşturan parçalardan bağımsız olması gerektiği ve bu parçaların nasıl birleştirileceğinden bağımsız olması gerektiği anlarda.
  - Bir nesneyi oluşturma sürecinin bu nesnenin farklı gösterimlerinin olmasına izin verebilmesi gerektiği anlarda.
- Sonuçlar:
  - Ürünün iç yapısının en az yan 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 gerektiği emrini verir).

47

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.



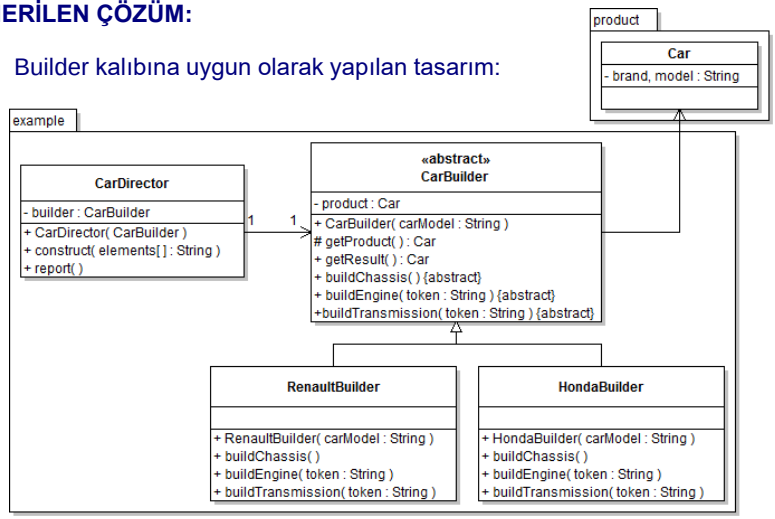
- Kaynak: YES

48

ÖRNEK KALIP GERÇEKLEMELERİ – BUILDER

ÖNERİLEN ÇÖZÜM:

- Builder kalıbına uygun olarak yapılan tasarım:

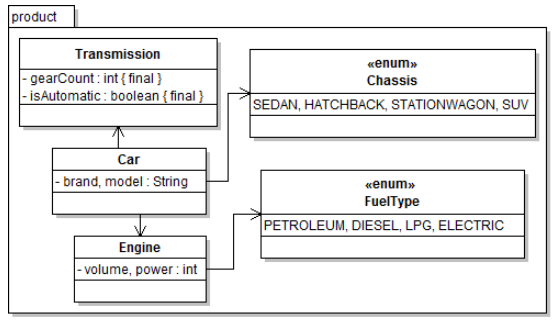


49

ÖRNEK KALIP GERÇEKLEMELERİ – BUILDER

ÖNERİLEN ÇÖZÜM:

- Product paketini açarsak:



50

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( );//Şasi'yi modele bağladım.
    public abstract void buildEngine(String token);
    public abstract void buildTransmission(String token);
}
```

51

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();
    }
}
```

52

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);
    }
```

53

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) );
        }
    }
```

54

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) );
}
}
```

55

YARATIMSAL (CREATIONAL) KALIPLAR

PROTOTYPE:

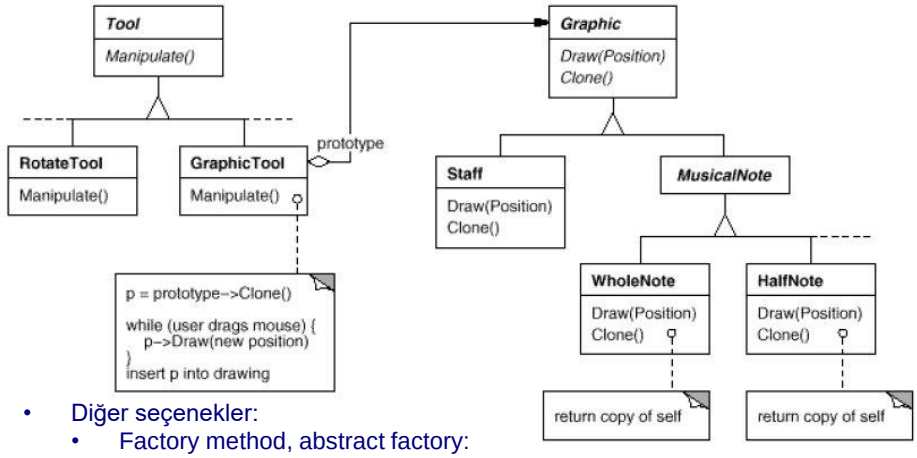
- Amaç:
  - Oluşturulacak nesnelerin türlerini belirlemek için bir prototip nesne kullanmak ve yeni nesneleri bu prototipi kopyalayarak oluşturmak.
- Örnek:
  - Grafik uygulamalarına yönelik mevcut bir çerçeve programını kullanarak, bir müzik besteleme programı yazılacak.
  - Çerçeve programının soyut bir Graphic sınıfı, çeşitli kontrol düğmelerinin yer alabileceği paletleri simgeleyen soyut bir Tool sınıfı, bundan kalıtımla 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?

56

YARATIMSAL (CREATIONAL) KALIPLAR

PROTOTYPE:

- Örnek Çözüm:



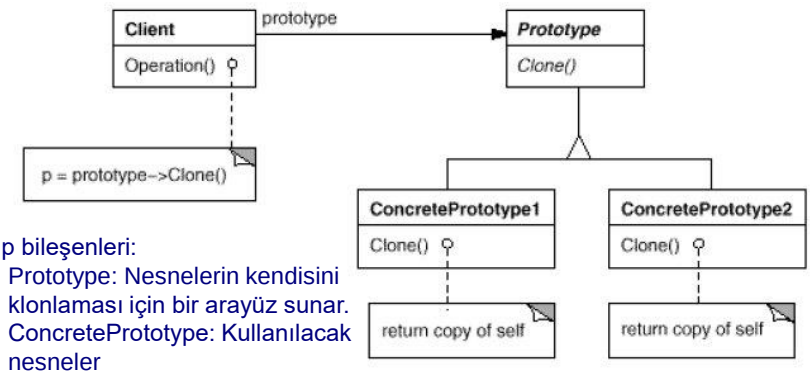
- Diğer seçenekler:
  - Factory method, abstract factory:
    - Her nota için ayrı fabrika sınıfı gerekecek.
  - Başka?

57

YARATIMSAL (CREATIONAL) KALIPLAR

PROTOTYPE:

- Kalıp yapısı:



- Kalıp bileşenleri:
  - Prototype: Nesnelerin kendisini klonlaması için bir arayüz sunar.
  - ConcretePrototype: Kullanılacak nesneler
  - Client: Bir prototipin kendisini kopyalamasını isteyerek yeni bir nesne oluşturur.

58

YARATIMSAL (CREATIONAL) KALIPLAR

PROTOTYPE:

- Kalıbın uygulanabileceği anlar:
  - Bir sistemin, kendisine ait ürünlerin nasıl oluşturulduğu, birleştirildiği ve sunulduğundan bağımsız olmasının istendiği 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 tasarlanıp tasarlanmadığı 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.

59

YARATIMSAL (CREATIONAL) KALIPLAR

PROTOTYPE:

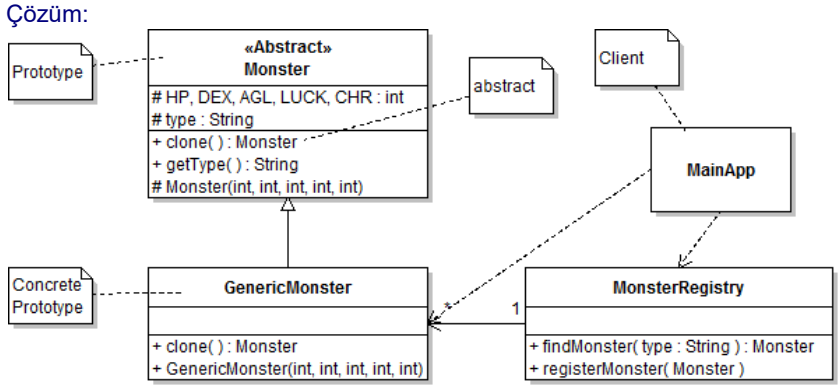
- Tartışmalar:
  - Sığ kopyalama veya derin kopyalama (shallow or deep copy)
    - Sığ kopyalama yetersiz kalabilir
    - Çevrimsellik derin kopyalamada sorun çıkartır.
  - Klonların üyelerine değer atanması için Prototype sınıfında ek metotlar tanımlamak gerekebilir.

60

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

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; }
}
```

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 );
    }
}
```

63

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);
    }
}
```

64

YARATIMSAL (CREATIONAL) KALIPLAR

SINGLETON:

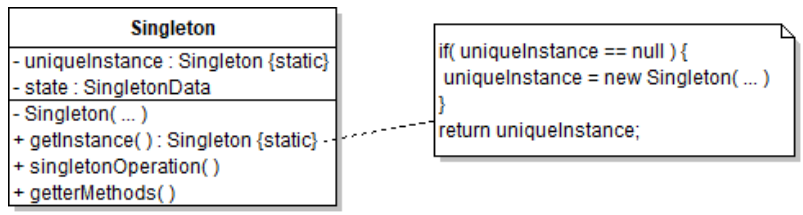
- Amaç:
  - Bir sınıfın sadece bir örneğinin bulunmasını sağlamak ve bu nesneye ortak bir erişim noktası vermek.
- Örnek:
  - Bazı nesnelerin türünün tek örneği olmasının gerekli olduğu alanlar bulunabilir.
  - Bir bilgisayarda birden fazla yazıcı tanımlı olabilmesine rağmen bir tek yazdırma biriktiricisi 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?

65

YARATIMSAL (CREATIONAL) KALIPLAR

SINGLETON:

- Önerilen Çözüm:
  - Bir sınıfa, 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.



66

YARATIMSAL (CREATIONAL) KALIPLAR

SINGLETON:

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

ÖRNEK KALIP GERÇEKLEMELERİ – SINGLETON

- Basit bir kalıp olduğundan bu aşamada örnek yapılmayacaktır.

67

Bu yansı ders notlarının düzeni için boş bırakılmıştır.

68

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

69

YAPISAL (STRUCTURAL) KALIPLAR

ADAPTER

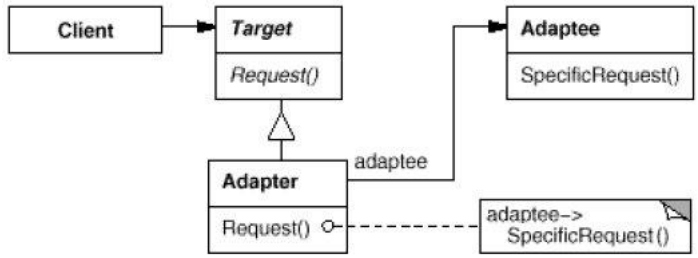
- Amaç:
  - İşlev açısından uyumlu ancak metot adları açısından uyumsuz bir istekçi sınıf ile bir sunucu sınıfı, birbirleri ile tek yönlü konuşturmak.
- Örnek:
  - İngiliz standardındaki elektrik prizine Avrupa standardındaki cihazı takabilmek için bir adaptör kullanmak gerekir.
  - Adapter kalıbı da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

70

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.

71

YAPISAL (STRUCTURAL) KALIPLAR

ADAPTER:

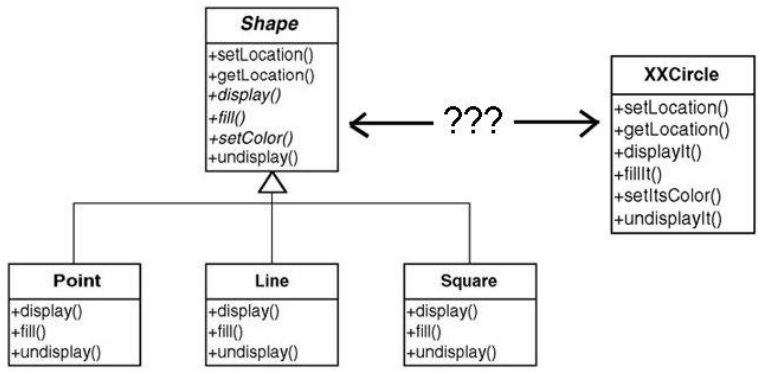
- Kalıbın uygulanabileceği anlar:
  - Elimizdeki bir sınıfın bir istekçi tarafından kullanılabilmesini önleyen tek engelin, istekçinin yolladığı mesajlar ile eldeki sınıfın anladığı mesajların adlarının farklı olduğu anlarda.

72

ÖRNEK KALIP GERÇEKLEMELERİ – ADAPTER

ÇÖZÜLECEK PROBLEM:

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



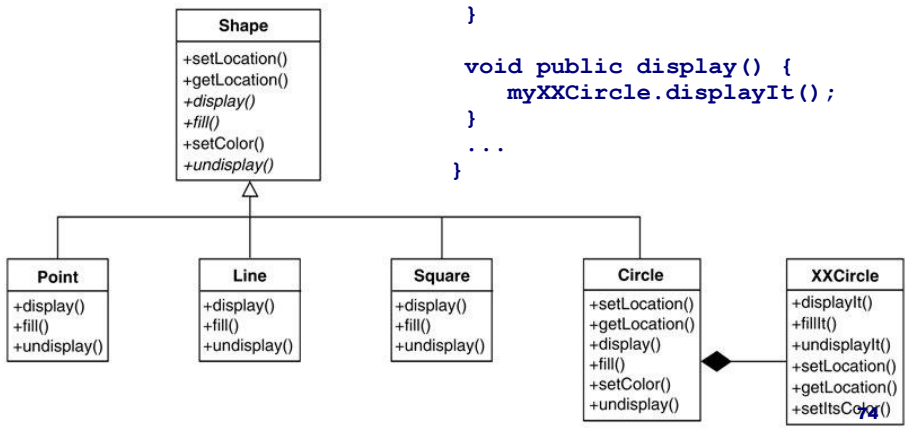
- Daire kodunu yazmadan önce farkettik ki, elimizde hazır bir daire sınıfı var.
- Ancak bu daire sınıfı mevcut programımıza uymuyor (yukarıda sağda).

73

ÖRNEK KALIP GERÇEKLEMELERİ – ADAPTER

ÇÖZÜM:

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

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



74

YAPISAL (STRUCTURAL) KALIPLAR

BRIDGE:

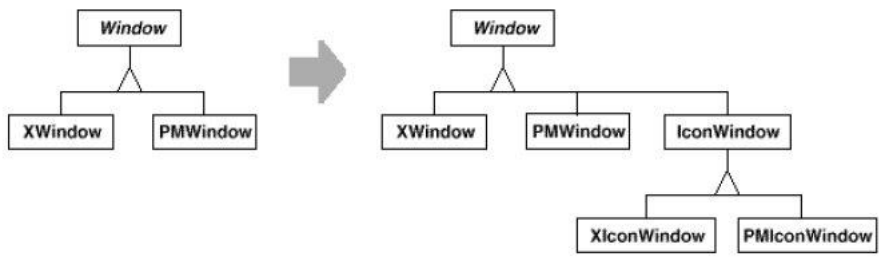
- Amaç:
  - Nesnenin arayüzü ile gerçeklemesini birbirinden ayırmak.
  - 'Handle' olarak da bilinir (Win32 programcıları!)
- Örnek:
  - Bir GUI çerçeve programının pencere soyutlaması.
  - Gnome, KDE gibi farklı masaüstü ortamlarının desteklenmesi isteniyor.
  - Pencere soyut sınıfından kalıtımla yeni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

75

YAPISAL (STRUCTURAL) KALIPLAR

BRIDGE:

- Sorun:

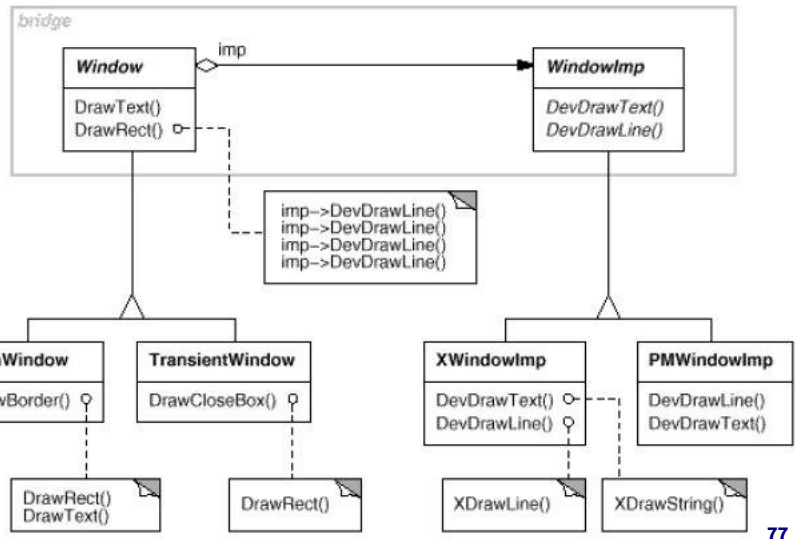


- Çözüm:
  - Pencere soyutlaması ve gerçeklemeleri ayrı sınıf hiyerarşilerine koyulur
  - Farklı gerçeklemeler soyutlama nesnesinde saklanır.
  - İşlemler soyutlamalar üzerinden çağırılır.
  - Soyutlama ile gerçekleme arasında **köprü** kurulmuş olur.

76

YAPISAL (STRUCTURAL) KALIPLAR

BRIDGE:

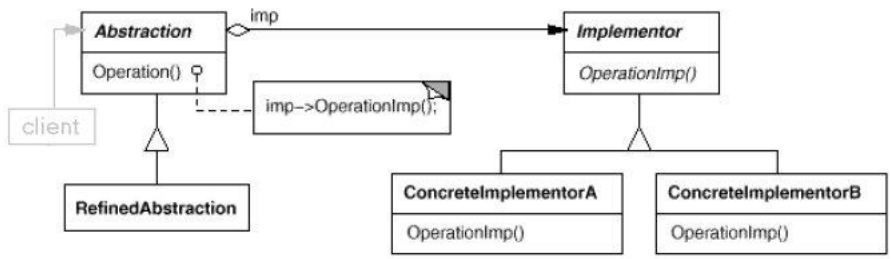


77

YAPISAL (STRUCTURAL) KALIPLAR

BRIDGE:

- Kalıp Yapısı:



- Kalıp bileşenleri:
  - Abstraction: Uygulamanın temel işlemlerini oluşturan arayüzünü belirler.
  - RefinedAbstraction: Arayüzü özelleştirmeye yarar.
  - Implementor: Uygulamanın iç yapısına yönelik arayüzünü belirler.
  - ConcreteImplementor: İç yapıyı gerçekleyen sınıf hiyerarşisi.

78

YAPISAL (STRUCTURAL) KALIPLAR

BRIDGE:

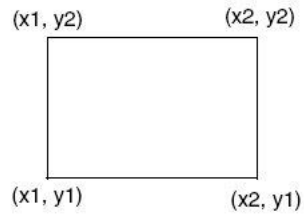
- Kalıbın uygulanabileceği anlar:
  - Bir soyutlama ile gerçeklemeleri arasındaki bağın kalıcı olmasının istenmediği anlarda.
  - Hem soyutlamaların hem de gerçeklemelerin ayrı ayrı özelleştirilmesine gerek duyulduğu anlarda.

79

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 çizileceğini biliyor.



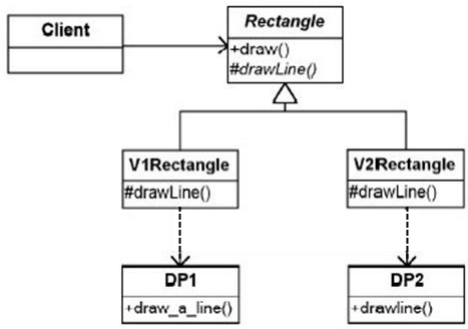
	DP1	DP2
Used to draw a line	`draw_a_line( x1, y1, x2, y2)`	`drawline( x1, x2, y1, y2)`

80

ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

ÖNERİLEN ÇÖZÜM (1):

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



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

81

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);
    }
}
```

82

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);
    }
}
```

83

ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

ÇÖZÜMÜN ZAYIF YÖNLERİ:

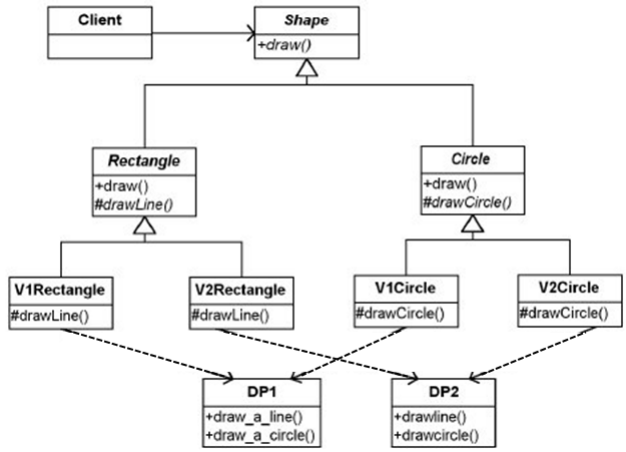
- Yeni bir gereksinim ortaya çıkıyor:
  - Çizim programındaki şekillere iki farklı çember ailesi de ekleniyor.
  - Farklı çember ailelerinin farklı çizim metotları var.
  - 1. dikdörtgen ailesi ile 1. çember ailesi ortaktır.
  - 2. dikdörtgen ailesi ile 2. çember ailesi ortaktır.
  - Bu durumda çember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

84

ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

ÇÖZÜMÜN ZAYIF YÖNLERİ:

- Yeni bir gereksinimi karşılayan sınıf şeması:



85

ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

ÇÖZÜMÜN ZAYIF YÖNLERİ:

- Değerlendirme:
  - Mevcut durumda 4 belirli şekil var (2 aile ve 2 şekil, 2*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*3=9).
  - Her yeni eklenecek çizim metodu veya şekil türü ile sınıf sayısı fırlayacak!

