Özcan Acar

Design Patterns

Tasarım Şablonlarının Pratik Uygulanış Rehberi

pratikprogramci.com

PRATİK DESIGN PATTERNS

Design Patterns

Tasarım Şablonlarının Pratik Uygulanış Rehberi

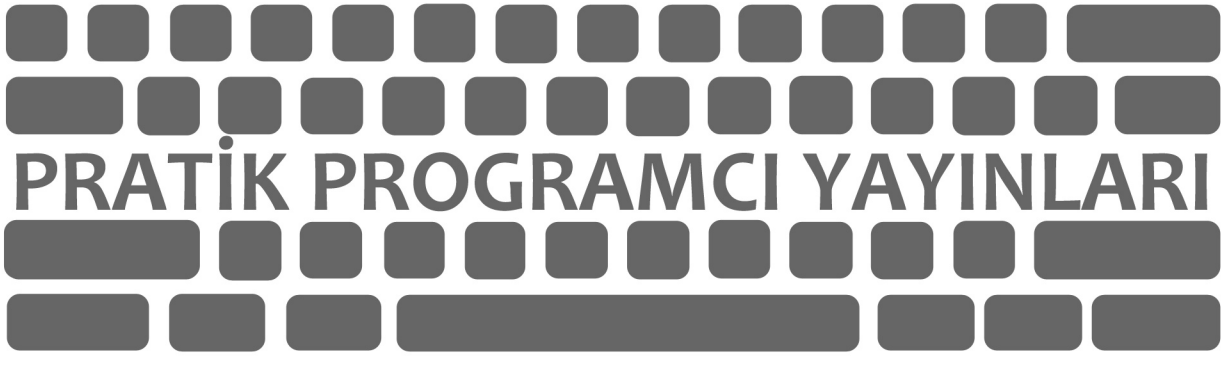
Özcan Acar

PRATİK PROGRAMCI YAYINLARI

PRATİK DESIGN PATTERNS . Copyright © 2016 Pratik Programcı Yayınları.

Tüm telif ve yayın hakları Pratik Programcı Yayınları'na aittir. Telif hakkı sahibinin yazılı izni olmadan kısmen ya da tamamen alıntı yapılamaz, kopya edilemez, çoğaltılamaz, dağıtılamaz ve yayınlanamaz.

Yazar: Özcan Acar
Yayınevi: Pratik Programcı Yayınları
İlk sürüm: Nisan 2016
Kapak tasarımı: Ahmet Yıldırım
Düzelti: Ahmet Yıldırım
Satış: http://www.pratikprogramci.com



pratikprogramci.com

Asya'daki (kızım) Vatan'ım (eşim), Türkiye'm için ...

Bilgi Paylaşım İle Çoğalır

Lütfen bu kitabın ve ihtiva ettiği bilginin yaygınlaşması için kitabı tanıdıklarınız için paylaşınız.

Pratik Programcı Yayınları
http://www.pratikprogramci.com

Bölüm Başlıkları

Kitap 10 bölümden oluşmaktadır. Ana bölüm başlıkları şunlardır:

- 1.Bölüm - Tasarım Şablonları Giriş
- 2.Bölüm - Nesneye Yönelik Programlama
- 3.Bölüm - Unified Modeling Language (UML) Giriş
- 4.Bölüm - Tasarım Prensipleri
- 5.Bölüm - Oluşturucu Tasarım Şablonları
- 6.Bölüm - Yapısal Tasarım Şablonları
- 7.Bölüm - Davranışsal Tasarım Şablonları
- 8.Bölüm - JEE Tasarım Şablonları
- 9.Bölüm - Daha Fazla Tasarım Şablonu
- 10.Bölüm - Yazılım Mimarisi ve Tasarım Şablonlarının Pratik Kullanımı

İçindekiler

Bilgi Paylaşım İle Çoğalır 12
Bölüm Başlıkları 16
Önsöz 23
Kitabın İçeriği Nedir? 23
Kitabın İçeriği Ne Degildir? 25
Kitap Kim İçin Yazıldı? 25
Yazar Hakkında 25
Kitap Nasıl Okunmalı? 26
Yazar İle İletişim 26
PratikProgramci.com 26
Kitapta Yer Alan Kod Örnekleri 26
1. Bölüm 28
Tasarım Şablonları Giriş 28
Tasarım Şablonu Nedir? 29
Tasarım Şablonu Neden Kullanılır? 29
Tasarım Şablonları Nasıl Kullanılmalıdır? 29
Kitapta Yer Alan Tasarım Şablonları 30
Oluşturucu tasarım şablonları 30
Yapısal tasarım şablonları 30
Davranışsal tasarım şablonları 31
Java EE tasarım şablonları 31
Diğer tasarım şablonları 31
2. Bölüm 34
Nesneye Yönelik Programlama - Object Oriented Programming 34
Interface Nedir? 44
Soyut (Abstract) Sınıf Nedir? 48
Interface Örneği 49
Neden Abstract ve Interface Sınıflar Yeterli Değildir? 56
3. Bölüm 62
Unified Modeling Language (UML) Giriş 62
Sınıf Diyagramları 63
Dizi (Sequence) Diyagramları 66
Kullanım Senaryo (Usecase) Diyagramları 68
Aktivite (Activity) Diyagramları 69
4. Bölüm 72
Tasarım Prensipleri 72
Loose Coupling (LC) - Esnek Bağ 73
Open Closed Principle (OCP) - Açık Kapalı Prensibi 79
Stratejik Kapama (Strategic Closure) 83
Single Responsibility Principle (SRP) – Tek Sorumluk Prensibi 83
Liskov Substitution Principle (LSP) – Liskov Yerine Geçme Prensibi 85
Dependency Inversion Principle (DIP) – Bağımlılıkların Tersine Çevrilmesi Prensibi 90
Interface Segregation Principle (ISP) – Arayüz Ayırma Prensibi 90
Paket Tasarım Prensipleri (Principles of Package Design) 92
Reuse-Release Equivalence Principle (REP) – Tekrar Kullanım ve Sürüm Eşitliği 92
Common Reuse Principle (CRP) – Ortak Yeniden Kullanım Prensibi 93
Common Closure Principle (CCP) – Ortak Kapama Prensibi 94