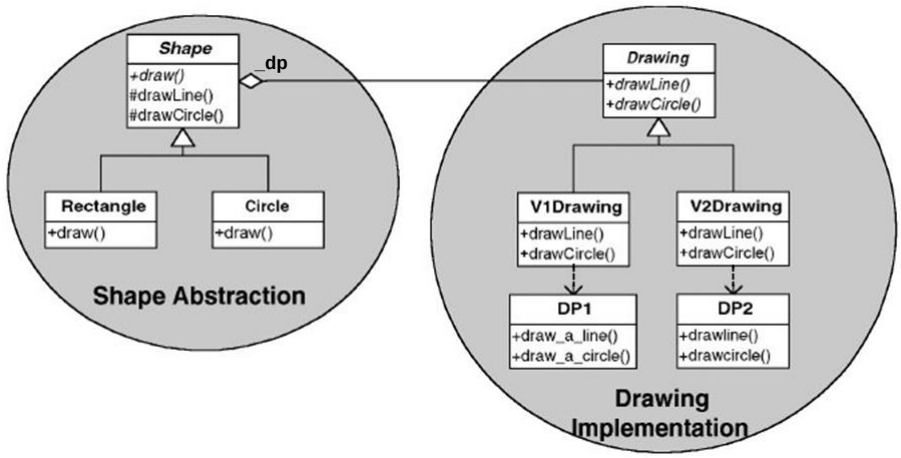
YENİ ÇÖZÜM ÖNERİSİ:

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

86

ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

BRIDGE KALIBI İLE ÇÖZÜM:



87

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); }
}
```

88

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);
    }
}
```

89

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);
    }
}
```

90

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);
    }
}
```

91

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
    }
}
```

92

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
    }
}
```

93

ÖRNEK KALIP GERÇEKLEMELERİ – BRIDGE

ÇIKARTILAN DERSLER = TASARIMA AİT GENEL KURALLAR

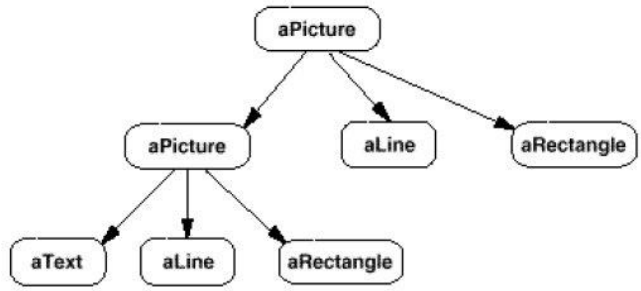
- Neyin değiştiğini bul ve onu sarmala.
  - Değişik türden şekiller ve değişik türden çizimler ortaya çıktı.
- Kalıtım ilişkisi yerine parça/bütün ilişkisini tercih et
  - Şekiller ve çizim türlerini aynı sınıflarda toplamak bağlaşımı arttırdı ve yeni bir şekil/çizim türü eklemek gerekli sınıf sayısını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

94

YAPISAL (STRUCTURAL) KALIPLAR

COMPOSITE:

- Amaç:
  - Hiyerarşi şeklindeki parça-bütün ilişkilerini desteklemeye yöneliktir.
  - Bu kalıp sayesinde bireysel nesneler ve bir nesneler bütünü, istekçilere aynı şekilde gözükür.
- 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.

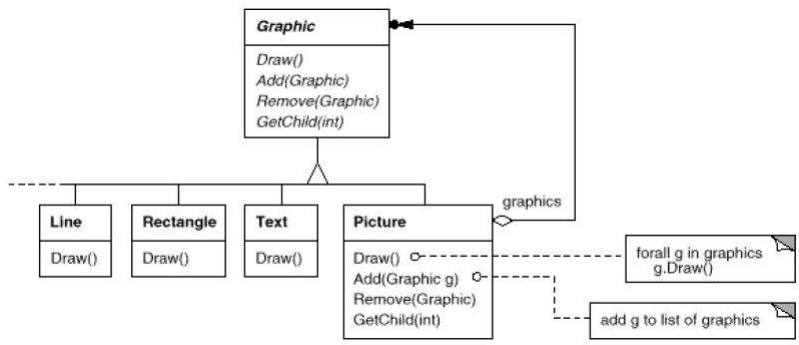


95

YAPISAL (STRUCTURAL) KALIPLAR

COMPOSITE:

- Örnek çözüm:



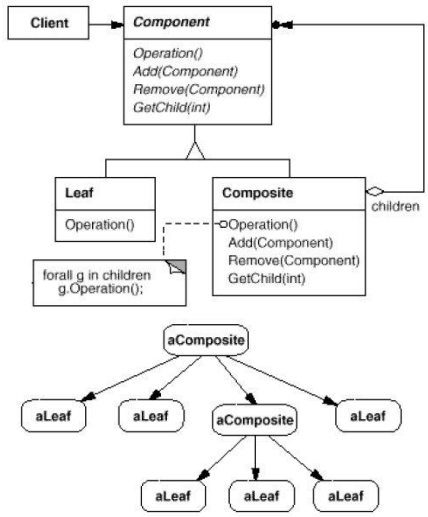
- Add ve Remove komutları ile çeşitli şekiller bir resim grubu altında toplanabilir.
- Bir grubun içerdiği şekillerden birine GetChild metodu ile erişilebilir.

96

YAPISAL (STRUCTURAL) KALIPLAR

COMPOSITE:

- Kalıp yapısı:



- Kalıp katılımcıları:
  - Component:
    - Nesneler topluluğunun katılımcıları için ortak kullanım arayüzü sunar.
    - Bir bütünün parçaları için erişim ve düzenleme metotları tanımlar.
  - Leaf:
    - Topluluktaki parçaları simgeler.
    - Hiyerarşik yapının en altındaki düğümlerdir, çocukları olamaz.
  - Composite:
    - Hiyerarşik yapının ara düğümleridir.
    - Farklı tür ara düğümler varsa bu sınıfın alt sınıfları şeklinde modellenir.
  - Client: Parça ve toplulukları benzer şekilde kullanabilir.
    - Client bir Visitor olabilir.

97

YAPISAL (STRUCTURAL) KALIPLAR

COMPOSITE:

- Kalıbın kullanılabileceği anlar:
  - Nesneler arasındaki hiyerarşik parça-bütün ilişkilerinin modellenmesi istenildiğinde.
  - Bir nesneler topluluğu ve topluluktaki bireysel nesnelere eş biçimli erişim gerekli olduğunda.
- Gerçekleme ayrıntıları:
  - Alt düğüm, üst düğümden haberdar kılınabilir.
  - Bir düğüm nasıl silinir? Kim siler? Nasıl bir veri yapısı kullanılır?
  - children üyesi belki üst sınıfa taşınabilir (zaten aşağıdaki ilk zayıflık mevcut olduğundan yapılabilir bir eylemdir).
  - …
- Kalıbın zayıf yönleri:
  - Kalıtım ilişkisinin özüne aykırı bir davranışta bulunur: Leaf bir alt sınıf olmasına rağmen, üst sınıfı olan Component'teki çocuk yönetim metotlarının bu alt sınıf için bir anlamı yoktur.
  - Halbuki alt sınıf üst sınıftaki tüm metotları kalıtımla almak zorundadır.
  - Programcının karar vermesi gereken gerçekleme ayrıntısı sayısı, diğer kalıplara göre çok fazladır.

98

ÖRNEK KALIP GERÇEKLEMELERİ – COMPOSITE

ÇÖZÜLECEK PROBLEM:

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

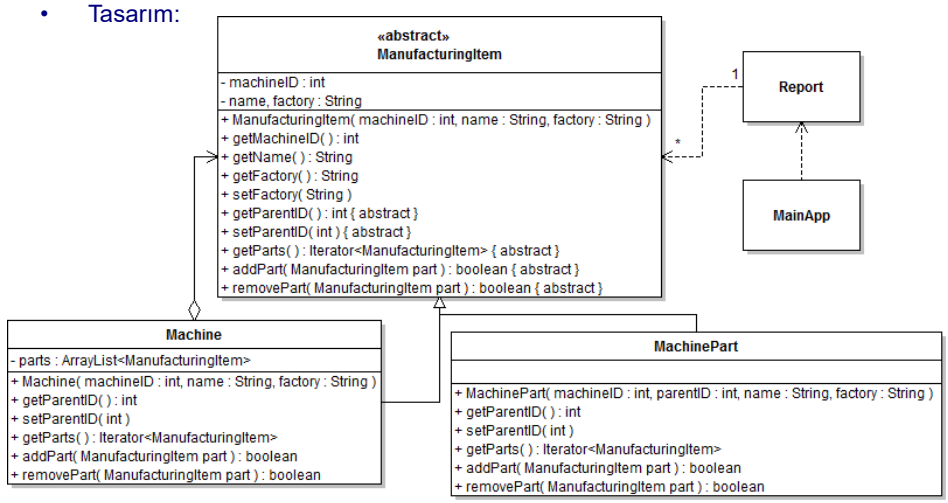
- Esinlenme: DP Java Workbook

99

ÖRNEK KALIP GERÇEKLEMELERİ – COMPOSITE

COMPOSITE KALIBI İLE ÇÖZÜM

- Tasarım:



100

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 );
}
```

101

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();
   }
}
```

102

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; }
}
```

103

```
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;
   }
}
```

104

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 şaftı KŞ-7", "Bursa");
        Machine jenerator = new Machine(135, "Jeneratör JMX-99", "İstanbul");
        jenerator.addPart(new MachinePart(259,135,"Flanş 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 sayi = Report.findItems(items, "Bursa");
        System.out.println("\t" + sayi + " parça bulundu.");
   }
}
```

105

YAPISAL (STRUCTURAL) KALIPLAR

DECORATOR:

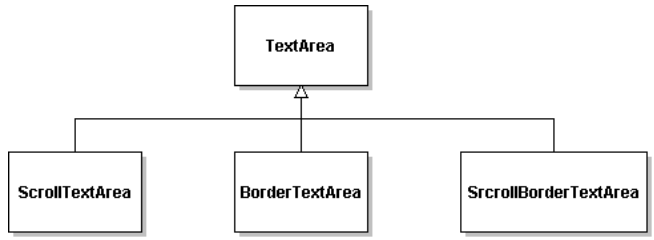
- Amaç:
  - Nesnelere çalışma anında (dinamik olarak) ek sorumluluklar atamak.
    - Bu iş kalıtımla ancak kodlama anında yapılabilir.
  - Böylece bireysel nesneler özelleştirilebilir.
    - Kalıtımda özelleşme sınıf düzeyinde olduğundan, aynı tipteki bütün nesneler birden özelleşecektir.
- Örnek:
 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

106

YAPISAL (STRUCTURAL) KALIPLAR

DECORATOR:

- Çözüm 1:



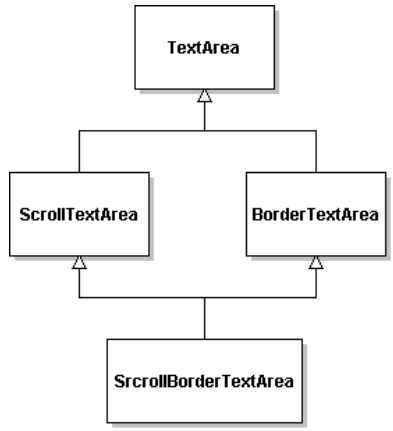
- Çözümün zayıf yönleri:
  - Scroll ve Border yetenekleri iki ayrı yerde tekrar gerçeklenmek zorunda.
  - TextArea'ya ek olarak bir de Panel sınıfına Scroll ve Border yeteneği kazandırmak için ek üç sınıf daha gerekecek.

107

YAPISAL (STRUCTURAL) KALIPLAR

DECORATOR:

- Çözüm 2:



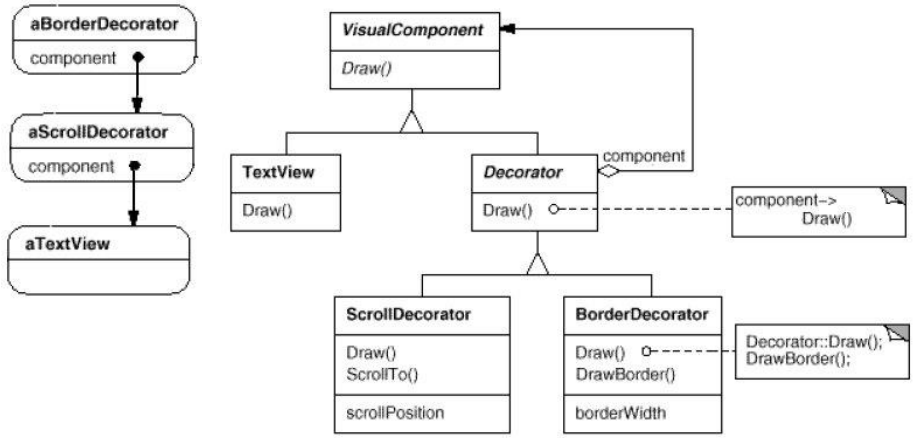
- Çözümün zayıf yönleri:
  - Çoklu kalıtım her dilde desteklenmez.
  - Çoklu kalıtımın bu çözümdeki şekli 'diamond problem' sorununun bir örneğidir.
  - TextArea'ya ek olarak bir de Panel sınıfına Scroll ve Border yeteneği kazandırmak için ek üç sınıf daha gerekecek.
    - Üstelik bu durumda bu örneğin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

108

YAPISAL (STRUCTURAL) KALIPLAR

DECORATOR:

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

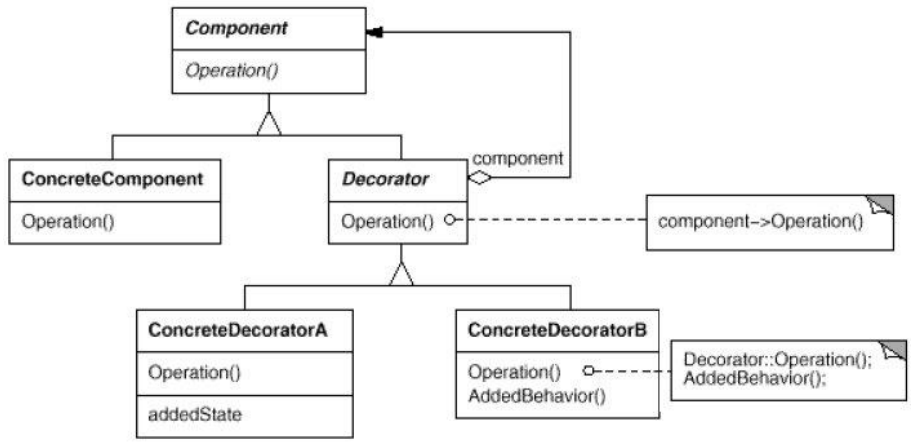


109

YAPISAL (STRUCTURAL) KALIPLAR

DECORATOR:

- Kalıp yapısı:



110

YAPISAL (STRUCTURAL) KALIPLAR

DECORATOR:

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

111

YAPISAL (STRUCTURAL) KALIPLAR

DECORATOR:

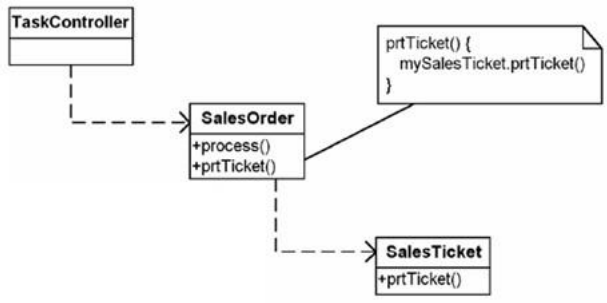
- Kalıbın kullanılabileceği anlar:
  - Tek tek nesnelere dinamik olarak (çalışma anında) yeni sorumluluklar eklenmek istediğinde
  - Tek tek nesnelere şeffaf olarak (diğerlerini etkilemeden) yeni sorumluluklar eklenmek istediğinde
  - Bazı sorumlulukların iptal edilebileceği veya askıya alınabileceği 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 kalıbı kullanır.
- Kalıbın 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.

112

ÖRNEK KALIP GERÇEKLEMELERİ – DECORATOR

ÇÖZÜLECEK PROBLEM:

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



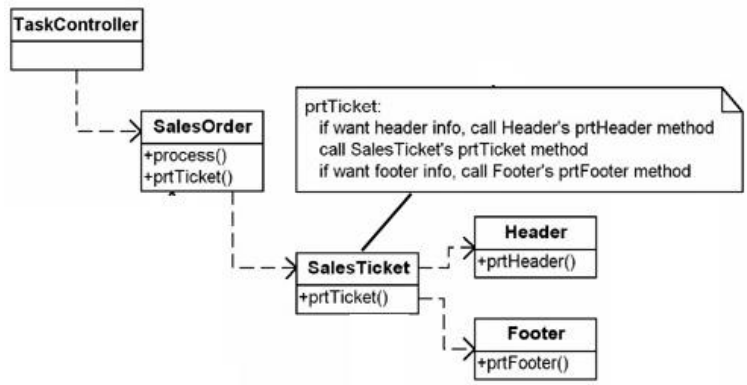
- Yeni gereksinim: Makbuzlara üstbilgi ve altbilgi (header/footer) eklemek.

113

ÖRNEK KALIP GERÇEKLEMELERİ – DECORATOR

ÇÖZÜM 1:

- Kullanım ilişkisi ile:



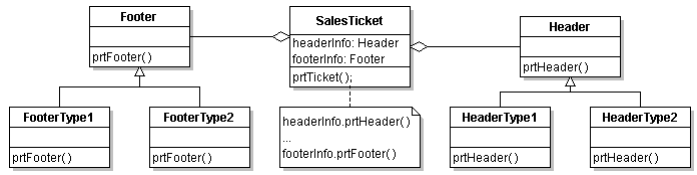
- Makbuz basılacağı zaman SalesTicket sınıfında ayrı if sınamaları ile üstbilgi ve/veya altbilgi basılıp basılmayacağına karar verilir.
- Farklı üstbilgi ve altbilgi türleri söz konusu ise ne olacak?

114

ÖRNEK KALIP GERÇEKLEMELERİ – DECORATOR

ÇÖZÜM 2:

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



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

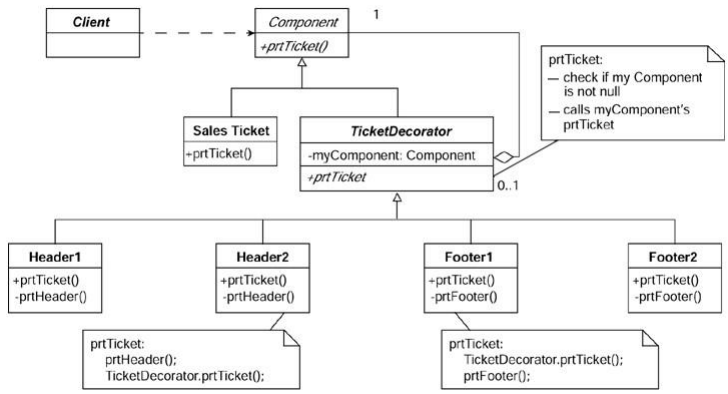
HEADER1 HEADER2 Receipt FOOTER2	HEADER1 Receipt FOOTER1 FOOTER2	...

- Bu durumda ancak Decorator kalıbı gereksinimi karşılar.

115

ÖRNEK KALIP GERÇEKLEMELERİ – DECORATOR

ÖNERİLEN ÇÖZÜM:



116

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();
     }
}
```

117

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();
     }
}
```

118

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());
   }
}
```

119

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();
    }
}
```

120

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:

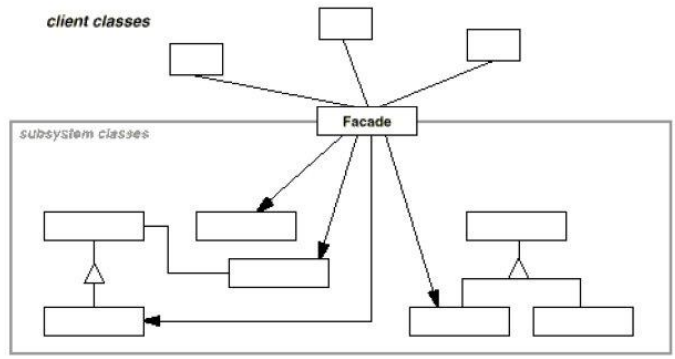
```
HEADER1
Receipt
FOOTER1
```

121

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:



122

YAPISAL (STRUCTURAL) KALIPLAR

FACADE:

- Kalıbın uygulanabileceği anlar:
  - Karmaşık bir alt sisteme basit bir arayüz sağlanmak 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.

123

YAPISAL (STRUCTURAL) KALIPLAR

FLYWEIGHT:

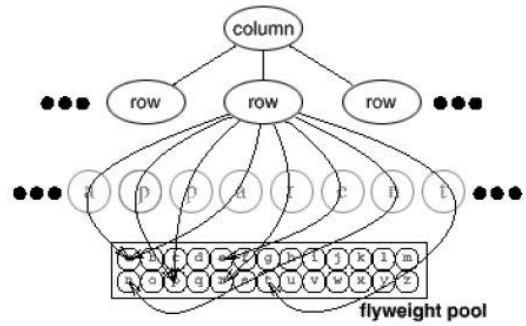
- Amaç:
  - Paylaşım yolu ile, çok sayıda basit nesnenin kullanımının desteklenmesi.
- Örnek:
  - Bir metin düzenleme programı yazılıyor.
  - Programın font ve diğer biçimlendirme bilgilerine göre, metin bilgisini çok net göstermesi isteniyor (excellent rendering).
  - Harfleri grafik nesneleri olarak 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.

124

YAPISAL (STRUCTURAL) KALIPLAR

FLYWEIGHT:

- Çözüm:
  - Unicode alfabesindeki 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 paylaşımlı kullanımını sağlayalım.
  - Bu tip paylaşılan nesnelere flyweight adı verilir.

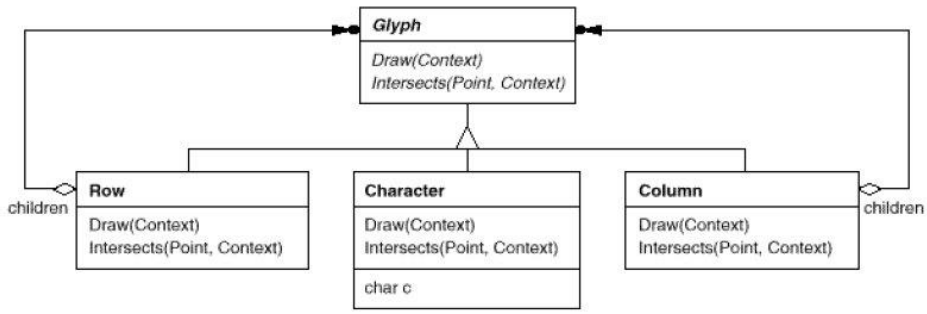


125

YAPISAL (STRUCTURAL) KALIPLAR

FLYWEIGHT:

- Örnek çözüm:

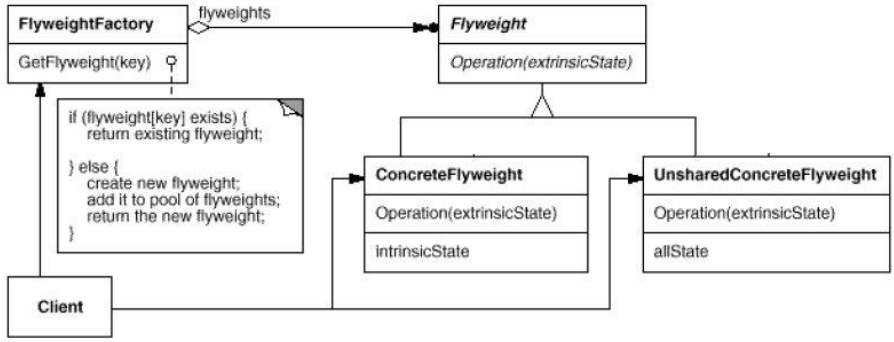


126

YAPISAL (STRUCTURAL) KALIPLAR

FLYWEIGHT:

- Kalıp yapısı:



127

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 kalıbın 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 metotlarını bir arayüzde tanımlayıp Flyweight sınıfını fabrikanın private iç sınıfı olarak kodlayarak fabrikanın nesne işaretçisi yerine arayüz işaretçisi göndermesi sağlanabilir (örnekte kodlanacak).

128

YAPISAL (STRUCTURAL) KALIPLAR

FLYWEIGHT:

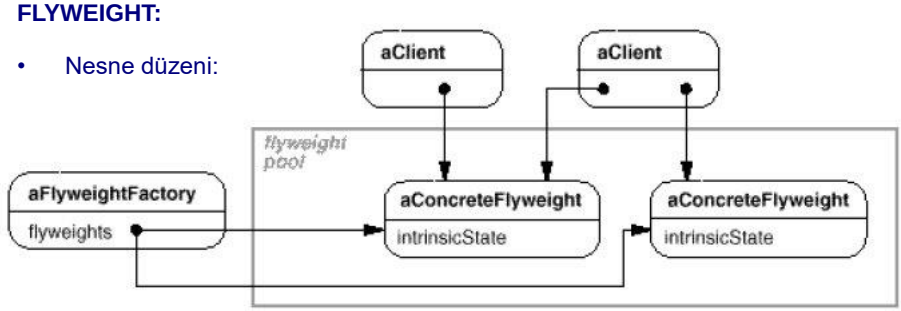
- Gerçekleme ayrıntıları (devam):
  - Kalıbın etkili olabilmesi için, paylaşılacak nesnelerin durum bilgisinin büyük bir kısmı çalışma anında hesaplanabilir olmalıdır.
    - Bu sayede flyweight nesnelerinin saklanması gereken durum bilgisi çok azalır ve kalıbın etkinliği artar.
    - Saklanması gereken durum bilgisine içsel (intrinsic), hesaplanabilen ve böylece harici bir nesneden bu nesneye bir metot çağrısı 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.

129

YAPISAL (STRUCTURAL) KALIPLAR

FLYWEIGHT:

- Nesne düzeni:



- Kalıp bileşenleri:
  - Flyweight: Paylaşılacak nesnelerin arayüzünü 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).

130

YAPISAL (STRUCTURAL) KALIPLAR

FLYWEIGHT:

- Kalıbın 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 dışsallaştırlabilmesi,
  - Dışsal durumları ayrılan nesnelerin bir çok grubunun yerini, paylaşılabilecek az sayıda nesnenin alabileceği,
  - Uygulamanın nesne kimliğine bağımlı olmadığı anlarda.

131

ÖRNEK KALIP GERÇEKLEMELERİ – FLYWEIGHT

ÇÖZÜLECEK PROBLEM:

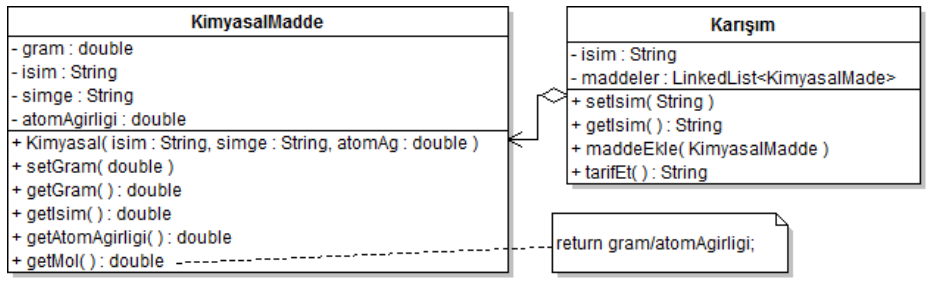
- Bir kimya fabrikasının üretim bilgi sistemi hazırlanıyor.
- Fabrikada bir çok kimyasal madde üretilmektedir.
- Fabrikada birkaç kimyasal maddenin birleşiminden oluşan çok sayıda karışımlar da hazırlanmaktadır.
- Aynı madde doğal olarak birden fazla karışımda yer alabilmektedir.

- Kaynak: DP Java Workbook p128

132

ÖRNEK KALIP GERÇEKLEMELERİ – FLYWEIGHT

İLK ÇÖZÜM:



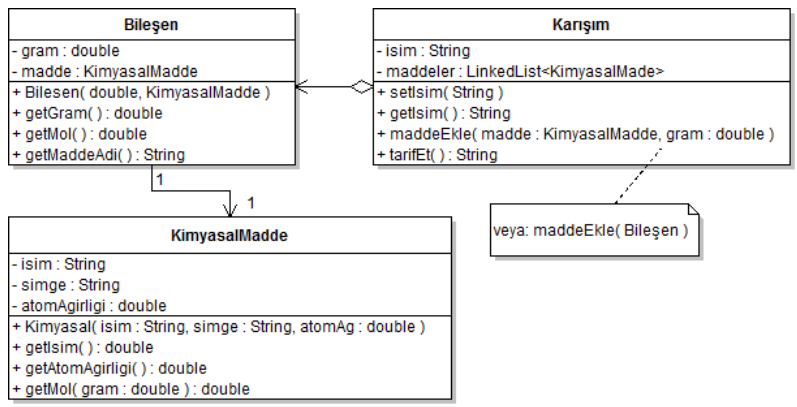
- Örneğin 100gr barutta 75gr potasyum nitrat, 15gr karbon, 10gr kükürt bulunur.
- Yukarıdaki çözüm bu tür bileşikleri destekler ancak örneğin karbon başka gramajla başka karışımlarda da bulunacaktır.
- Sadece gram bilgisi farklı olan, kalan durum bilgisi aynı olan çok sayıda karbon nesnesi mi oluşturulmalıdır?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

133

ÖRNEK KALIP GERÇEKLEMELERİ – FLYWEIGHT

DAHA DOĞRU MODELLEME:

- Gram bilgisinin madde sınıfından ayrıldığı yeni tasarım:

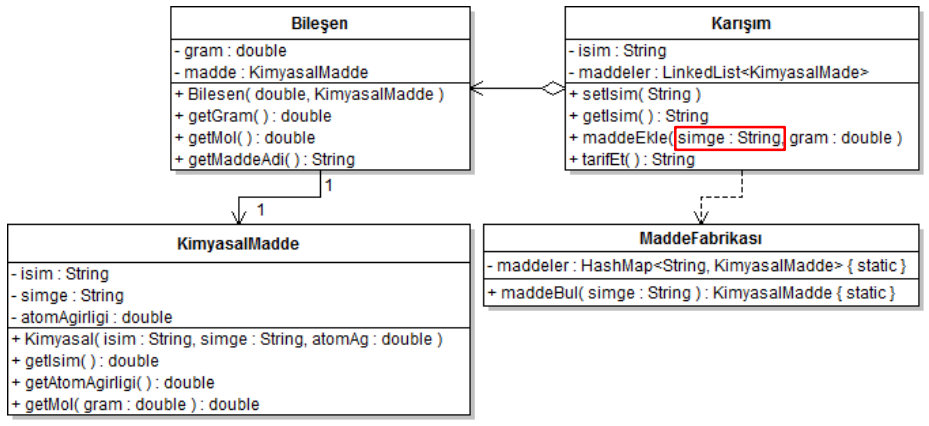


- Bu noktada gram, dışsal (extrinsic) bilgi olarak dikkatimizi çeker.
- Çok sayıda madde nesnesi olacağını da düşünüp Flyweight kalıbına yöneliriz.

134

ÖRNEK KALIP GERÇEKLEMELERİ – FLYWEIGHT

FLYWEIGHT KALIBI İLE ÇÖZÜM



135

ÖRNEK KALIP GERÇEKLEMELERİ – FLYWEIGHT

FLYWEIGHT KALIBI İLE ÇÖZÜM

- Kaynak kodlar:

```
package dp.flyweight.solution3;

public class KimyasalMadde {
   private String isim;
   private String simge;
   private double atomAgirligi;

   public KimyasalMadde(String isim, String simge, double atomAgirligi) {
      this.isim = isim; this.simge = simge;
      this.atomAgirligi = atomAgirligi;
   }
   public String getIsim() { return isim; }
   public String getSimge() { return simge; }
   public double getAtomAgirligi() { return atomAgirligi; }
   public double getMol( double gram ) {
      return gram/atomAgirligi;
   }
}
```

136

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);
   }
}
```

137

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( ); }
}
```

138

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;
   }
}
```

139

ÖRNEK KALIP GERÇEKLEMELERİ – FLYWEIGHT

FLYWEIGHT KALIBI İLE ÇÖZÜM

- Kaynak kodlar:

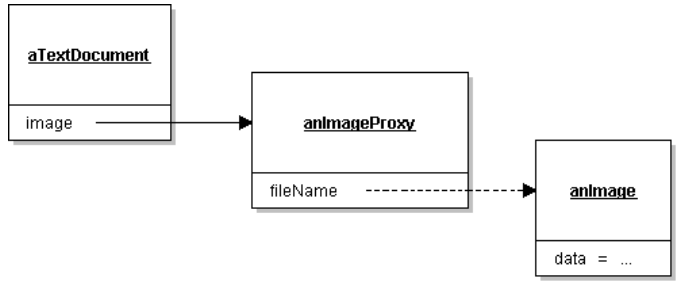
```
package dp.flyweight.solution3;
public class MainApp {
    public static void main(String[] args) {
        Karisim karaBarut = new Karisim();
        karaBarut.setIsim("Kara barut");
        karaBarut.maddeEkle("NaNO3", 75);
        karaBarut.maddeEkle("C", 15);
        karaBarut.maddeEkle("S",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

140

YAPISAL (STRUCTURAL) KALIPLAR

PROXY:

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

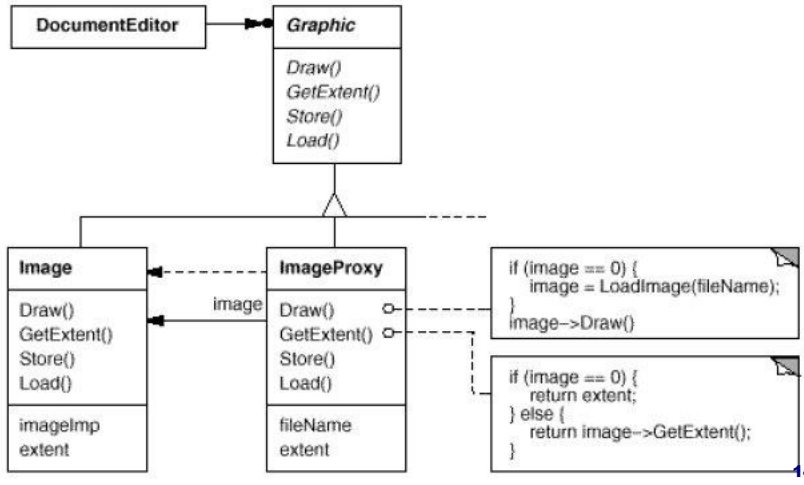


141

YAPISAL (STRUCTURAL) KALIPLAR

PROXY:

- Örnek çözüm:

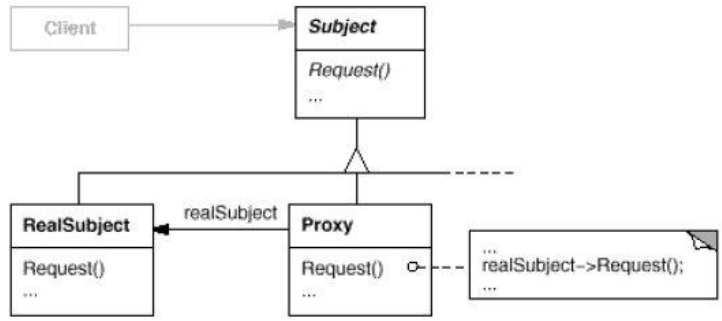


142

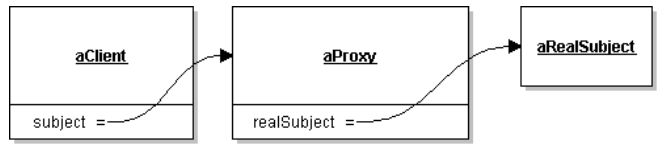
YAPISAL (STRUCTURAL) KALIPLAR

PROXY:

- Kalıp yapısı :



- Bellekteki durum :



143

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 aslının yerine geçebilmesi için, vekil ve asıl nesneler için ortak bir arayüz sunar.
- Kalıbın uygulanabileceği anlar:
  - Herhangi bir nesneye, basit bir işaretçiden daha fazlasının gerektiği her durumda kullanılabilir.
  - Gereken yetenekler proxy sınıfına kodlanır.
  - İşaret edilen nesne tamamen farklı bir adres uzayında bulunabilir.

144

ÖRNEK KALIP GERÇEKLEMELERİ – PROXY

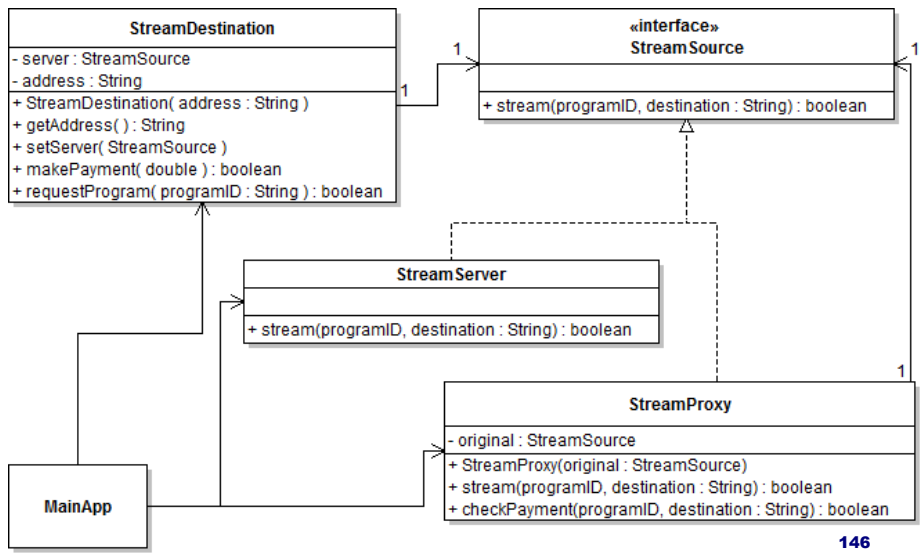
ÇÖZÜLECEK PROBLEM:

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

145

ÖRNEK KALIP GERÇEKLEMELERİ – PROXY

ÇÖZÜM:



146

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;
    }
}
```

147

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);
    }
}
```

148

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

149

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

CHAIN OF RESPONSIBILITY:

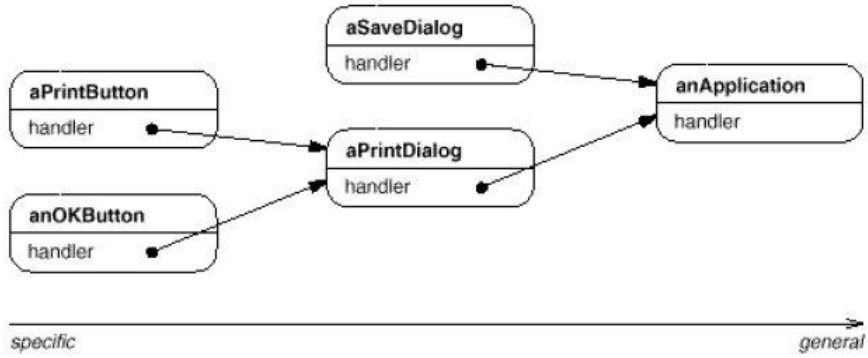
- Amaç:
  - Birbirleri ile ilişkili bir nesneler zinciri veya hiyerarşisindeki bir nesnenin, aldığı bir mesajı işleyemeyeceği durumlarda, bu mesajın gerekli işlemi yapabilecek bir başka nesneye iletmek.
  - Mesajın işlenememesinin nedeni nesnenin arayüzünde o mesajın olmaması değil, nesnenin mesajı işleyecek 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 iletilir.
  - İletim mesaj bir nesne tarafından cevaplanana kadar veya zincirin sonuna gelinceye dek sürer.
  - Bir bileşen, hangi üst bileşen içinde yer aldığını bilecektir.