Acyclic Dependency Principle (ADP) – Çevrimsiz Bağımlılık Prensibi 95
Stable Dependencies Principle (SDP) – Stabil Bağımlılıklar Prensibi 97
Stable Abstractions Principle (SAP) – Stabil Soyutluk Prensibi 99
Soyutluk (A) ve Instability (I) Arasındaki İlişki 100
Iyi Bir Tasarım 102
Kalıtım yerine kompozisyon kullanılmalıdır 102
Statik metot ve tekil yapılar kullanılmamalıdır 103
Bağımlılıkların izole edilmesi gerekir 104
Bağımlılıkların enjekte edilmesi testleri kolaylaştırır 105
5. Bölüm 109
Oluşturucu Tasarım Şablonları - Creational Patterns 109
Fabrika (Factory) 110
Fabrika tasarım şablonu ne zaman kullanılır? 118
İlişkili tasarım şablonları 118
Fabrika Metodu (Factory Method) 118
Fabrika metodu tasarım şablonu ne zaman kullanılır? 122
İlişkili tasarım şablonları 122
Soyut Fabrika (Abstract Factory) 122
Soyut fabrika tasarım şablonu ne zaman kullanılır? 128
İlişkili tasarım şablonları 128
Tekillik (Singleton Pattern) 128
Tekillik tasarım şablonu ne zaman kullanılır? 133
İlişkili tasarım şablonları 133
Yapıcı (Builder) 133
Yapıcı tasarım şablonu ne zaman kullanılır? 143
İlişkili tasarım şablonları 144
Prototip (Prototype) 144
Prototip tasarım şablonu ne zaman kullanılır? 150
İlişkili tasarım şablonları 150
Nesne Havuzu (Object Pool) 150
Nesne havuzu tasarım şablonu ne zaman kullanılır? 154
İlişkili tasarım şablonları 154
6. Bölüm 156
Yapısal Tasarım Şablonları - Structural Patterns 156
Adaptör (Adapter) 157
Sınıf Adaptörü (Class Adapter) 157
Nesne Adaptörü (Object Adapter) 161
Adaptör tasarım şablonu ne zaman kullanılır? 164
İlişkili tasarım şablonları 164
Köprü (Bridge) 164
Köprü tasarım şablonu ne zaman kullanılır? 175
İlişkili tasarım şablonları 176
Cephe (Facade) 176
Cephe tasarım şablonu ne zaman kullanılır? 181
İlişkili tasarım şablonları 181
Bileşik (Composite) 181
Bileşik tasarım şablonu ne zaman kullanılır? 187
İlişkili tasarım şablonları 187
Dekoratör (Decorator) 188
Dekoratör tasarım şablonu ne zaman kullanılır? 192
İlişkili tasarım şablonları 193
Vekil (Proxy) 193
Sanal Vekil (Virtual Proxy) 194
Koruyucu Vekil (Protection Proxy) 195
Dinamik Vekil 206