150

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

CHAIN OF RESPONSIBILITY:

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

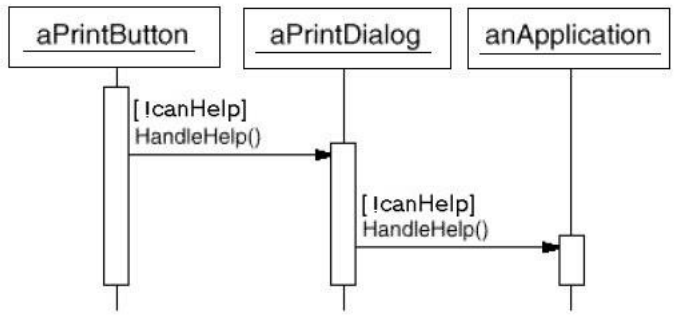


151

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.

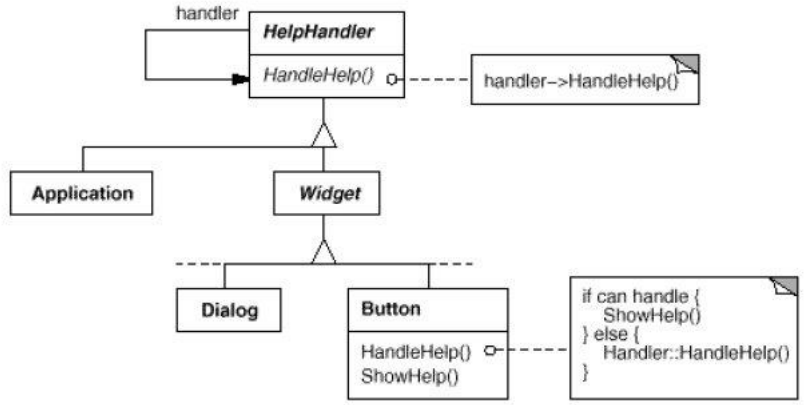


152

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

CHAIN OF RESPONSIBILITY:

- Çözümü oluşturacak sınıf yapısı:

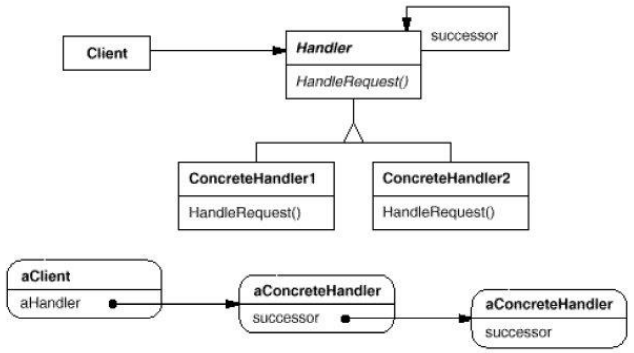


153

DAVRANIŞSAL (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 bağlantılı olduğu bir başkasına iletir.

154

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

CHAIN OF RESPONSIBILITY:

- Kalıbın uygulanabileceği anlar:
  - Bir mesajı yanıtlayabilecek birden fazla nesnenin bulunduğu ve yanıt verecek nesnenin önceden bilinemeyeceği anlarda.
  - Bir mesajı yanıtlayabilecek nesnelerin çalışma anında tanımlanabileceği anlarda.
- Kalıbın zayıf yönleri:
  - Zincir doğru yapılandırılmazsa, bir mesajın yanıtlanacağının garantisi 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 kalıtım hiyerarşisi kurulabilir.

155

ÖRNEK KALIP GERÇEKLEMELERİ – CHAIN OF RESPONSIBILITY

ÇÖZÜLECEK PROBLEM:

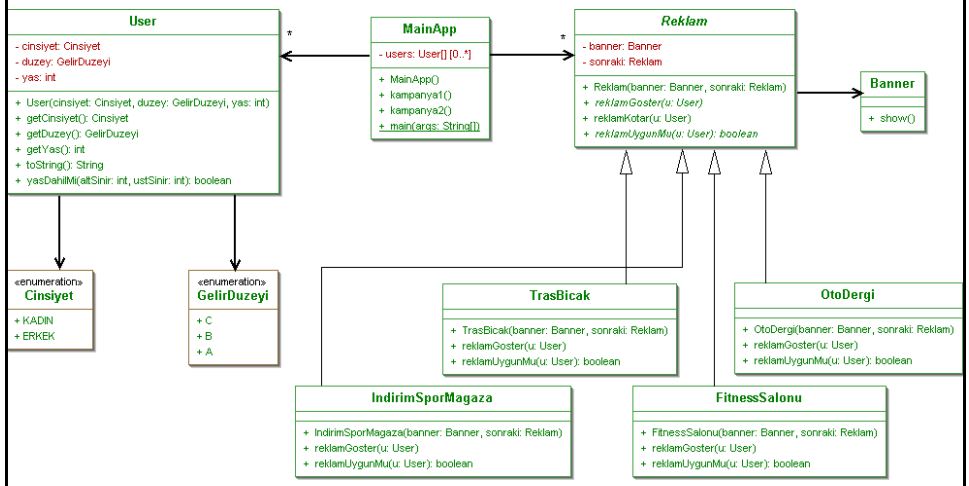
- Bir reklam yönetim yazılımı hazırlanıyor.
  - Farklı web sitelerinde kampanyaların çeşitli kurallarına göre çeşitli reklamlar gösterilmek isteniyor.
  - Kampanya 1:
    - Spor sitesinin 18-39 yaş arası erkek ziyaretçilerine traş bıçağı reklamı göster, 18-39 yaş arası kadın ziyaretçilerine fitness salonu reklamı göster, diğer ziyaretçilere spor mağazasının indirim kampanyasının reklamını göster.
  - Kampanya 2:
    - Haber sitesinin 18-29 yaş arası A gelir grubu erkek ziyaretçilerine 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

156

ÖRNEK KALIP GERÇEKLEMELERİ – CHAIN OF RESPONSIBILITY

ÇÖZÜM:



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

157

ÖRNEK KALIP GERÇEKLEMELERİ – CHAIN OF RESPONSIBILITY

ÇÖZÜM:

- Kaynak kodlar

```
package dp.chainOfResponsibility.example;
public abstract class Reklam {
    private Banner banner;    private Reklam sonraki;
    public Reklam(Banner banner, Reklam sonraki) {
        this.banner = banner; this.sonraki = sonraki;        }
    public abstract boolean reklamUygunMu ( User u );
    public abstract void reklamGoster( User u );
    public final void reklamKotar( User u ) {
        if( reklamUygunMu(u) ) {
            System.out.println( this.getClass().getName() + "->" + u );
            reklamGoster(u);
        }
        else if( sonraki != null ) {
            System.out.println( this.getClass().getName() + "->" +
                sonraki.getClass().getName());
            sonraki.reklamKotar(u);
        }       }
}
```

158

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;
    }
}
```

159

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;
   }
}
```

160

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;
   }
}
```

161

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.

162

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 kampanya1( ) {
        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
```

163

ÖRNEK KALIP GERÇEKLEMELERİ – CHAIN OF RESPONSIBILITY

ÇÖZÜM:

- Kaynak kodlar (devam)

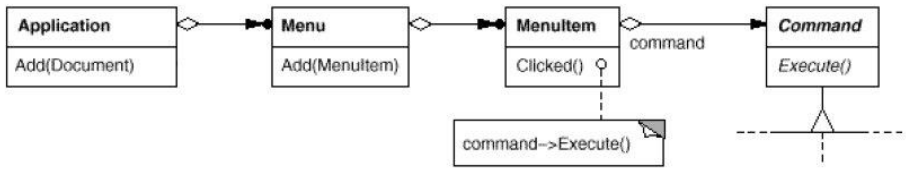
```
    public void kampanya2( ) {
        Reklam[] reklamlar = new Reklam[4];
        reklamlar[3] = new IndirimSporMagaza(new Banner(), null);
        reklamlar[2] = new FitnessSalonu(new Banner(), reklamlar[3]);
        reklamlar[1] = new TrasBicak(new Banner(), reklamlar[2]);
        reklamlar[0] = 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.kampanya1();
        app.kampanya2();
    }
} //end class
```

164

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

COMMAND:

- Amaç:
  - Nesnelere giden istekleri de birer nesne haline çevirmek.
  - Bir başka deyişle: Metotları nesne yapmak!
  - Böylece mesajları kuyruklamak 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 çalıştırmak
- Çözüm: 'Herhangi bir nesnenin herhangi bir metodu' yerine, belli bir arayüzün belli bir metodu çalıştırılır.

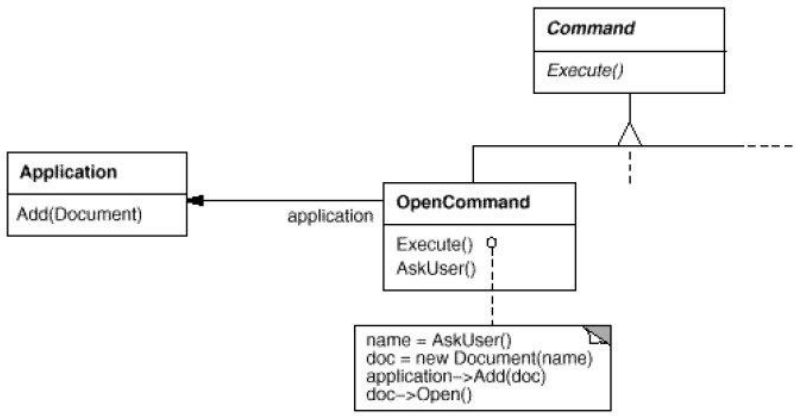


165

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

COMMAND:

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

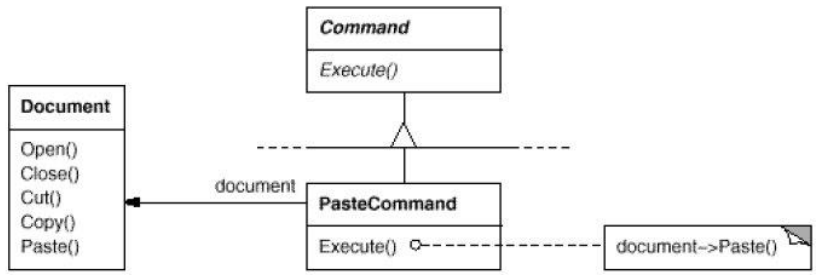


166

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

COMMAND:

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

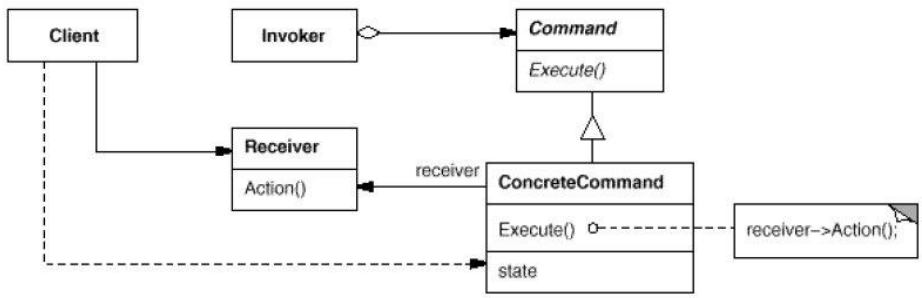


167

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

COMMAND:

- Kalıp yapısı:



168

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

COMMAND:

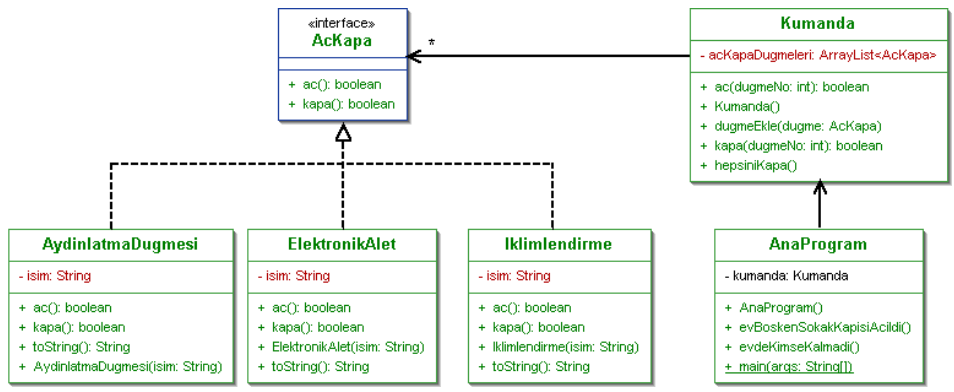
- Kalıp bileşenleri:
  - Command: Komut tanımlama arayüzü
  - ConcreteCommand: Komut gerçeklemeleri. Komutlar belli bir alıcının eylemlerini kullanacak şekilde gerçeklenebilir.
  - Client: Bir komut gerçeklemesi nesnesi oluşturur ve komutun alıcısını belirler.
  - Receiver: Alıcı nesne gerçeklemesi.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

169

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

170

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; }
}
```

171

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();
    }
}
```

172

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 çıkılıyor");
        prg.evdeKimseKalmadi();
    }
}
```

173

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

ITERATOR:

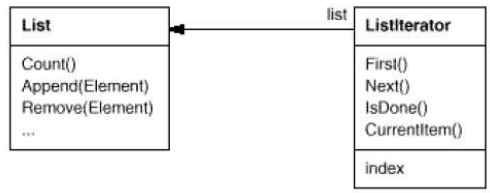
- Amaç:
  - Parçalardan oluşan bir bileşenin parçalarını,
  - bileşenlerin iç yapısını açığa çıkarmadan,
  - istenilen bir biçimde dolaşmak.
- Örnek:
  - Bir veri yapısı içerisinde dolaşmak.
- Sorun:
  - Veri yapısının arayüzünü birçok farklı dolaşım türü ile şişirmek istemeyiz.
- Not: Java'da herhangi 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.

174

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

ITERATOR:

- 1. çözüm:



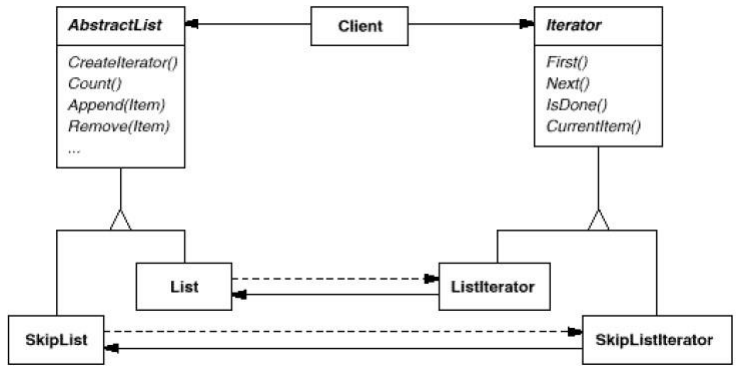
- Çözümün güçlü yönü:
  - Veri yapısı ile dolaşım biçimi birbirinden soyutlanmıştır.
- Çözümün zayıf yönü:
  - Belli bir veri yapısı ile belli bir dolaşım biçimi arasında bağlılık oluşmuştur.
  - Bu bağlılığı azaltmak üzere genel bir liste yapısı arayüzü ve genel bir dolaşım biçimi arayüzü oluşturulabilir.

175

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

ITERATOR:

- 2. çözüm:

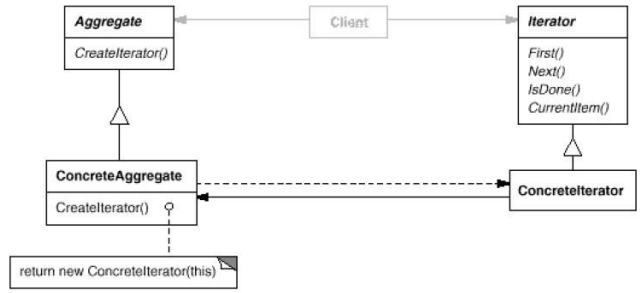


176

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

ITERATOR:

- Kalıp yapısı:



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

177

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.

178

ÖRNEK KALIP GERÇEKLEMELERİ – ITERATOR

ÇÖZÜLECEK PROBLEM:

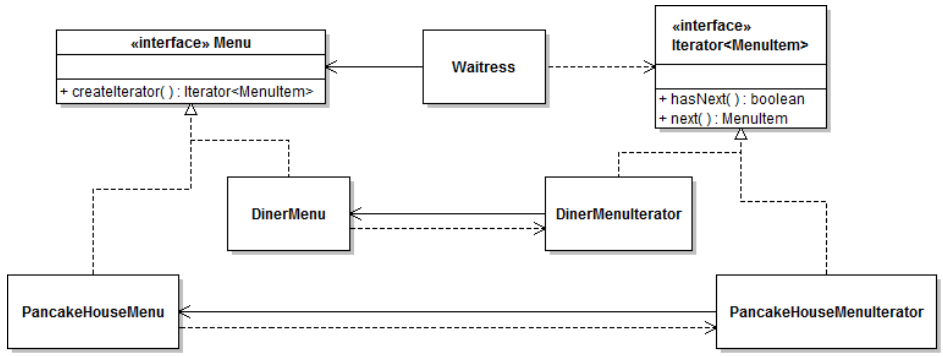
- Online yemek siparişi verebileceğiniz bir firma hızla kurulmak isteniyor.
- Bu amaçla biri kahvaltılıkta, biri yemekte uzman iki firma satın alınıyor.
- Menü elemanları gerçeklemesi aynı denk gelmiş ama her firmanın kendi ayrı yemek menüsü gerçeklemesi var.
- Farklı menü gerçeklemelerinden yola çıkarak, 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

179

ÖRNEK KALIP GERÇEKLEMELERİ – ITERATOR

ITERATOR KALIBI İLE ÇÖZÜM:



180

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());
    }
```

181

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);
                }
        }
    }
}
```

182

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 kalıbını kullanan iyileştirilmiş tasarımdan gelen ek*/
    public Iterator<MenuItem> createIterator() {
        return new DinerMenuIterator(menuItems);
    }
}
```

183

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 kalıbını kullanan iyileştirilmiş tasarımdan gelen ek*/
    public Iterator<MenuItem> createIterator() {
        return new PancakeHouseMenuIterator(menuItems);
    }
}
```

184

ÖRNEK KALIP GERÇEKLEMELERİ – ITERATOR

ITERATOR KALIBI İLE ÇÖZÜM:

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

185

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

MEMENTO:

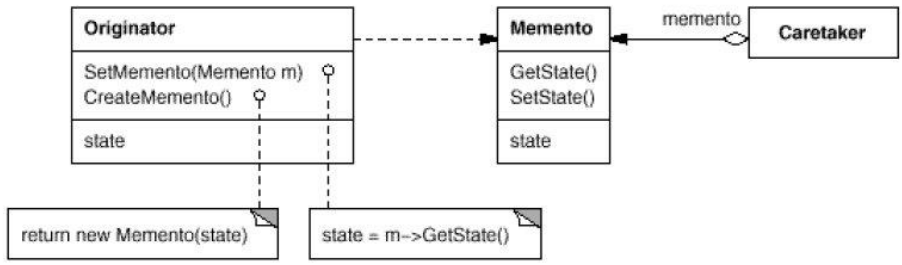
- Amaç:
  - Bir nesnenin durum bilgisini sarmalama (encapsulation) ilkesini bozmadan elde etmek.
  - Böylece nesnelerin ikincil saklama ortamlarında saklanması, geri yüklenmesi, ağ üzerinden aktarımı mümkün olabilir.
  - Yine bu şekilde sona yapılan işlemi geri alma düzeneği (undo) kurulabilir.
  - Durum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

186

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

MEMENTO:

- Kalıp yapısı:



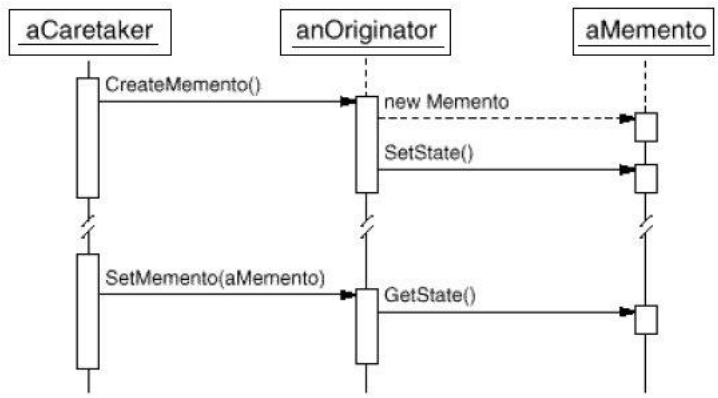
- Kalıp bileşenleri:
  - Originator: Durum bilgisi saklanacak olan nesne
  - Memento: Durum bilgisinin gerekli kısımlarını saklar
  - Caretaker: Gerekli anlarda durum bilgisinin bir kopyasını alır ve elindeki kopyalardan birini geçerli durum haline getirir.

187

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

MEMENTO:

- Kalıp bileşenlerinin etkileşimleri:



188

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

MEMENTO:

- Kalıbın uygulanabileceği anlar:
  - Nesnelerin durum bilgisinin tümü veya bir kısmının daha sonra o duruma dönebilmek üzere saklanması VE
  - Nesneye doğrudan bir işaretçinin veri gizliliği ilkesini bozmasının istenmediği anlarda.
- Sorun:
  - Veri gizliliği ilkesini sağlamak için Memento'nun metotlarına sadece Originator erişebilmelidir.
  - Bu gereksinim her dilde karşılanamaz.
    - C++: Memento metotları private yapılır ve Originator sınıfı Memento sınıfının arkadaşı (friend) olarak tanımlanır.
    - Java: Memento metotları 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 metotları konulmaz.
    - Java diğer seçenek: Memento, Originator sınıfının iç sınıfı yapılır.
      - Ancak bu durumda State clonable yapılmalı ve Originator. createMemento metodu klon döndürmelidir.

189

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

MEMENTO:

- Dış sınıf ile çözüm: Önceki yansıda 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;
   }
}
```