Uzak Vekil 218
Vekil tasarım şablonu ne zaman kullanılır? 219
İlişkili tasarım şablonları 219
Sinek Siklet (Flyweight) 220
Sinek siklet tasarım şablonu ne zaman kullanılır? 226
İlişkili tasarım şablonları 226
7. Bölüm 228
Davranışsal Tasarım Şablonları 228
Komut (Command Pattern) 229
Komut tasarım şablonu ne zaman kullanılır? 233
İlişkili tasarım şablonları 233
Döngücü (Iterator Pattern) 233
Döngücü tasarım şablonu ne zaman kullanılır? 243
İlişkili tasarım şablonları 244
Hatırlayan (Memento Pattern) 244
Memento tasarım şablonu ne zaman kullanılır? 249
İlişkili tasarım şablonları 249
Durum (State Pattern) 249
Durum tasarım şablonu ne zaman kullanılır? 254
İlişkili tasarım şablonları 254
Gözetleyici (Obverser Pattern) 254
Gözetleyici tasarım şablonu ne zaman kullanılır? 260
İlişkili tasarım şablonları 260
Strateji (Strategy Pattern) 260
Strategy tasarım şablonu ne zaman kullanılır? 268
İlişkili tasarım şablonları 268
Sorumluluk Zinciri (Chain Of Responsibility Pattern) 268
Chain of Responsibility tasarım şablonu ne zaman kullanılır? 276
İlişkili tasarım şablonları 276
Aracı (Mediator Pattern) 277
Mediator tasarım şablonu ne zaman kullanılır? 289
İlişkili tasarım şablonları 289
Ziyaretçi (Visitor Pattern) 289
Visitor tasarım şablonu ne zaman kullanılır? 294
İlişkili tasarım şablonları 294
Şablon Metodu (Template Method Pattern) 294
Template Method tasarım şablonu ne zaman kullanılır? 297
İlişkili tasarım şablonları 298
Yorumlayıcı (Interpreter Pattern) 298
Yorumlayıcı tasarım şablonu ne zaman kullanılır? 301
İlişkili tasarım şablonları 301
8. Bölüm 303
JEE Tasarım Şablonları 303
Java Enterprise Edition (JEE) Platformu 304
MVC Tasarım Tasarım Şablonu 304
Front Controller Tasarım Şablonu 305
Front Controller tasarım şablonu ne zaman kullanılır? 310
İlişkili tasarım şablonları 310
DAO (Data Access Objects) Tasarım Şablonu 310
DAO tasarım şablonu ne zaman kullanılır? 322
İlişkili tasarım şablonları 322
Servise Yönlendirme Tasarım Şablonu (Business Delegate) 322
Business Delegate tasarım şablonu ne zaman kullanılır? 329
İlişkili tasarım şablonları 330
Servis Lokalizasyonu Tasarım Şablonu (Service Locator) 330

Service Locator tasarım şablonu ne zaman kullanılır? 331
İlişkili tasarım şanlonları 331
Filtreleme Tasarım Şablonu (Intercepting Filter) 331
Intercepting Filter tasarım şablonu ne zaman kullanılır? 335
İlişkili tasarım şablonları 335
Business Object Tasarım Şablonu 336
Business Object tasarım şablonu ne zaman kullanılır? 337
İlişkili tasarım şablonları 338
9. Bölüm 340
Daha Fazla Tasarım Şablonu 340
DataMapper Tasarım Şablonu 341
DataMapper tasarım şablonu ne zaman kullanılır? 346
İlişkili Tasarım Şablonları 346
RequestMapper ve ResponseMapper Tasarım Şablonları 346
RequestMapper/ResponseMapper tasarım şablonu ne zaman kullanılır? 347
İlişkili Tasarım Şablonları 347
Active Record Tasarım Şablonu 348
Active Record tasarım şablonu ne zaman kullanılır? 350
İlişkili Tasarım Şablonları 350
Message Channel Tasarım Şablonu 350
Message Channel tasarım şablonu ne zaman kullanılır? 355
İlişkili Tasarım Şablonları 355
Message Router Tasarım Şablonu 355
Message Router tasarım şablonu ne zaman kullanılır? 360
İlişkili tasarım şablonları 360
Registry Tasarım Şablonu 360
Registry tasarım şablonu ne zaman kullanılır? 362
İlişkili tasarım şablonları 362
Null Object Tasarım Şablonu 362
Null Object tasarım şablonu ne zaman kullanılır? 364
İlişkili tasarım şablonları 364
Dependency Injection Tekniği 364
Dependency Injection tekniği ne zaman kullanılır? 367
İlişkili tasarım şablonları 368
10. Bölüm 370
Yazılım Mimarisi ve Tasarım Şablonlarının Pratik Kullanımı 370
3 Katmanlı Mimari 371
Otel Rezervasyon Programı 373
UML Diagramı 375
Gösterim Katmanı Sınıfları 376
İşletme Katmanı Sınıfları 376
Veri Depolama / Edinme Katmanı 377
UML Dizge (Sequence) Diagramı 378
UML Kullanım Senaryosu (Use Case) Diagramı 379
Gösterim (Presentation) Katmanı 380
İşletme (Business) Katmanı 394
Veri Depolama (Persistence) / Edinme Katmanı 402
Bu Uygulamayı Nasıl Çalışır Hale Getirebilirsiniz 409
Son Söz 412

Önsöz

Bu kitapta profesyonel yazılım için vazgeçilmez olan tasarım şablonlarını (design patterns) tematize etmek istedim. Çalıştığım birçok projede görevim mevcut sistemlere yeni fonksiyonlar eklemekti. Başka bir programcı tarafından yazılmış bir kodun anlaşılmasının ne kadar zor olabileceğini programcılar çok iyi bilirler. Bunun yanı sıra genel yazılım kuralları izlenmediği için çoğu zaman var olan program parçalarının yeni metotlar eklenerek genişletilemediğini gördüm. Bu durum beni zaman içinde daha çok refactoring olarak bilinen kodun yeniden yapılandırılması ve tasarım şablonları üzerinde bilgi edinmeye itti. Yaptığım araştırmalar sonunda mevcut kodun refactoring ile sadeleştirilebileceğini ve uyguladığım tasarım şablonları ile kodun, benim ve benden sonra kod üzerinde çalışan programcıların hayatını daha kolaylaştırıdığını gördüm.

Ağaç yaşken eğilir demiş atalarımız. Bunu yazılım için de geçerli bir deyim olarak kabul edebiliriz. Her proje başlangıcında kodun nasıl yapılandırılabileceği ve hangi tasarım şablonlarının kullanılacağı tesbit edilmeli ve ön çalışmalar buna göre yürütülmelidir. Zaman içinde uygulanan tasarım şablonlarının programcı ekibin hayatını daha da kolaylaştırdığı gözlenecektir.

Yazılım disiplini başlı başına bir bilim dalıdır. Bunu bilgisayar mühendisliği okumuş ya da programcı olarak çalışan arkadaşlarımız tastikleyecektir. Tabii ki karşılaştığımız her soruna hemen bir çözüm bulamayabiliriz ya da bu zaman alabilir. Her sorunu kendi başımıza çözmek yerine, buna tekeri yeniden icat etmek diyebiliriz, mevcut tecrübelerin neticesi olan tasarım şablonlarından faydalanabiliriz. Tasarım şablonları uzun yıllar edinilen tecrübelerle oluşturulmuş kalıplardır. Bu kalıpların kullanılması, yazılım süreceni hızlandıracak ve netice itibari ile bakımı ve geliştirilmesi daha kolay programların oluşturulmasını sağlayacaktır.

Kitabın İçeriği Nedir?

Kitabın içeriğini tasarım şablonları oluşturmaktadır. Her tasarım şablonu Java dilinde hazırlanmış bir örnek ile açıklanmıştır. Böylece okuyucu pratik olarak bir tasarım şablonunun nasıl uygulanabileceğini görmektedir. Kitapta yer alan örnekleri takip edebilmek için okuyucunun temel Java bilgisine ihtiyacı bulunmaktadır.

Kitapta UML diyagramları kullanılarak, tasarım şablonları görsel olarak açıklanmıştır. İçerik aşağıdaki bölümlerden oluşmaktadır:

Bölüm 1:

Bu bölüm tasarım şablonlarına giriş mahiyetindedir ve tasarım şablonları hakkında genel

tanımlamalar ihtiva etmektedir.

Bölüm 2:

Bu bölümde nesneye yönelik programlama tekniği (object oriented programming) tanıtılmaktadır.

Bölüm 3:

Kitapta yer alan tasarım şablonları UML (unified modeling language) diyagramları kullanılarak görsellestirilmiştir. Üçüncü bölüm UML e giriş niteliğindedir ve bu bölümde temel UML diyagramları tanıtılmaktadır.

Bölüm 4:

Bu bölümün ana konusu tasarım prenbipleridir. Tasarım şablonları yanı sıra bir uygulamayı esnek, değiştirilebilir ve genişletilebilir halde tutmak için tasarım prensipleri uygulanmaktadır.

Bölüm 5:

Oluşturucu tasarım şablonu kategorisinde yer alan factory, abstract factory, builder, prototype ve singleton tasarım şablonları bu bölümde detaylı olarak incelenmektedir.

Bölüm 6:

Yapısal tasarım şablonu kategorisinde yer alan adapter, bridge, facade, decorator, composite, flyweight ve proxy tasarım şablonları bu bölümde detaylı olarak incelenmektedir.

Bölüm 7:

Davranışsal tasarım şablonu kategorisinde yer alan command, memento, strategy, iterator, state, chain of responsibility, mediator, observer, template method ve visitor tasarım şablonları bu bölümde detaylı olarak incelenmektedir.

Bölüm 8:

JEE tasarım şablonu kategorisinde yer alan MVC, front controller, data access object, business delegate, service locator ve intercepting filter tasarım şablonları bu bölümde detaylı olarak incelenmektedir.

Bölüm 9:

Bu bölümde

datamapper, business object, active record, message channel, message router, registry ve null object gibi diğer tasarım şablonları incelenmekedir.

Bölüm 10:

Bu bölümde kitapta yer alan birçok tasarım şablonu kullanılarak oluşturulan ve otel rezervasyon platformu ismini taşıyan örnek bir yazılım yer almaktadır. Üç katmanlı mimari üzerinde kurulu olan bu program bünyesinde, tasarım şablonlarının nasıl uygulanabileceği incelenmektedir.

Kitabın İçeriği Ne Degildir?

Bu kitabın amacı Java dilinde nasıl program yazıldığını öğretmek değildir! Java dilinde kendisini geliştirmek isteyen okuyuculara diğer Java kaynakları tavsiye edilmektedir. Kitapta yer alan örnekler Java dilinde hazırlanmıştır. Bu sebeple okuyucunun Java dilini biliyor olmasında fayda vardır. Kitapta yer alan Java örnekleri anlatımı kolaylaştırmak için basit tutulmuştur. Bu yüzden Java dil bilgisine sahip olmayan okurlar da örnekleri takip edebilirler.

Kitap Kim İçin Yazıldı?