190

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; }
   }
}
```

191

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

MEMENTO:

- İç sınıf ile çözüm (devam):

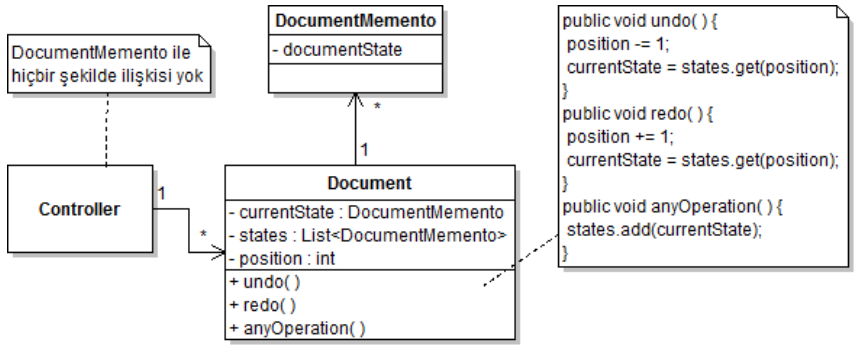
```
package dp.mementoV2Outer;
import dp.mementoV2.innerClass.*;
import dp.mementoV2.innerClass.Originator.Memento;

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();
   }
}
```

192

ÖRNEK KALIP GERÇEKLEMELERİ – MEMENTO

ÇÖZÜLECEK PROBLEM:

- Bir belge işleme programında geri alma ve yineleme işlemleri

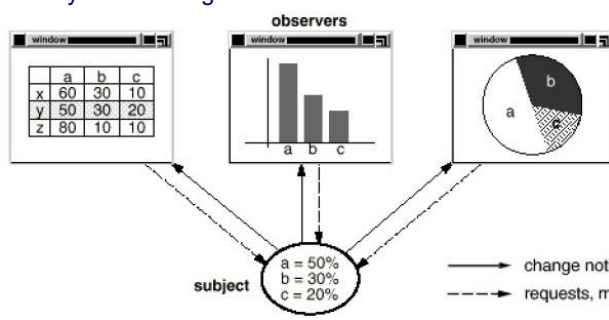


193

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 anında kurulması gereken nesnelerde bu iş nasıl yapılabilir?
- Örnek:
  - Bir hesap tablosundaki bilgi değiştiğinde çeşitli çizelgelere de son durumun anında yansıtılması gerekebilir.

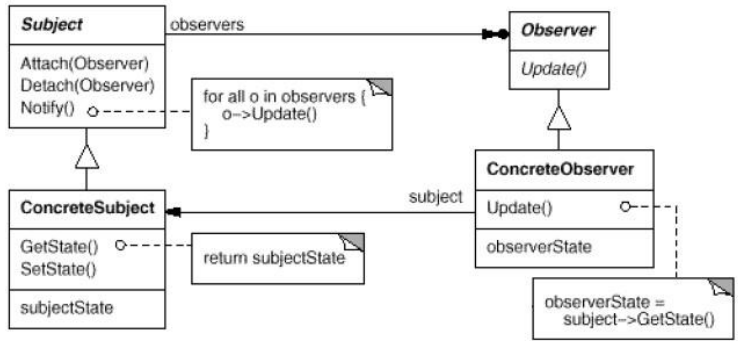


194

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

OBSERVER:

- Kalıp yapısı:



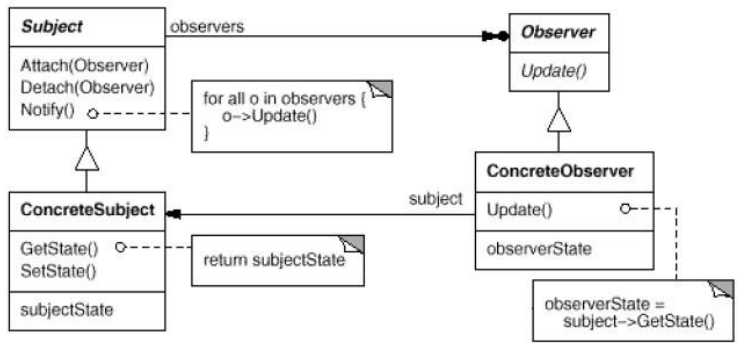
- Kalıp bileşenleri:
  - Subject: Durumundaki değişikliklerin izleneceği nesnenin arayüzü.
    - Birden fazla gözleyicisi olabilir. Bu nedenle gözleyici ekle/sil metotlarını tanımlayan soyut bir sınıf olmalıdır.
  - Observer: Gözleyici nesnelerin arayüzü.

195

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

OBSERVER:

- Kalıp yapısı:



- Kalıp bileşenleri:
  - ConcreteSubject: Subject gerçeklemeleri. Durumu değiştiğinde gözleyicilerini Observer arayüzünü kullanarak bilgilendirir.
  - ConcreteObserver: Observer gerçeklemeleri. Gözlediği nesneyi gösteren bir işaretçi tutabilir.

196

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.

197

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

OBSERVER:

- Gerçekleme ayrıntıları (devam):
  - Bir gözleyicinin birden fazla nesneyi gözlemesi istenebilir.
  - Gözlenen nesnenin durumundaki farklı türden değişiklikleri 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.
- Kalıbın kullanılabileceği anlar:
  - Herhangi bir soyutlamanın birbirine bağımlı birden fazla parçası olduğunda.
  - Bir nesnedeki değişikliklerin derleme anında bilinmeyen sayıda nesneyi de etkilemesi gereken anlarda.
  - Bağlaşımı arttırmadan bir nesnenin diğer nesneleri herhangi bir durumdan haberdar etmesi istenen anlarda.

198

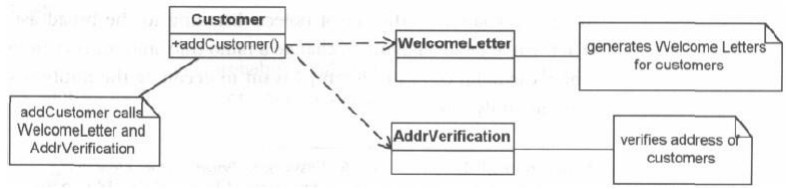
ÖRNEK KALIP GERÇEKLEMELERİ – OBSERVER

ÇÖZÜLECEK PROBLEM:

- Yazdığımız bir müşteri ilişkileri programına yeni müşteri eklendiğinde, müşteriye 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?

199

ÖRNEK KALIP GERÇEKLEMELERİ – OBSERVER

ÇÖZÜM 2:

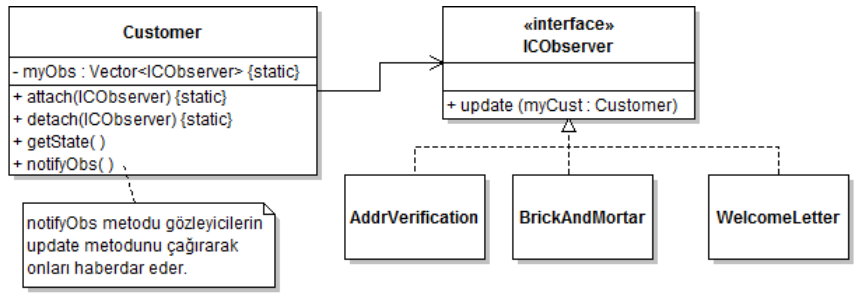
- Yeni bir gereksinim ortaya çıkıyor:
  - Müşteri bir şubemizin 10km. yakınında ise ona gönderilecek mektupta indirim kuponları da olsun.
  - Benzer başka gereksinimlerin de ortaya çıkabileceğini görüp, şimdiden önlemimizi alalım.
  - Sisteme her yeni müşteri eklendiğinde, bir veya birkaç farklı işlem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

200

ÖRNEK KALIP GERÇEKLEMELERİ – OBSERVER

ÇÖZÜM 2:

- Observer kalıbının kullanılması:



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

201

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
}
```

202

ÖRNEK KALIP GERÇEKLEMELERİ – OBSERVER

ÇÖZÜM 2:

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

203

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

STRATEGY:

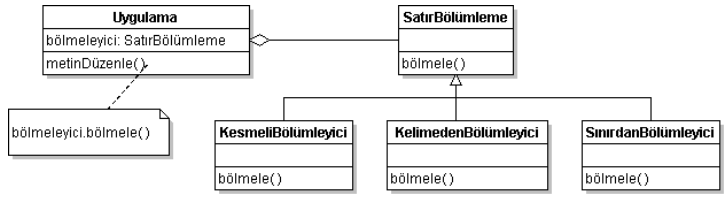
- Amaç:
  - Aynı iş için olan farklı algoritmalar ile bu işin yapılmasını isteyecek istemcileri birbirlerinden soyutlamak.
  - Böylece kullanılan algoritmayı çalışma anında değiştirilebilir.
- Örnek:
  - Metinlerin gösteriminde kullanılabilecek alt satıra geçme (line breaking) algoritmaları çok çeşitlidir.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

204

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

STRATEGY:

- Örnek çözüm:

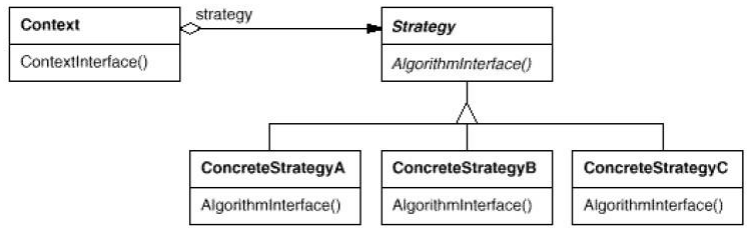


205

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

STRATEGY:

- Kalıp yapısı:



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

206

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

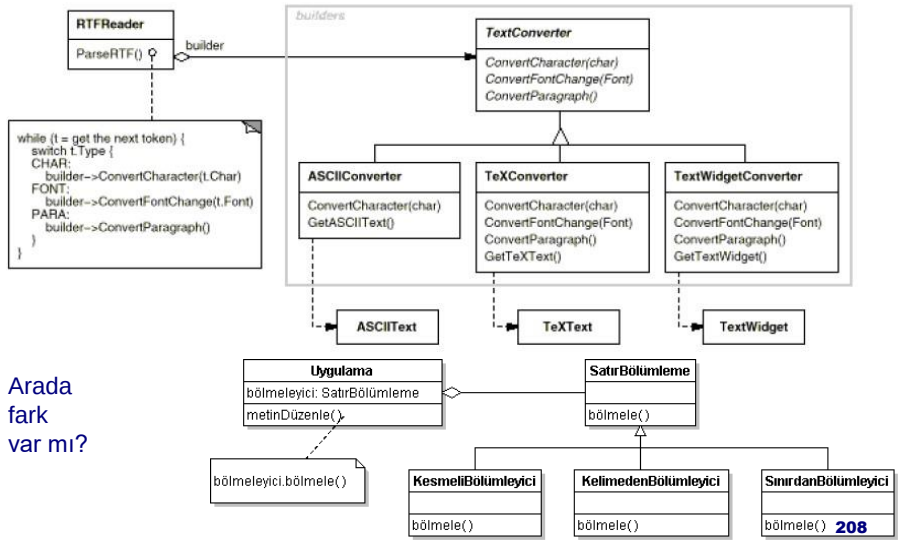
STRATEGY:

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

207

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

STRATEGY ve BUILDER KALIPLARI:



Arada fark var mı?

208

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

STRATEGY ve BUILDER KALIPLARI:

- Tasarım kalıpları benzer sınıf yapılarına sahip olabilir.
- Kalıpları farklılaştıran 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.

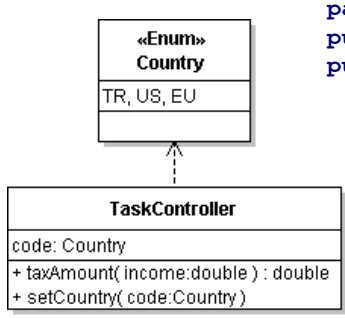
209

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



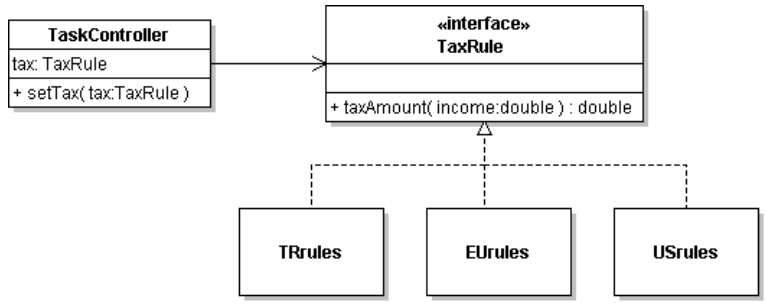

```
package strategy.solution1;
public enum Country { TR, US, EU }
public class TaskController {
    private Country code;
    public void setCountry(Country code){
        this.code = code;
    }
    public double taxAmount(double income){
        switch (code) {
        case TR: return income * 0.18;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

- Kaynak: DP Explained

210

ÖRNEK KALIP GERÇEKLEMELERİ – STRATEGY

ÇÖZÜM 2: Strategy Tasarım Kalıbı ile



211

ÖRNEK KALIP GERÇEKLEMELERİ – STRATEGY

ÇÖZÜM 2: Strategy Tasarım Kalıbı ile

```
package strategy.solution2;
public interface TaxRule {
    public double taxAmount( double income );
}
public class TaskController {
    private TaxRule tax;
    public void setTax(TaxRule tax) { this.tax = tax; }
    public double taxAmount( double income ) {
        return tax.taxAmount( income );
    }
}
public class TRrules implements TaxRule {
    public double taxAmount(double income) {
        return income * 0.18;
    }
}
```

212

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;
    }
}
```

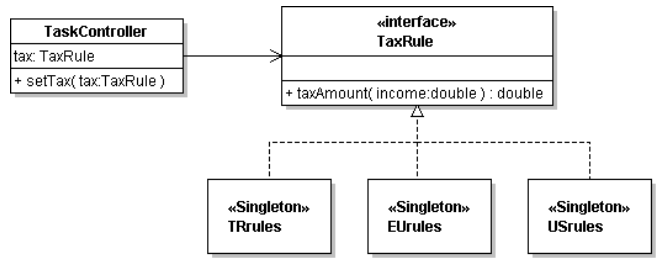
- Kalıp asıl esnekliğini vergi hesaplamanın daha karmaşık kurallara dayandığı zaman gösterir ki, gerçek hayatta da durum öyledir.
- Diğer mali işlemler için Strategy kalıbı tekrarlanır.

213

ÖRNEK KALIP GERÇEKLEMELERİ – SINGLETON

SINGLETON ve STRATEGY BİRLİKTELİĞİ:

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



214

ÖRNEK KALIP GERÇEKLEMELERİ – SINGLETON

ÇÖZÜM:

```
package singleton.example;
public class TRrules implements TaxRule {
    private static TRrules instance;
    private TRrules( ) {
        //gerekli ilklendirmeler
    }
    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.

215

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

STATE:

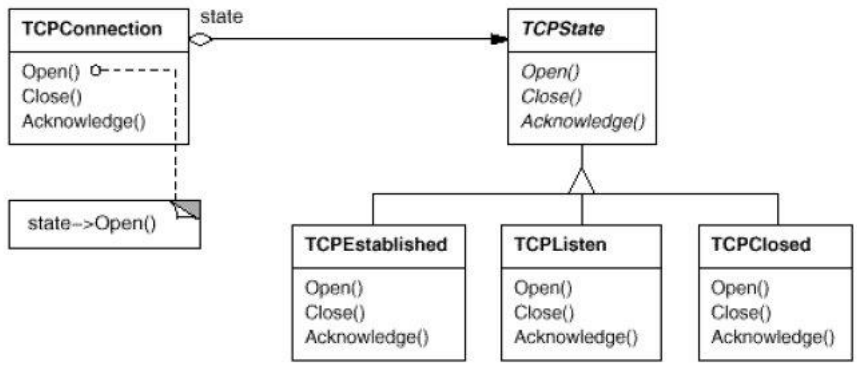
- Amaç:
  - Bir nesnenin iç durumu değiştiğinde davranışının da değişmesini sağlamak.
  - Bu değişiklik büyük ölçekli olmalıdır.
    - Küçük ölçekli: Bir üye alanın değerinin değişmesi yüzünden bir metodun hesapladığı sonucun değişmesi
    - Büyük ölçekli: Nesnenin önceki çalışma biçimini tamamen değiştirmesi.
- 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!

216

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

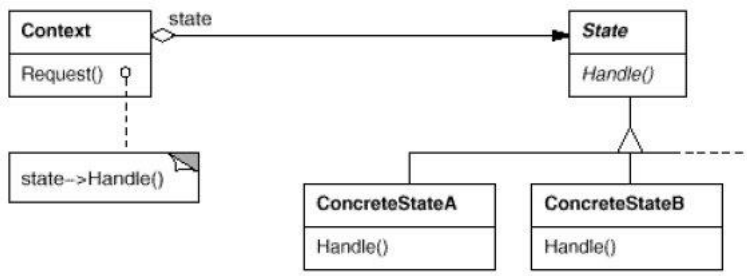


217

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?

218

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

STATE:

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

STATE ve STRATEGY KALIPLARININ FARKI:

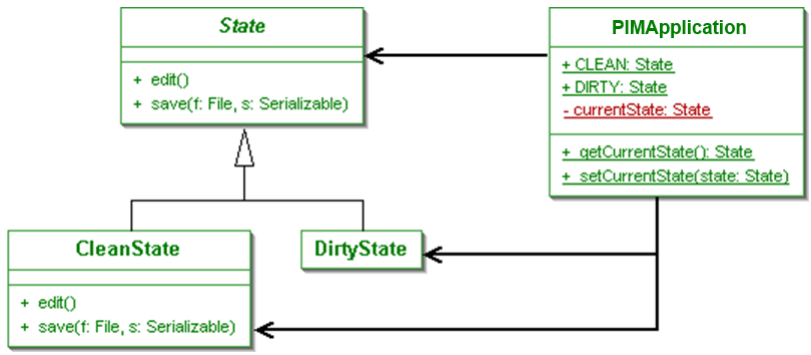
- State:
  - Context'teki durum değiştikçe, Context kendi ConcreteState'ini seçer.
  - State geçişleri açıkça yapılır (Context'te verilen komutlarla veya ConcreteState'ler içerisinden)
  - Durum geçişleri daha sıktır.
- Strategy:
  - Strateji geçişleri seyrektir.
  - Strateji nesneleri Context hakkında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

219

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

220

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*/ }
}
```

221

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 { }
```

222

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;
    }
}
```

223

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

TEMPLATE METHOD:

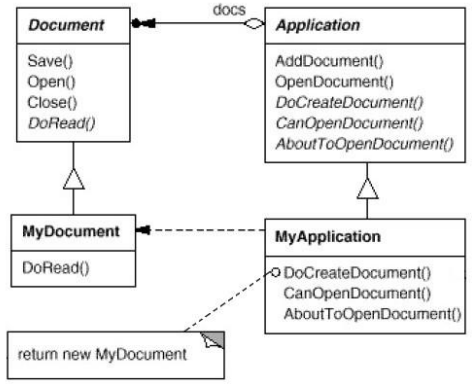
- Amaç: Bir algoritmanın bazı adımlarını belirleyip bazılarının ayrıntılarının belirlenmesini ertelemek.
- Örnek:
  - Farklı belge türleri ile çalışabilecek uygulamalar geliştirmeyi sağlayan bir çerçeve program hazırlanıyor.
  - Kullanıcı çerçevenin sunduğu Belge ve Uygulama sınıflarından kalıtım ile 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.