Bu kitap tasarım şablonlarını öğrenmek isteyen yazılımcılar için hazırlanmıştır. Bu kitapta yer alan tasarım şablonları isimleri itibariyle bir kelime hazinesi olarak düşünülebilir. Bu kelime hazinesi yazılımcıların kendi aralarındaki iletişimi kolaylaştırıcı niteliktedir. Aynı kelime hazinesine sahip yazılımcıların ortak çalışmaları daha verimli hale gelecektir.

Yazar Hakkında

İsmim Özcan Acar. 1974 İzmir doğumluyum. İlk ve orta öğrenimimi İzmir'de tamamladıktan sonra Almanya'da bulunan ailemin yanına gittim. Doksanlı yılların sonunda Almanya'nın Darmstadt şehrinde bulunan FH Darmstadt üniversiteden bilgisayar mühendisi olarak mezun oldum. 2001 senesinde ilk kitabım Perl CGI, 2008 senesinde Java Tasarım Şablonları ve Yazılım Mimarileri isimli ikinci kitabım, 2009 yılında Extreme Programming isimli üçüncü kitabım Pusula tarafından yayımlanmıştır.

PratikProgramci.com bünyesinde Pratik Spring, Pratik Agile ve Pratik Git isimlerinde kitaplarım bulunmaktadır.

KurumsalJava.com, Mikrodevre.com, SmartHomeProgrammer.com ve DevOnBike.com adresleri

altında bloglar yazıyorum.

Kitap Nasıl Okunmalı?

Eğer temel tasarım prensipleri, nesneye yönelik programlama ve UML (unified modeling language) hakkında yeterli bilgiye sahip olduğunuzu düşünüyorsanız, doğrudan beşinci bölüme geçebilirsiniz. Bu bölümden itibaren kitabın ana konusunu oluşturan tasarım şablonları incelenmektedir. Tasarım şablonları için gerekli temel bilgileri edinmek için kitabın ilk dört bölümünü okumanızı tavsiye ediyorum.

Yazar İle İletişim

Kitap ile ilgili sorularınızı acar@agilementor e-posta adresime gönderebilirsiniz. Benimle iletişim kurmadan önce lütfen BTSoru.com adresinde sorunuz hakkında araştırma yapınız. Bilgi paylaşımını geniş çaplı tutmak için okurlarımın sorularına BTSoru.com'da cevap vermeye çalışıyorum. BTSoru.com'da araştırma yaparken ya da soru sorarken soruların bu kitaba ait olduğunu görebilmek için lütfen pratik-design-patterns etiketini kullanın.

PratikProgramci.com

PratikProgramci.com programcılar için hizmet veren bir dijital eğitim platformudur. Bu kitap sadece PratikProgramci.com bünyesinde ekitap olarak satılmaktadır.

PratikProgramci.com bünyesinde programcılar için kaynak kitaplar yanı sıra görsel eğitim setleri de bulunmaktadır. Gelişmeleri http://www.pratikprogramci.com adresinden takip edebilirsiniz.

Kitapta Yer Alan Kod Örnekleri

Kitapta kullanılan kod örneklerini Maven projesi halinde http://www.pratikprogramci.com/?wpdmact=process&did=MjEuaG90bGluaw== adresinden edinebilirsiniz.

1. Bölüm

Tasarım Şablonları Giriş

Tasarım Şablonu Nedir?

Yazılım esnasında tekrar eden sorunları çözmek için kullanılan ve tekrar kullanılabilir yapıda kod yazılımını destekleyen, bir ya da birden fazla sınıftan oluşmuş modül ve program parçalarına tasarım şablonu (design pattern) ismi verilir. Tasarım şablonları programcılar tarafından edindikleri tecrübeler doğrultusunda oluşturulmuş kalıplardır. Bu kalıplar sorunu tanımlıyarak, çözümü için gerekli atılması gereken adımları ihtiva ederler. Kullanıcısı kalıbı tanımlanmış sorunu çözmek için tekeri tekrar icat etmek zorunda kalmadan kullanabilir.

Tasarım şablonları aşağıda yer alan ortak özelliklere sahiptirler:

- Edinilen tecrübeler sonunda ortaya çıkmışlardır.
- Tekerin tekrar icat edilmesini önlerler.
- Tekrar kullanılabilir kalıplardır.
- Ortak kullanılarak daha büyük problemlerin çözülmesine katkı sağlarlar.
- Devamlı geliştirilerek, genel bir çözüm olmaları için çaba sarfedilir.

Uygulama bakımı ve geliştirilmesi için ilk yazılım sürecinden daha cok enerji sarfedilir. Bu yüzden yazılım esnasında esnek bir yapının ve mimarinin oluşturulmasına dikkat edilmesi gerekmektedir. Esnek mimariler için değişik türde tasarım şablonları kullanılabilir. En basit ve uygulaması kolay bir tasarım şablonunun kullanılması, hiçbir tasarım şablonu kullanılmamasından daha iyidir. İyi bir yazılım mühendisi olabilmek için tasarım şablonları ve kullanım alanları hakkında ihtisas yapmış olmak gerekmektedir.