224

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

TEMPLATE METHOD:

- Örnek çözüm:

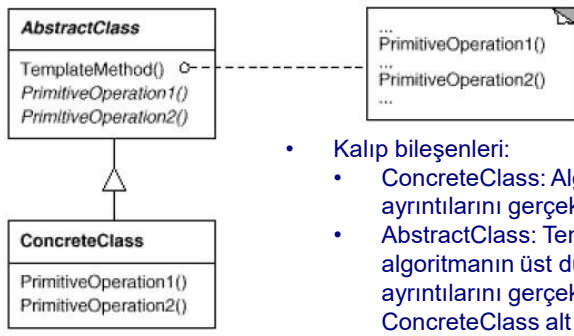



```
void Application::OpenDocument (const
     char* name) {
   if (!CanOpenDocument(name)) {
     // cannot handle this document
     return;
   }
   Document* doc = DoCreateDocument();
   if (doc) {
     _docs->AddDocument(doc);
     AboutToOpenDocument(doc);
     doc->Open();
     doc->DoRead();
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


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

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

TEMPLATE METHOD:

- Kalıbın uygulanabileceği anlar:
  - Bir algoritmanın değişmez kısımlarını bir kere kodlayıp değişebilecek kısımlarını alt sınıflar üzerinden gerçeklemek istenildiğinde
  - Alt sınıfların davranışlarının denetlenmesi istenildiğinde
  - Tutarsızlığı veya kod tekrarını önlemek üzere kardeş sınıfların ortak davranışlarını bir noktada (üst sınıfta) toplamak istenildiğinde

227

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 faklılık 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 kalıbının kullanımını daha doğru bir seçim kılar.

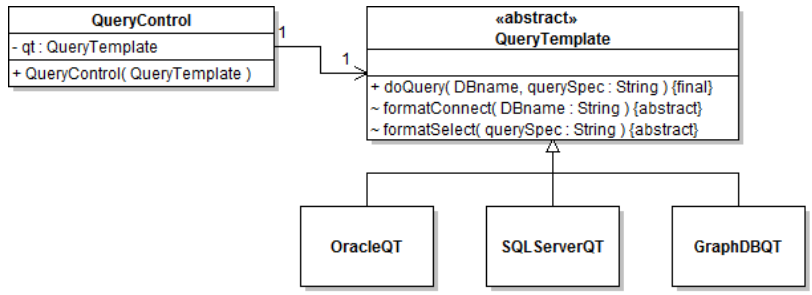
- Kaynak: DP Explained

228

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 görünürlüğü kullanılabilir.



229

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 bağlantıyı kur.
        dbCommand = formatSelect( querySpec );
        //dbCommand komutunu yürüterek kayıtları oku.
        //Oluşan veri kümesini geri döndür.
    }
    abstract String formatConnect( String DBname );
    abstract String formatSelect( String querySpec );
}
```

230

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; }
}
```

231

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", "*");
    }
}
```

232

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

VISITOR:

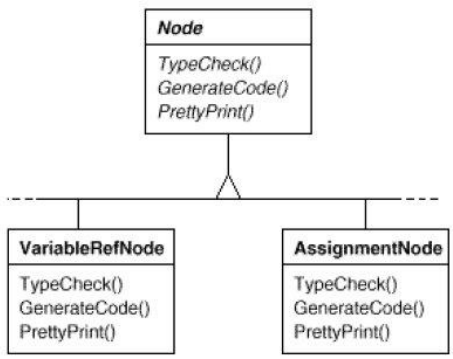
- Amaç:
  - Herhangi bir nesne yapısını oluşturan elemanlar üzerinde yapılacak bazı işlemler tanımlamak.
- Sorun:
  - Nesne yapısı her türden bağıntılar içeren sınıfların örnekleri arasında kurulmuş olabilir.
  - Üzerinde işlem yapılan elemanların sınıf yapılarını değiştirmek istemiyoruz.
- Eleştiri:
  - 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 ayırıyor.
  - Bu parçalar (Node) farklı tiplerden oluşmaktadır.
  - Derleyici bütün parçalar üzerinde çeşitli işlemler yapacaktır.
  - Farklı tip parçalarda işlemler farklı yürütülecektir.

233

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

VISITOR:

- Örnek parça yapısı:



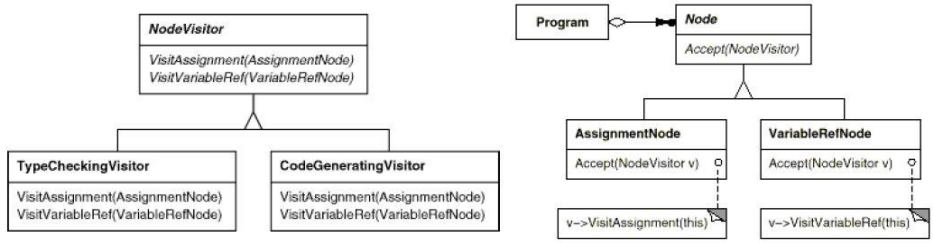
- Bu işlemlerin çeşitli parça yapılarına dağıtılması, bir bakım kabusu oluşturur.
- Her tür düğümde her tür işlemin yapılmasının da anlamı yoktur.

234

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

VISITOR:

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



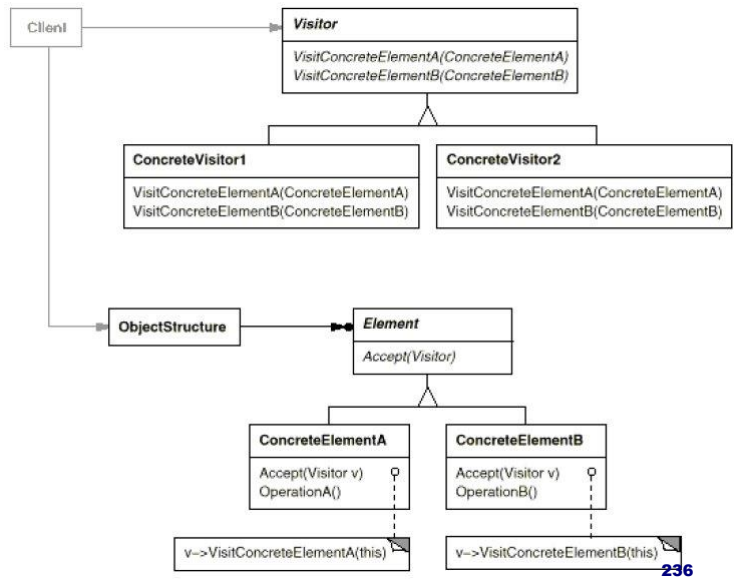
- Düğüm gerçeklemeleri kendi üzerlerindeki gerekli işlemleri çağıracak ziyaretçi gerçeklemelerini kabul eder (Accept metodu ile).
- Bu şekilde parçalarda sadece gerekli işlemler yürütülür.
- Çözümün zayıf yönü: Düğümler ziyaretçinin hangi metodunun bu düğüm türü için olduğunu bilmelidir.

235

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

VISITOR:

- Kalıp yapısı:



236

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

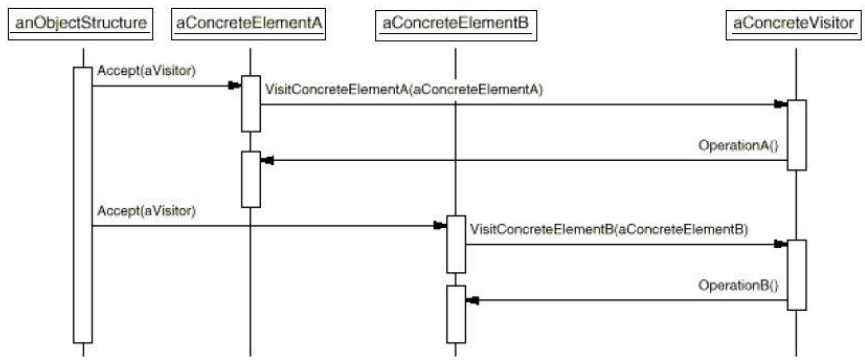
VISITOR:

- 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 yapılırken 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. 237

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

VISITOR:

- Kalıp etkileşimleri:



- Kalıbın kullanılabileceği anlar:
  - Değişen arayüzlere sahip nesnelerden oluşan bir yapıda, nesneler üzerinde farklı işlemler yapılması gerektiği anlarda VE
  - Bu nesne yapısındaki türlerin değişmediği/ender değiştiği koşullarda. 238

ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

ÇÖZÜLECEK PROBLEM – Çözüm 1 :

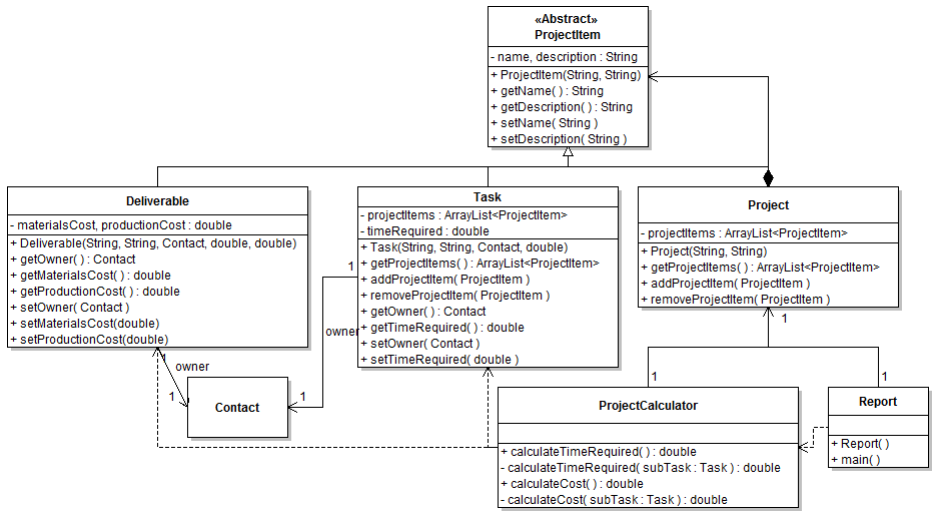
- Bir proje yönetim yazılımı hazırlıyoruz.
- Aşağıdaki temel varlıkları modelimize alıyoruz:
  - Project: Proje hiyerarşisinin kökü
  - Task: Projede tanımlı bir görev
  - DependentTask: Tamamlanabilmesi için başka görevlere bağlı bir görev
  - Deliverable: Projenin ara çıktıları veya sonucu olarak üretilebilecek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

- Kaynak: Applied Java Patterns

239

ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

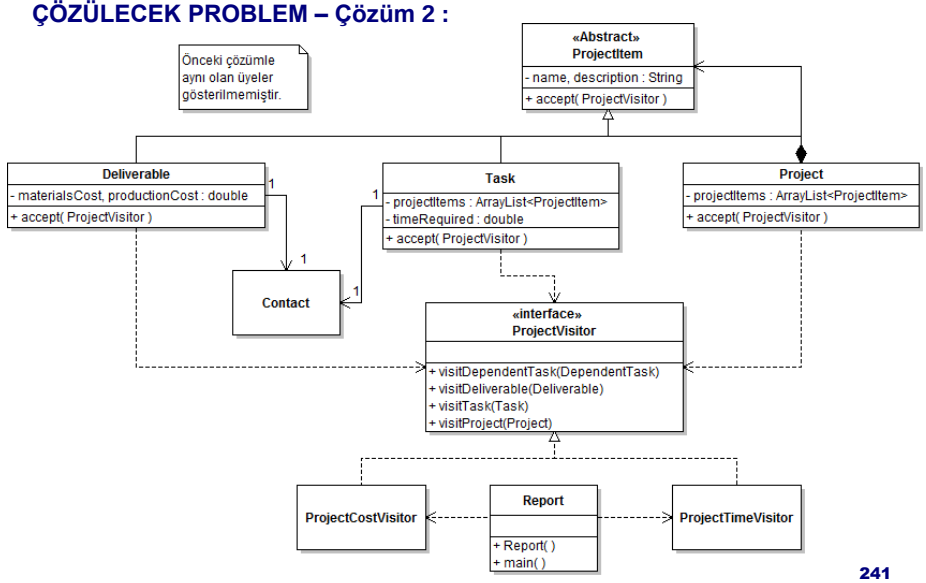
ÇÖZÜLECEK PROBLEM – Çözüm 1 :



240

ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

ÇÖZÜLECEK PROBLEM – Çözüm 2 :



241

ÖRNEK KALIP GERÇEKLEMELERİ – VISITOR

ÇÖZÜLECEK PROBLEM – Çözüm 1 :

- Kaynak kodlar (uygulama alanı):

```
package dp.visitor.solution1;
public abstract class ProjectItem{
   private String name;
   private String description;

   public ProjectItem(String newName, String newDescription){
          name = newName;
          description = newDescription;
   }
   public String getName(){ return name; }
   public String getDescription(){ return description; }
   public void setName(String newName){ name = newName; }
   public void setDescription(String newDescription){ description = newDescription; }
}
```

242

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);
   }
}
```

243

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);
   }
}
```

244

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; }
}
```

245

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.");
   }
}
```

246

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;
   }
```

247

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;
   }
}
```

248

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); }
}
```

249

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);
}
```

250

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;
   }
}
```

251

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;
   }
}
```

252

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.");
   }
}
```

253

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

MEDIATOR:

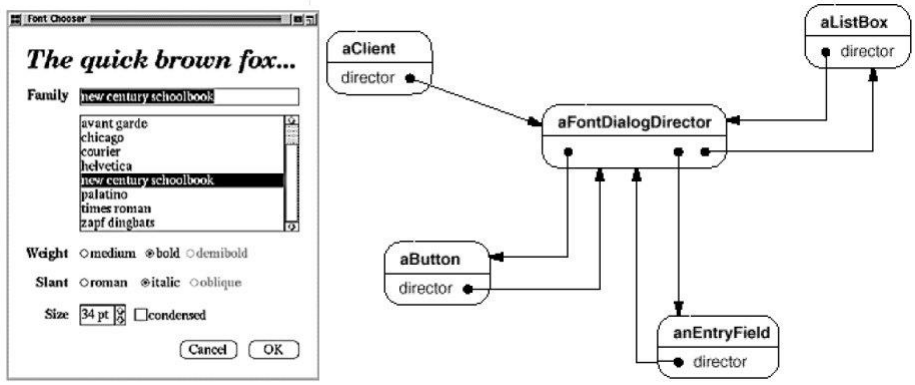
- Amaç:
  - Bir nesneler kümesinin aralarındaki etkileşimlerin sarmalanması.
  - Böylece nesneler birbirleri ile doğrudan değil de dolaylı olarak ilişki kurarlar.
  - Sonuç olarak söz konusu nesnelerin arasındaki ilişkiler bu nesneleri değiştirmeden de değiştirilebilir.
- Örnek:
  - Çeşitli GUI bileşenleri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

254

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

MEDIATOR:

- Örnek:

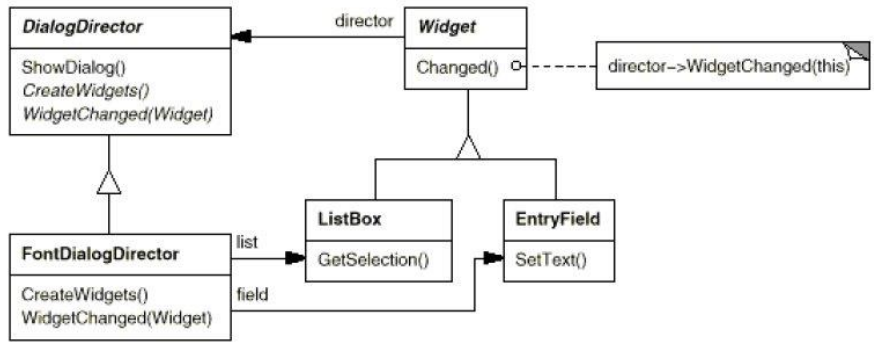


255

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

MEDIATOR:

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

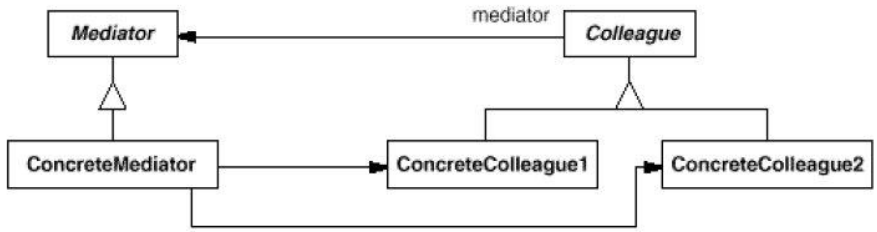


256

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

MEDIATOR:

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



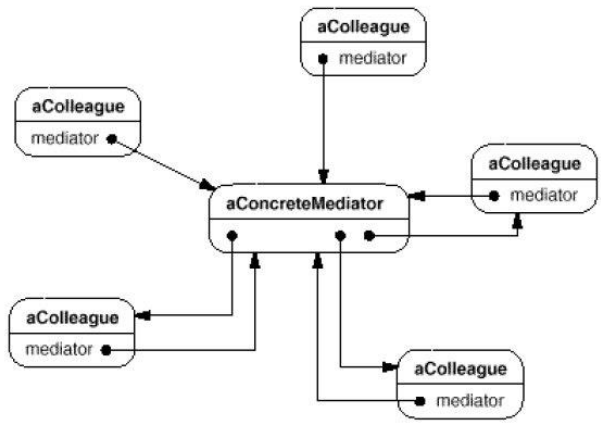
- Kalıp bileşenleri:
  - Mediator: Colleauge nesneleri ile etkileşim kurabilecek bir arayüz sağlar.
  - ConcreteMediator: Colleauge nesneleri arasında eşgüdüm sağlar ve bunların idamesinden sorumludur (sahiplik ilişkisi).
  - Colleauge/ConcreteColleauge: Etkileşimde bulunacak nesnelerin arayüzü ve gerçeklemeleri.

257

DAVRANIŞSAL (BEHAVIORAL) KALIPLAR

MEDIATOR:

- Nesneler arasındaki etkileşimler:



258

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 davranışı tek bir sınıfta toplamak istendiği, ancak bu davranışın fazla değişim göstermeyeceği durumlarda.

259

ÖRNEK KALIP GERÇEKLEMELERİ – MEDIATOR

ÇÖZÜLECEK PROBLEM:

- Bu kalıp için örnek yapılmayacaktır.

- Bu yansı ders notlarının düzeni için boş bırakılmıştır.

260

NESNEYE DAYALI TASARIM VE MODELLEME
KISIM 2: REFACTORING

261

REFACTORING (YENİDEN YAPILANDIRMA)

REFACTORING NEDİR?

- Bir kod parçasının ne yaptığını değiştirmeden nasıl yaptığını değiştirmeye refactoring adı verilmiştir.
- Yazılımın dışarıdan görünen davranışı değiştirilmeden 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.

262

REFACTORING (YENİDEN YAPILANDIRMA)

REFACTORING NASIL YAPILIR?

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

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!

263

CODE SMELLS (KÖTÜ KOKAN / KUSURLU KOD)

CODE SMELL NEDİR?

- Yazılan kodu iyileştirme gerektiğinin 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 yapılır.
  - 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.

264

JUNIT İLE BİRİM SINAMALARI

BİRİM SINAMALARI (Unit Testing)

- En küçük yazılım bileşeninin sınanmasıdır.
  - NYP'de bireysel sınıfların sınanmasıdır.
- Ne zaman tasarlanır?
  - Kodlamadan önce (TDD), kodlama sırasında veya kodlamanın 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

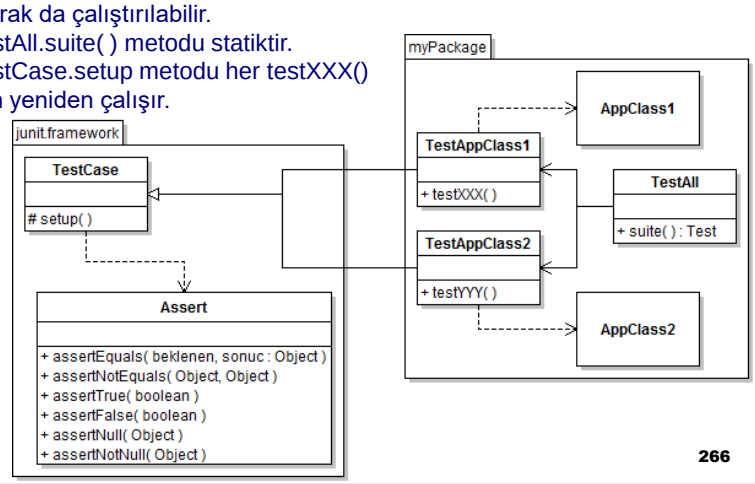
- Java ile yazılmış kodun birim sınamaları için bir çerçeve programdır (framework)
- Diğer diller için de sürümleri bulunmaktadır.
  - Ör. C# için csUnit
- IDE desteği:
  - Eclipse: IDE ile hazır geliyor (build path → add libraries)
  - NetBeans: IDE ile hazır geliyor

265

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.



266

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 sınama sınıflarının TestCase sınıfından kalıtımla türetilmesi gerekmemektedir ancak şu eklemeler yapılmalıdır:

```
import static org.junit.Assert.*;
import org.junit.*;
```

  - Artık sınama metotlarının adlarını test kelimesi ile başlatmak gerekli değildir, ilgili metotların başına @Test annotation koymak yeterlidir.
    - setup adlı metodun yerine ise @Before annotation gelmiştir.