Tasarım Şablonu Neden Kullanılır?

Her tasarım şablonunun belirli bir ismi vardır ve bu isim kullanıldığı zaman hangi tasarım şablonundan bahsedildiği hemen anlaşılır. Bu sebepten dolayı yazılım ekibinin kullanacağı ortak bir kelime hazinesi oluşur. Programcılar takım içinde tasarım şablonlarının isimlerini kullanarak, hangi sorunlar üzerinde çalıştıklarını kolaylıkla anlatabilirler. Bu durum ayrıca takım içinde tasarım şablonlarını tanımayan programcılar için duydukları tasarım şablonlarını öğrenmeye yönlendirecek bir motivasyon kaynağı oluşturur. Tasarım şablonlarının kullanılması konsepsiyonel olarak bir üst seviyede çalışılmasını ve düşünülmesini sağlar. Nesneler seviyesinde sorunları çözmek her zaman kolay olmayabilir, lakin tasarım kalıpları seviyesinde düşünüldüğü zaman, problem çözüm işlemi kolaylaşır.

Tasarım Şablonları Nasıl Kullanılmalıdır?

Tasarım şablonlarını uygulamak için mümkün olan her fırsatı değerlendirmek ya da fırsat aramak yerine, öncelikle tasarım şablonu uygulayışının ne kadar gerekli olduğu sorgulanmalıdır. Her şeyin fazlasından zarar geldiği gibi, yerli, yersiz kullanılan tasarım şablonları da uygulamanın mimarisi ve tasarımı için zararlı bir hale gelebilirler. Bunun önüne geçmek amacıyla gereklilik ve ihtiyaç itina ile tartılmalıdır.

Çoğu tasarım şablonu mevcut kodu değiştirmeden uygulamaya yeni davranış biçimleri ekleme amacına hizmet etmektedir. Eğer sürekli mevcut kodu değiştirerek, uygulamaya yeni özellikler kazandırmak zorundaysanız, belki bir tasarım şablonu kullanma vakti sizin için gelmiş olabilir.

Öncelikli amaç her zaman uygulamanın önünü açık tutmak ve gereksiz yapılar oluşturmamak olmalıdır. En iyi çözüm en basit olanıdır. Bir tasarım şablonu uygulandığında, uygulamanın genel karmaşa (complexity) seviyesinin arttığı göz ardı edilmemelidir. Eğer tasarım şablonu kullanmadan daha basit ve sade bir çözüm münkün ise, tasarım şablonu kullanımı reddedilmelidir.

Kitapta Yer Alan Tasarım Şablonları

Bu kitapta yer alan tasarım şablonları beş ana gruba ayrılmıştır. Bu gruplar:

Oluşturucu tasarım şablonları

- Factory
- Factory Method
- Abstract Factory
- Singleton
- Builder
- Prototype
- Object Pool

Yapısal tasarım şablonları

- Class Adapter
- Object Adapter
- Bridge
- Facade
- Composite
- Decorator
- Virtual Proxy

- Dinamic Proxy
- Protection Proxy
- Remote Proxy
- Flyweight

Davranışsal tasarım şablonları

- Command
- Iterator
- Memento
- State
- Observer
- Strategy
- Chain Of Responsibility
- Mediator
- Visitor
- Template Method
- Interpreter

Java EE tasarım şablonları

- Model View Controller
- Front Controller
- Data Access Object
- Business Delegate
- Service Locator
- Intercepting Filter
- Business Object

Diğer tasarım şablonları

- DataMapper
- RequestMapper
- ResponseMapper
- Active Record
- Message Channel
- Message Router
- Registry
- Null Object

- Dependeny Injection

2. Bölüm

Nesneye Yönelik Programlama - Object Oriented Programming

Günümüzde hemen hemen her dilin ihtiva ettiği metotlar belli bir işleve hizmet eden iş mantığını (business logic) kapsüllemek için kullanılmaktadırlar. Geniş tabanlı uygulamalarda oluşan kodun mantıklı bir cerçevede organize edilebilmesi için sadece metotların kullanımı yeterli değildir.

Bir metot bünyesinde iş mantığını belli kalıplara sokmak mümkün değildir. Buna metotların kodsal ilişkilerin ifadesindeki yeterliliğini de eklediğimizde, sadece metotlar ya da fonksiyonlar aracılığı ile kodun organize edilmesi işlemi çıkmaz bir sokak haline gelecektir.

Kodsal ilişkileri ifade etmek ve belli kalıplar oluşturabilmek için metotlar haricinde başka bir mekanizmaya daha ihtiyaç duymaktayız. Böyle bir mekanizmayı nesneye yönelik programlama (object oriented programming) tekniği sunmaktadır.

Nesneye yönelik programlama tekniğinde kod birimlerini sınıf olarak isimlendirilen şablonlar aracılığı ile şekillendirebilmekteyiz. Bu şablonlardan nesne olarak ifade edilen yapılar oluşturulmaktadır. Nesneleri canlı varlıklar olarak düşünebiliriz. Her canlı varlığı bir birey olarak düşünecek olursak, bireyler davranış ve sahip oldukları iç dünyaları ile birbirlerinden farklı olacaklardır. Nitekim bir sınıftan oluşturulan iki nesne değişik hafıza alanlarında yer aldıklarından, farklı kimliklere sahip olacaklardır.

Nesneye yönelik programlama tekniğinin bir uygulamayı geliştirmek için nasıl kullanıldığını küçük bir örnek üzerinde göstermek istiyorum. Müşterimiz ürün satmak amacıyla bir shop sistemine ihtiyaç duymaktadır. Müşteri ile yaptığımız görüşmelerde aşağıdaki terimlerin kullanıldığını fark ettik:

- Müşteri
- Ürün
- Sipariş
- Sepet
- Ödeme
- İade
- Adres

Bu listede yer alan her kelime uygulamanın nasıl şekillendirilmesi gerektigi hakkında bize ip ucu vermektedir. Nesneye yönelik programlamada gerçek hayattaki yapıları modellemek için sınıflar kullanılmaktadır. Bu listede yer alan her yapıyı birer sınıf olarak modelleyebiliriz. Kod 1 de müşteri teriminin sınıf olan karşılığı yer almaktadır.

```
// Kod 1

public class Musteri{
```

```
    private String isim;
    private String soyad;
}
```

Bir nesneyi nesne yapan bilgiler sınıf bünyesinde sınıf değişkenlerinde tutulmaktadır. Örneğin kod 1 de bir müşteriyi temsil etmek amacıyla isim ve soyad gibi değişkenler tanımlanmıştır. Bu sınıftan bir müşteri nesnesini şu şekilde oluşturabiliriz:

```
// Kod 2

Musteri müsteri = new Musteri();
```

New operatörü ile yeni bir müşteri nesnesi oluşturulmaktadır. Bu nesne hafızanın herhangi bir yerinde konuşlandırılmaktadır. Programcı olarak nesnenin hafızaya yerleştirilme işlemine müdahale etme şansımız yoktur. Sadece bir referans aracılığı ile bu nesne olan bağı koruyabiliriz. Kod 2 de bu referans musteri ismini taşımaktadır. Birden fazla referans aynı nesneye işaret etmek için kullanılabilirler. Kod 3 de yer alan örnekte musteri1 ve musteri2 nesne referanslarıdır ve new ile oluşturulan aynı nesneye işaret etmektedirler.

```
// Kod 3

Musteri musteri1  = new Musteri();
Müşteri musteri2 = musteri1;
```

Herhangi bir sınıfdan bir nesne oluşturmak için sınıfın ismini kullanmak zorundayız. Kod 3 de kullandığımız sınıf ismi Musteri dir. Kod 3 de yer alan musteri1 isimli referansı bir değişken olarak düşünecek olursak, bu değişkeni tanımlamak için kullandığımız sınıf bir veri tipi haline gelmektedir. Sınıflar programcılar tarafından kullanılan dili genişletmek için oluşturulan veri tipleridir. Örneğin int ya da String nasıl bir veri tipi ise ve bir değişken tanımlamak için kullanılabiliyorsa, Musteri de bir veri tipidir ve müşteri nesnelerini temsil eden değişkenler tanımlamak için kullanılmaktadır.

Sınıflar değişkenler haricinde metotlara sahip olabilirler. Bu metotlar sınıf değişkenlerine olan erişimi sağlamak ve sınıf değişkenlerinin sahip oldukları değerleri değiştirmek için kullanılabilirler. Bu metotlar bünyesinde ayrıca iş mantığı implemente edilir. Kod 4 de yer alan örnekte isim değişkenine olan erişim getIsim(), isim değişkeni üzerindeki değişiklikler setIsim() metodu aracılığı ile yapılmaktadır.

```
// Kod 4

public class Musteri{
```

```
    private String isim;
    private String soyad;

    public String getIsim(){
        return this.isim;
    }

    public void setString(String isim){
        this.isim = isim;
    }
}
```

Burada nesneye yönelik programlamanın temel bir prensibini görmekteyiz. Nesnelerin iç dünyalarını oluşturan sınıf değişkenlerine doğrudan erişim mümkün olmamalıdır. Aşağıdaki yapı nesneye yönelik programlamaya aykırı bir durumdur, çünkü sınıf değişkenleri public olarak tanımlanmıştır ve bu değişkenler doğrudan erişim ile değiştirilebilirler.

```
// Kod 5

public class Musteri{
    public String isim;
    public String soyad;
}

Musteri musteri = new Musteri();
musteri.isim = "Ali";
```

Nesneler iç dünyalarını yani sahip oldukları değişken değerlerini koruma hakkına sahiptirler. Bu değişkenlere olan erişim kontrollü bir şekilde yapılmalıdır. Kod 6 da bu erişimin sınıf metotları aracılığı ile nasıl yapıldığını görmekteyiz.

```
// Kod 6

Musteri musteri = new Musteri();
String isim = musteri.getIsim();
String yeniIsim = "Ahmet";
musteri.setIsim(yeniIsim);
```