```
@Before
public void setUp() { /*Preparations*/ }
@Test
public void testSomething() { /*Do test*/ }
```

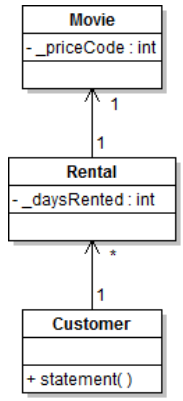
  - Exception tanımlanan yazılımlarda atılması gereken exception'ların gerçekten ortaya çıkıp çıkmadığının sınanması da mümkün olmuştur.

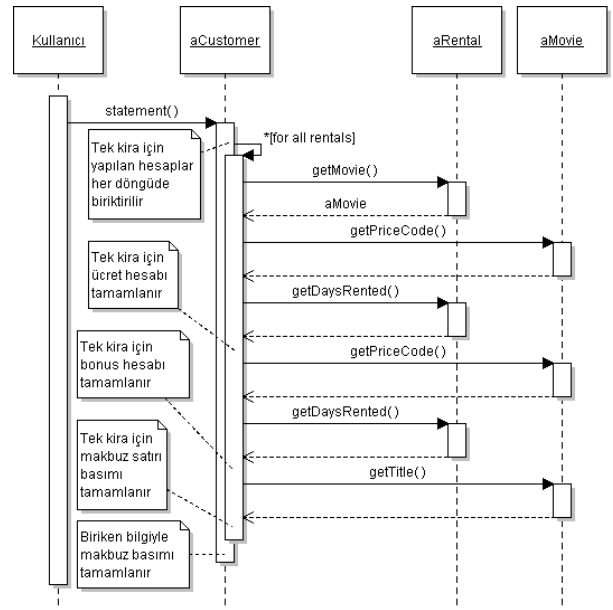
```
@Test(expected=SomeException.class)
public void testTheException() throws SomeException {
    doSomethingThatCreatesTheException();
}
```

267

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);
   }
```

269

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);
   }
```

270

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);
    }
}
```

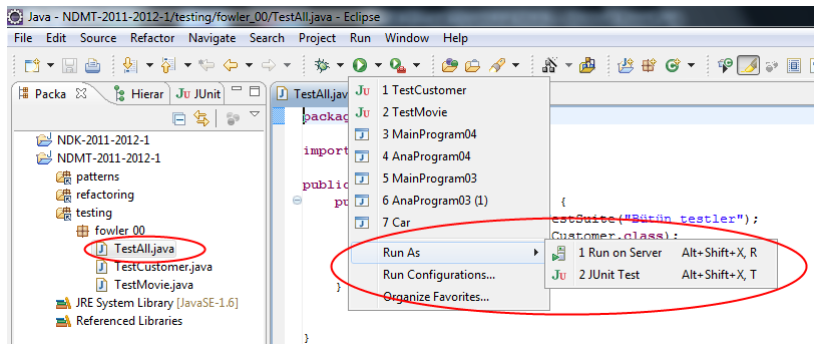
- Test suite:

```
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;
   }
}
```

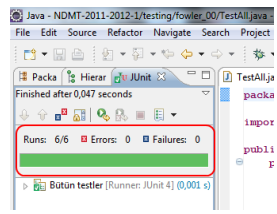
271

JUNIT İLE BİRİM SINAMALARI

- Testi çalıştıralım …



- … ve sonuç:



272

CODE SMELLS – DUPLICATED CODE

YİNELENEN KOD

- Aynı kodun veya çok benzer kod yapısının birden fazla yerde tekrarlanması durumudur.
- Bu kodun işlevselliğinde değişiklik yapılması gerektiğinde, 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 değinilecektir.

ÜYE ALAN ADLANDIRMA

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

273

REFACTORING – EXTRACT METHOD

METOT ÇIKARTMA

- Bir metot içindeki kodun bir kısmını yeni bir metot olarak belirlemeye dayanır.
- Bu yapılandırma bir çok kod kusurunu (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);
}
```

274

CODE SMELLS – LONG METHOD

AŞIRI UZUN KOD

- Bir metodun uzunluğu arttıkça:
  - Anlaşılması 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.

275

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);
}
```

276

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 metotlar yerine kısa ve adı ile kendi kendini açıklayan metotların kullanılması önerilir.
  - Öyle ki, kodda yorum satırı bulunmasına hiç gerek kalmasın.
  - Kısa sınıflar için verilen yararlar kısa metotlar 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 metotların kullanımı yüzünden başarım olumsuz etkilenmez.

277

CODE SMELLS – DUPLICATED CODE

YİNELENEN KOD (devam)

- Çözüme götürecek diğer düzenlemeler:
  - Pull Up Method: Yinelenen kod kalıtım ağacındaki kardeş veya kuzen sınıflara yayılmışsa, oluşturulan metot 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 metot, oluşturulacak yeni bir sınıfa atanır ve diğerleri bu sınıfı kullanır.
  - Özetle, Extract Method ile oluşturulan yeni metoda en uygun yer aranır.

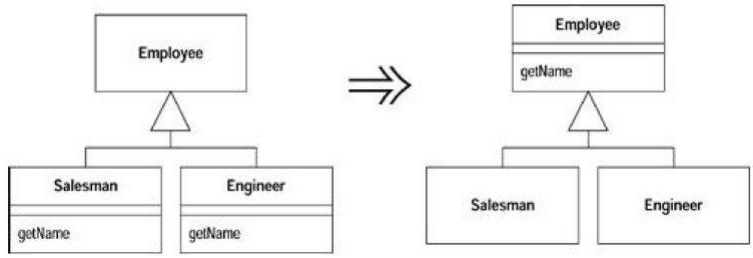
278

REFACTORING – PULL UP METHOD

METODU YUKARI ÇEKME

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

/*** ÖNCE: ***/	/*** SONRA: ***/



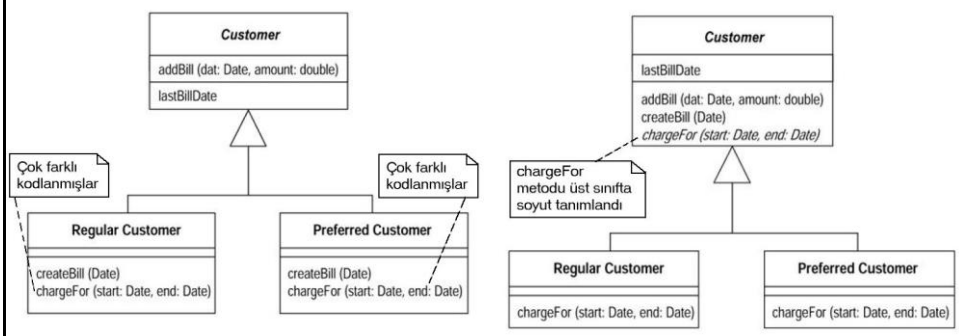
279

REFACTORING – PULL UP METHOD

METODU YUKARI ÇEKME

- Özel durumlar:

/*** ÖNCE: ***/	/*** SONRA: ***/



- Özel durumlar:
  - Template Method tasarım kalıbı kullanılabilir.

280

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.

281

REFACTORING – DECOMPOSE CONDITIONAL

KARAR VERME KOMUTUNU PARÇALAMA

- Aşırı karmaşık if komutlarını sadeleştirmek, anlaşılabilirliği arttırır:

```
/*** 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;
}
```

282

CODE SMELLS – LONG PARAMETER LIST

UZUN PARAMETRE LİSTESİ

- Bir metodun parametre sayısının fazla olması, hatta bu parametrelerden bazılarının 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 düzenlemedir.
  2. Preserve Whole Object: Metot parametrelerinin sayısı çok ise azaltmaya yönelik diğer bir düzenlemedir.

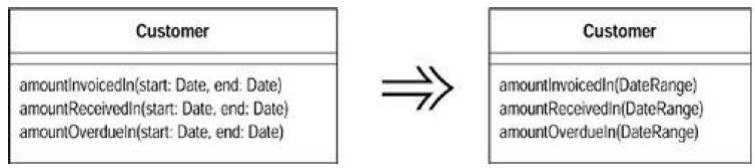
283

REFACTORING – INTRODUCE PARAMETER OBJECT

PARAMETRE NESNESİ OLUŞTURMA

- Bir metot çok fazla parametre alıyorsa, uygun parametreleri üye alan olarak yeni bir sınıfta 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 metotlar da içererek, “decompose conditional” veya “extract method” düzenlemesi de yapılmış olunur.

284

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 gerektiği gibi değiştirilir.
```

- 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.

285

REFACTORING – REPLACE TEMP WITH QUERY

GEÇİCİ DEĞİŞKEN YERİNE ERİŞİM METODU KULLANIMI

- Bir ifadeyi bir geçici değişkende saklayarak kullanmak yerine, ifadeden metot(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;
}
```

286

REFACTORING – REPLACE TEMP WITH QUERY

GEÇİCİ DEĞİŞKEN YERİNE ERİŞİM 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 sınıfta buradan kopyala-yapıştır yapılan yerler aranıp her yerde düzeltme yapmak gerekecek.
  - Halbuki geçici değişkenlerle yapılan işlemler sorgu metotlarında toplanırsa, düzeltmeler de sadece sorgu metotlarında yapılacaktır.
- Tartışma: Bu düzenleme başarımı düşürür.
  - Sadece işlem yapmak yerine bir de fazladan metot çağırma yükü geliyor.
- Şerh:
  - Modern derleyiciler bu ek yükü ortadan kaldırmış veya önemsizleştirmiştir.

287

REFACTORING – REPLACE TEMP WITH QUERY

GEÇİCİ DEĞİŞKEN YERİNE ERİŞİM METODU KULLANIMI

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

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

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

288

REFACTORING – REPLACE TEMP WITH QUERY

GEÇİCİ DEĞİŞKEN YERİNE ERİŞİM METODU KULLANIMI

- Tartışma:
  - Az önceki durumda başarım daha da düşecek.
- Şerh:
  - Başarım, 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.

289

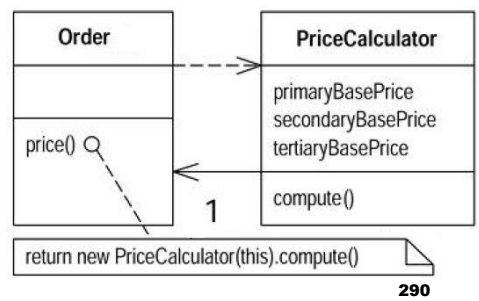
REFACTORING – REPLACE METHOD WITH METHOD OBJECT

METOD YERİNE YENİ TÜRDEN NESNE KULLANIMI

- Metot öyle uzun ve geçici yerel değişkenleri öyle arap saçı 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 bildiğiniz düzenleme eylemlerini daha rahat yürütebilirsiniz.
- 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: ***/
```



290

CODE SMELLS – LARGE CLASS

AŞIRI BÜYÜK SINIF

- Sınıfın büyük olması, sınıfa 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 eşgüdüm Observer tasarım kalıbına göre sağlanır.

291

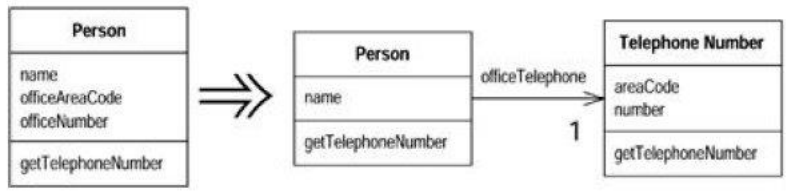
REFACTORING – EXTRACT CLASS

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: ***/
```



292

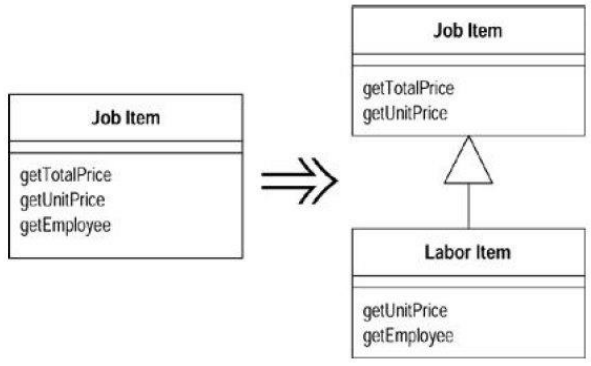
REFACTORING – EXTRACT SUBCLASS

ALT SINIF ÇIKARTMA

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

```
/*** ÖNCE: ***/
```

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



293

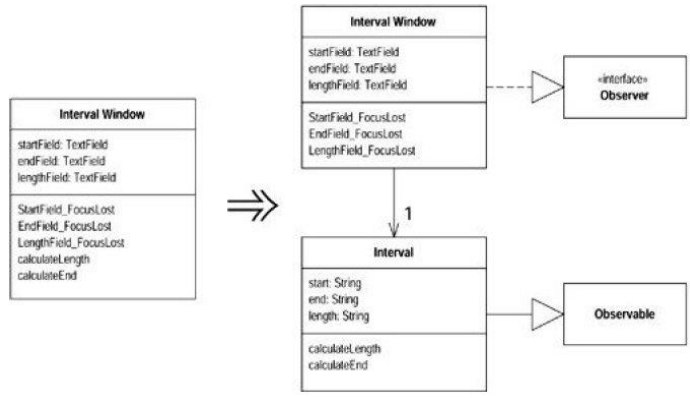
REFACTORING – DUPLICATE OBSERVED DATA

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 taşınmalı ve,
- GUI nesnesi ile yeni bilgi nesnesi arasında Observer tasarım kalıbı kurulmalıdır.

```
/*** ÖNCE: ***/
```

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



294

CODE SMELLS – DIVERGENT CHANGE & SHOTGUN SURGERY

DEĞİŞİKLİKLERİN FARKLI NOKTALARDA YAPILMASININ GEREKMESİ

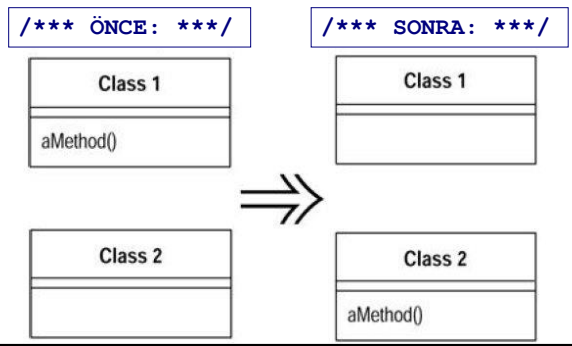
- Gereksinimlerde bir düzenleme olduğunda veya bir hatayı düzeltmek için kodda değişiklikler yapılması gerekecektir.
- Bu değişikliklerin mümkün olduğunca az yerde yapılması tercih edilir.
- Halbuki bir gereksinim değişikliğini karşılamak için gerekli kod değişiklikleri aynı sınıfın farklı metotlarında yapılıyorsa (divergent change) veya birden fazla sınıfta yapılıyorsa (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 sınıftaki şu 5 metot değişmeli
- Çözüme götürecek düzenlemeler:
  - Extract Class: Söz konusu metotlar ve ilgili üyelerden yeni bir sınıf çıkartılır.
  - Move Method / Move Field: Söz konusu üyeler daha uygun bir mevcut sınıfa taşınır.

295

REFACTORING – MOVE METHOD

METOT TAŞINMASI

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



296

CODE SMELLS – PRIMITIVE OBSESSION

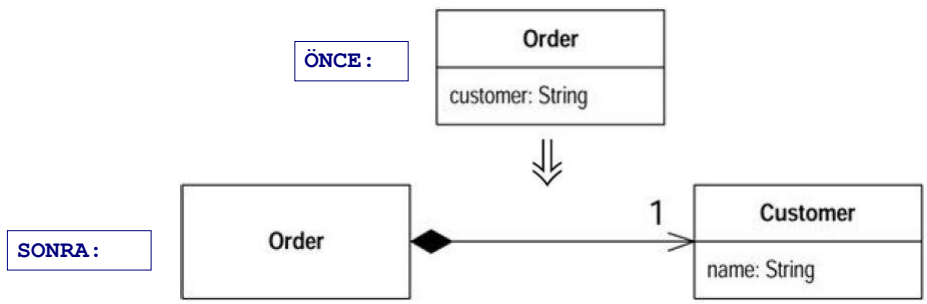
İLKEL TİPLERİN AŞIRI KULLANIMI

- Yapısal programlamadan gelenler, ilkelleri 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 yönelimli bir program hazırlamak için kullanılabilir.

297

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:

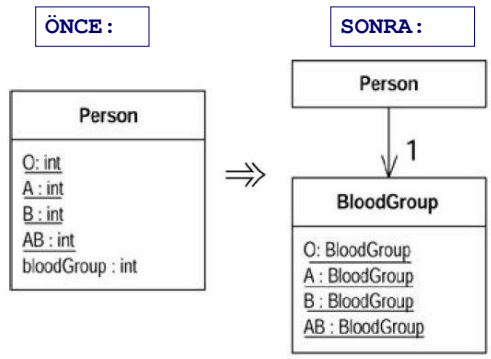


- Programın tümü incelendiğinde, belli bir müşterinin tüm siparişlerinin aranması gibi işlemler yapıldığı görülecektir.
- Bu işlemler çoğaldıkça, Müşteri sınıfı ayrılmadığı sürece, Sipariş sınıfı aşırı büyüyecektir.
- İyisi mi şimdiden bu sınıfları ayıralım.

298

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 yansıyı atlayabiliriz.

299

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;
  }
}
```

- Dikkat: BloodGroup sınıfının tüm metotarı private.

```
/*** 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]; }
}
```

300

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;
  }
}
```

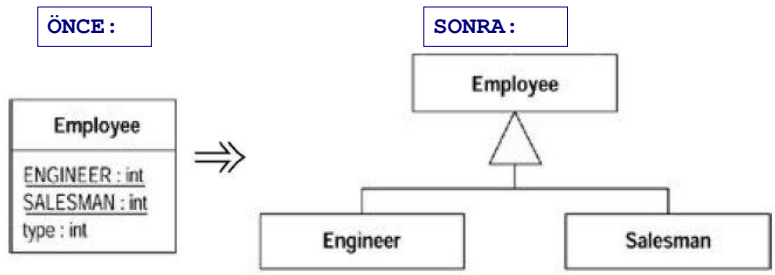
```
/*** SONRA: ***/
class Person {
  private BloodGroup _bloodGroup;
  public Person (BloodGroup bg) {
    _bloodGroup = bg; }
  public BloodGroup
    getBloodGroup()
  { return _bloodGroup; }
  public void setBloodGroup (
    BloodGroup bg ) {
    _bloodGroup = bg; }
}
```

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

301

REFACTORING – REPLACE TYPE CODE WITH SUBCLASSES

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



302

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
    Employe {
   public void work( ) {
     //do something as an engineer
   }
}
public class Salesman...
```

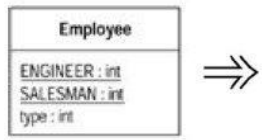
- Not: Employee'nin önceki halinde case default varsa, sonraki halde Employee sınıfı abstract yapılmaz.
- Not: Bu örnek Replace Conditional with Polymorphism düzenlemesine de benzemektedir.

303

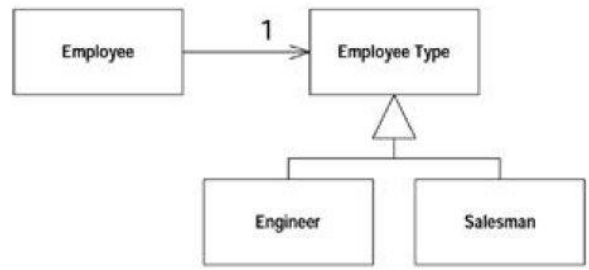
REFACTORING – REPLACE TYPE CODE WITH STATE/STRATEGY

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

ÖNCE:



SONRA:



304

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");
```

305

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 metotlarını 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.

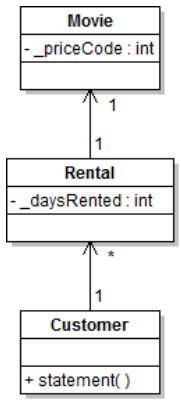
BİR TASARIMIN ARDIŞIL DÜZENLEMELER İLE DEĞİŞTİRİLMESİ

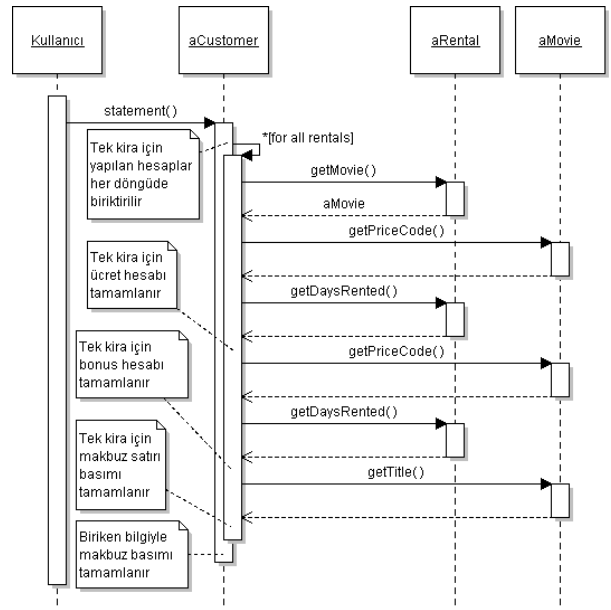
- Şimdiye kadar incelenen düzenlemelerden bazılarının birlikte kullanılarak, yapısal programlama yaklaşımı izleri taşıyan bir tasarımın nasıl değiştirildiğinin bir örneği için Fowler'ın Refactoring kitabının 1. bölümü incelenebilir.

306

REFACTORING ÖRNEĞİ – İLK DURUM

- Basit video kiralama örneğini hatırlayalım:
  - Örneğin kaynak kodu: refactoring/fowler00





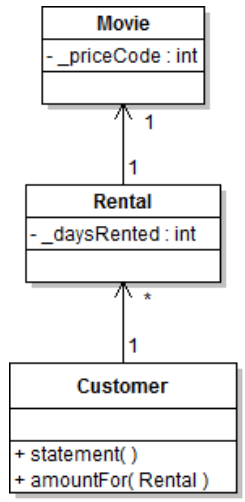
REFACTORING ÖRNEĞİ – 1. MÜDAHALE

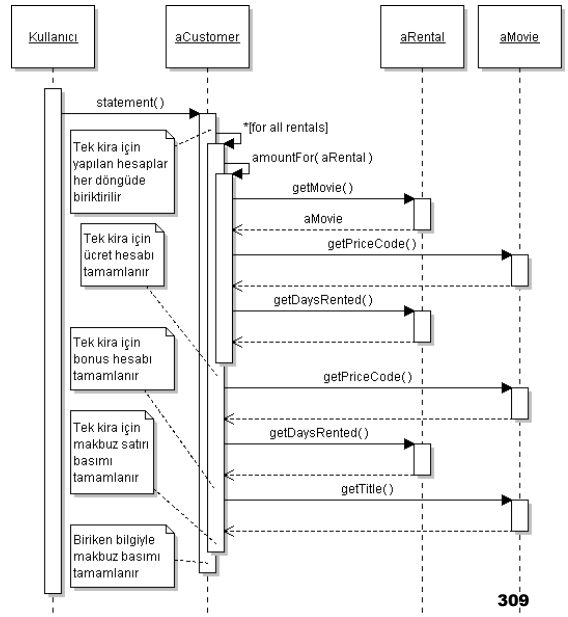
- İlk adım: Aşırı uzun olan statement metodunu parçalamak
  - statement metodunun da en uzun parçası switch bloğu olarak gözüküyor.
  - Aslında switch kullanımı çokbiçimliliğe ihtiyaç duyulduğunun göstergesidir.
  - Ancak değişiklikleri küçük adımlarla yapmak, hata yapmamız halinde hatamızı düzeltmeyi kolaylaştıracaktır.
  - Sonuç olarak bu adımda switch bloğu, geçici değişkenlere de dikkat ederek, amountFor adlı bir metotda taşınmıştır:
    - Taşınacak alanda iki yerel değişkene erişim yapıldığı için, Eclipse düşünülen düzenlemeyi gerçekleştirememiştir.
  - Sonuç kod: Eclipse üzerinden
    - refactoring/fowler01 dizininde
  - Test case'lerimizi yeniden çalıştırmayı unutmayalım.
    - Sadece Customer sınıfında değişiklik yaptığımız için, TestCustomer birim testini çalıştırmak yeter.
      - testing/fowler01 dizininde
      - Aslında test kodu değişmedi ancak Customer sınıfının yeni halini fowler01 paketine aldığımız için TestCustomer sınıfı da bu yeni pakete kopyalandı.

308

REFACTORING ÖRNEĞİ – 1. MÜDAHALE

- Sonuç:





309

REFACTORING ÖRNEĞİ – 2. MÜDAHALE

- İkinci adım: amountFor metodunun değişken adlarının düzenlenmesi
  - Eclipse ile gayet kolay.
  - Sonuç kod: Eclipse üzerinden, karşılaştırma ile.
    - each → aRental
    - thisAmount → result

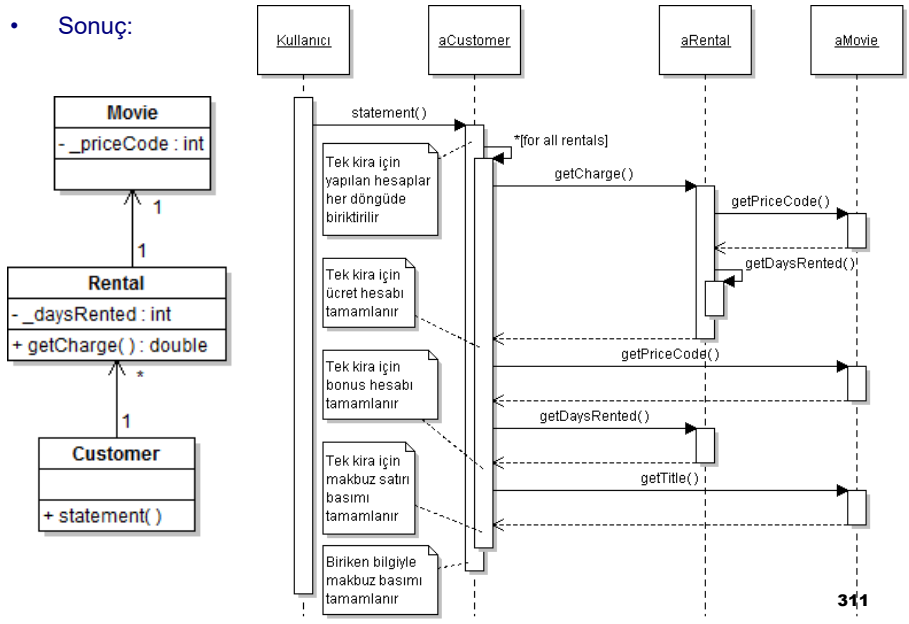
REFACTORING ÖRNEĞİ – 3. MÜDAHALE

- Üçüncü adım: amountFor metodunun taşınması
  - Bu metot Customer sınıfında olmasına rağmen müşteri hakkında herhangi bir bilgi kullanmıyor.
  - Bu metot Rental sınıfının üyelerine erişiyor.
  - Bu nedenle uyumu (cohesion) yükseltmek için amountFor metodunu daha uygun bir isim de vererek, Rental sınıfına taşıyalım:
    - amountFor → getCharge
    - Eclipse ile gayet kolay.
    - Sonuç kod: Eclipse üzerinden

310

REFACTORING ÖRNEĞİ – 3. MÜDAHALE

- Sonuç:



311

TESTLERDE KARA KUTU/BEYAZ KUTU YAKLAŞIMI

- Kara kutu sınaması (Black-box testing): Sınanacak birimin iç işleyişi bilinmez, sadece birimin beklenen girdilere karşı beklenen çıktıları üretip üretilmediğine bakılır.
- Beyaz kutu sınaması (White-box testing): Sınanacak birimin iç işleyişi bilinir ve yapılacak sınamalar buna göre belirlenir.
- Tartışma:
  - Üçüncü adımın sonucunda, Rental sınıfı yeni bir metot kazandı.
  - Acaba Rental.getCharge metodu için yeni bir test case oluşturalım mı (beyaz kutu), yoksa mevcut test case'ler ile devam mı edelim (kara kutu)?
  - Beyaz kutuyu seçmek test kalitemizi arttırır ancak sınırlı proje zamanımızın uzamasına neden olabilir.
  - Bu aşamada mevcut test case'ler ile devam etmeyi seçtim.

312

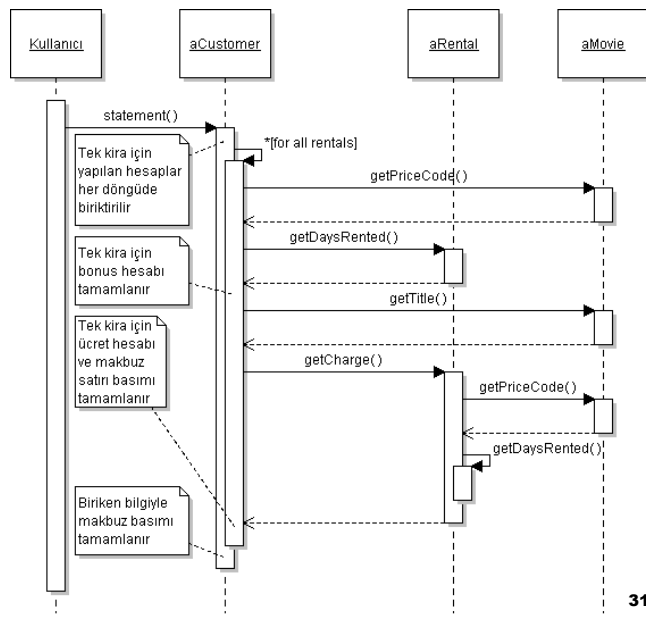
REFACTORING ÖRNEĞİ – 4. MÜDAHALE

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

313

REFACTORING ÖRNEĞİ – 4. MÜDAHALE

- Sonuç:



314

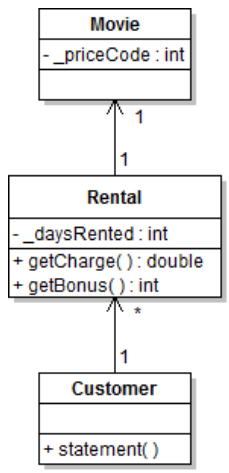
REFACTORING ÖRNEĞİ – 5. MÜDAHALE

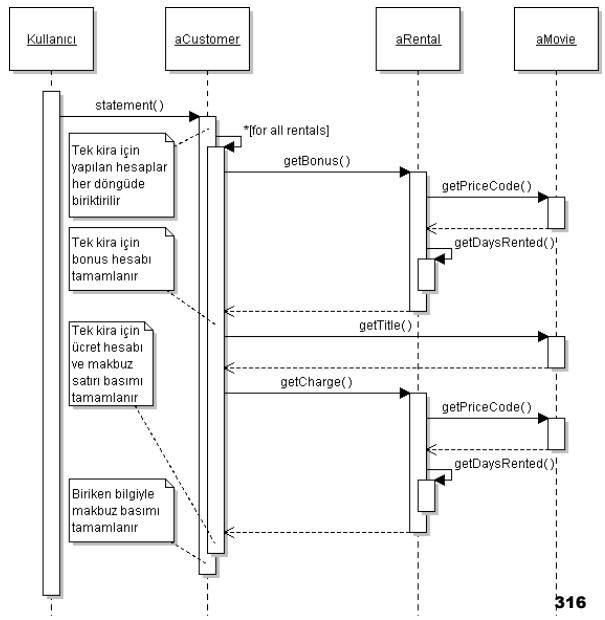
- Beşinci adım: Bonus hesaplaması için metot çıkartma işlemi.
  - Bonus (frequentRenterPoints) kiralama işlemi ile ilgili olduğu için, bu işlemin Rental sınıfına alınması yerinde olacaktır.
  - FrequentRenterPoints çok uzun → Bonus kullanıldı.
  - Sonuç kod: Eclipse üzerinden, karşılaştırma ile.
- UML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

315

REFACTORING ÖRNEĞİ – 5. MÜDAHALE

- Sonuç:

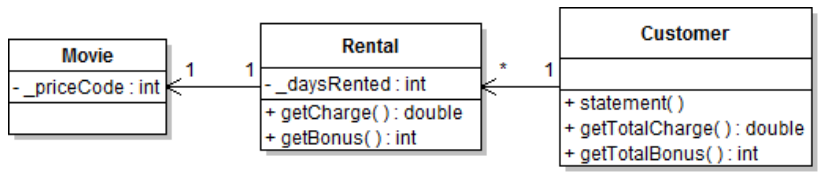




316

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:



317

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.

318

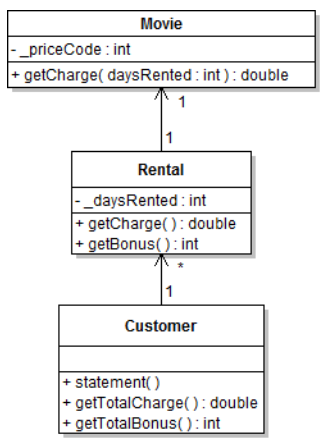
REFACTORING ÖRNEĞİ – 7. MÜDAHALE

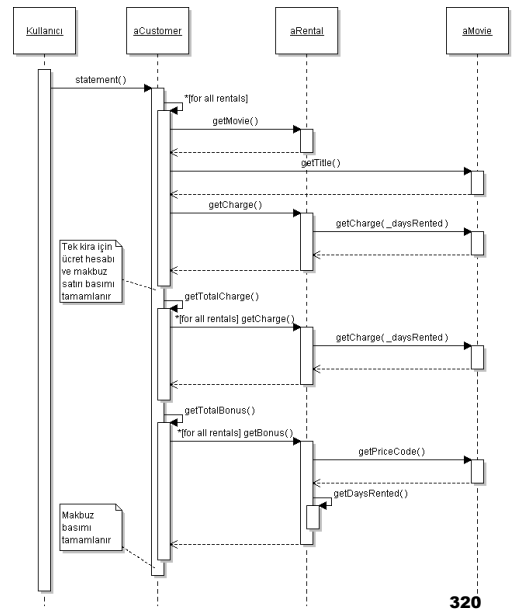
- Yedinci değişiklik: getCharge metodunun bölünmesi
  - getCharge metodu Rental üyelerine erişiyor diye 3. adımda bu sınıfa taşındı.
  - Ancak bu metot aynı zamanda Movie üyelerine de erişiyor.
  - Üstelik bu erişim switch-case bloğunda kullanılıyor.
  - Bir başka sınıfın üyelerinin durumlarına switch-case bloğunda erişmek akıllıca 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.

319

REFACTORING ÖRNEĞİ – 7. MÜDAHALE

- Sonuç:

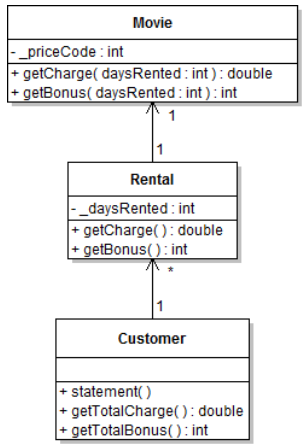


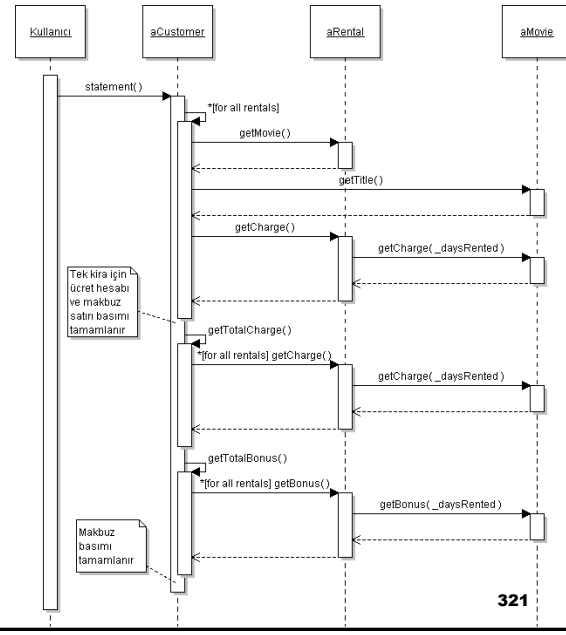


320

REFACTORING ÖRNEĞİ – 8. MÜDAHALE

- Sekizinci değişiklik: getBonus metodunun da önceki gibi bölünmesi:





321

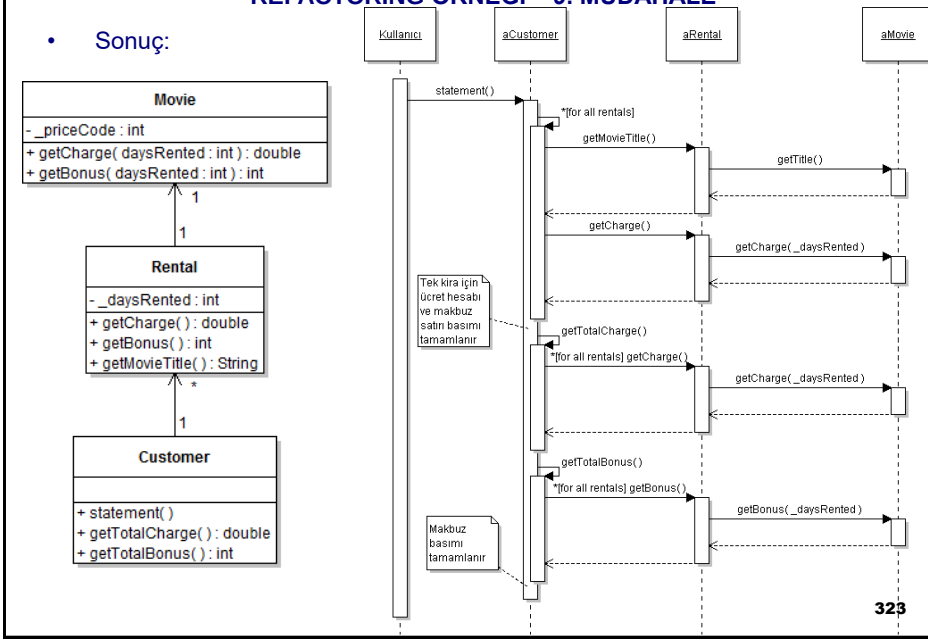
REFACTORING ÖRNEĞİ – 9. MÜDAHALE

- 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.

322

REFACTORING ÖRNEĞİ – 9. MÜDAHALE

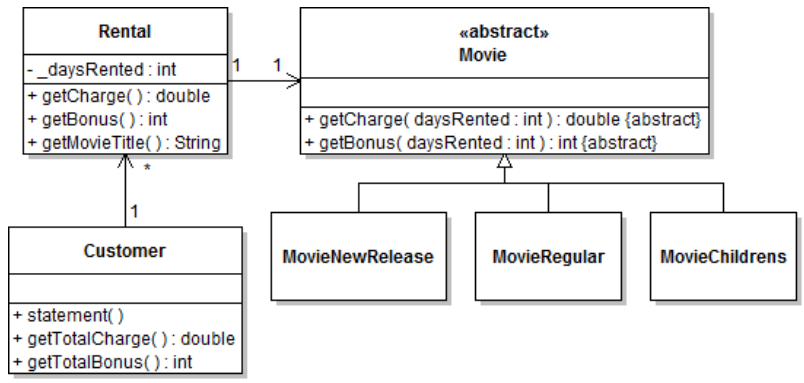
- Sonuç:



323

REFACTORING ÖRNEĞİ – 10. MÜDAHALE

- Onuncu değişiklik: getCharge metodundaki switch/case ve getBonus metodundaki if deyimleri yerine çokbiçimlilik ilişkisi kullanılması (kalıtım ve soyut metotlar kullanılarak).



324

ANTI PATTERNS (KARŞIT KALIPLAR)

ANTIPATTERN NEDİR?

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

KARŞIT KALIP İLE KOD KUSURU FARKI:

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

KARŞIT KALIP TÜRLERİ

- Karşıt kalıplar üç ayrı grupta incelenmektedir: (henüz müfredata eklenmedi)
  - Yazılım geliştirme karşıt kalıpları
  - Yazılım mimarisi karşıt kalıpları
  - Yazılım proje yönetimi karşıt kalıpları

325