Nesne oluşturmak için kullanılan sınıf metotlarına konstrüktör (constructor) ismi verilmektedir. Her sınıf bünyesinde parametresiz bir standart konstrüktör bulunmaktadır. Bu yüzden kod 6 da new operatörü ile yeni bir Müşteri nesnesi oluşturabildik.

```
// Kod 7
```

```
public class Musteri{
    private String isim;
    private String soyad;

    public Musteri(String isim){
        this.isim = isim;
    }
}
```

Kod 7 de Müşteri sınıfı bünyesinde standart konstrüktör yanı sıra ikinci bir konstrüktör tanımladık. Bu konstrüktör aldığı parametre ile yeni bir Musteri nesnesi oluşturmaktadır. Bu konstrüktörün kullanımı kod 8 de yer almaktadır:

```
// Kod 8

Musteri musteri = new Musteri("Ahmet");
```

Mevcut bir nesne referansına null değerini atayarak, hafızadan silinmesini sağlayabiliriz.

```
// Kod 9

Musteri musteri = new Musteri();
müşteri = null;
```

Hafızada bulunan bir nesne kendisine işaret eden bir referans olduğu sürece hafızada kalır. Aksi taktirde sistem tarafından hafızadan silinir.

Sınıflar arası ilişkiler sınıflar içinde kullanılan sınıf değişkenleri aracılığı ile oluşturulabilir. Aşağıdaki örnekte Musteri sınıfı Adres sınıfını kullanmaktadır.

```
// Kod 10

public class Musteri{
    private Adres adres;
}
```

Bir sınıfın başka bir sınıfı genişleterek, o sınıf bünyesindeki tüm değişkenleri ve metotları miras olarak alması mümkündür. Buna nesneye yönelik programlama terminolojisinde kalıtım (inheritance) ismi verilmektedir. Aşağıdaki örnekte Ford Araba sınıfını genişletmektedir.

```
// Kod 11
```

```
public class Araba{

    private String model;
    private String marka;

    public Araba(String marka, String model){
        this.marka = marka;
        this.model = model;
    }

    public String getModel(){
        return this.model;
    }

    public String getMarka(){
        return this.marka;
    }
}

public class Ford extends Araba{

    private List<Aksesuar> akseruarlar;

    public Ford(String model, List<Aksesuar> akseruarlar){
        super("Ford", model);
        this.aksesuarlar = aksesuarlar;
    }
}
```

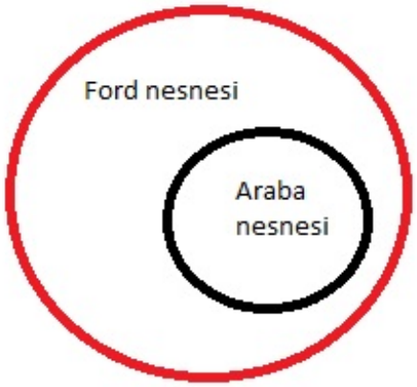
Ford Araba sınıfını genişleterek, Araba sınıfında yer alan yapıları kendi bünyesinde kullanma hakkına sahip olmaktadır. Ford Araba bünyesinde tanımlı her public ve protected metodu kendi metoduymuş gibi yeniden tanımlamak zorunda kalmadan kullanabilmektedir. Aynı şey public ve protected olan değişkenler için de geçerlidir. Bu mekanizmanın nasıl işlediğini kod 12 de görmekteyiz.

```
// Kod 12

Aksesuar aksesuar = new Aksesuar("Deri Koltuk");
List<Aksesuar> aksesuarlar = new ArrayList<Aksesuar>();
aksesuarlar.add(aksesuar);
Ford ford = new Ford("Fiesta", aksesuarlar);
String marka = ford.getMarka();
```

Ford sınıfı bünyesinde getMarka() isminde bir metot bulunmamaktadır. Ama kendisi Araba sınıfını

genişlettiği ve bu metot Araba sınıfı bünyesinde public olduğu için bu metodu kullanabilmektedir. Eğer getMarka() metodu private olmuş olsaydı, o taktirde ford.getMarka() kullanımı mümkün olmazdı.

Kod 12 de bir Ford nesnesi oluşturmak için sınıf konstrüktöründe super() metodunun kullanıldığını görmekteyiz. Oluşturulan her Ford nesnesi bünyesinde bir adet Araba nesnesi yer alacaktır. Sadece bu şekilde Ford nesnesi kendi bünyesinde olmayan metot ve değişkenleri bünyesinde barındırdığı Araba nesnesi aracılığı ile kullanabilir. Bu yüzden Ford bünyesindeki konstrüktörün ihtiyaç duyulan üstsınıf nesnesini super() ile oluşturması gerekmektedir. Super() metodunu üstsınıf konstrüktörünün koşturulması olarak düşünebiliriz. İki nesne arasındaki ilişki resim 1 de yer almaktadır.



Resim 1

Ford sınıfı miras olarak aldığı metotları yeniden implemente edebilir. Bu sayede üstsınıfta tanımlı olan davranış biçimlerini (metotlar) altsınıfta değiştirmek ve altsınıfa has hale getirmek mümkündür. Kod 13 de Ford sınıfı getMarka() metodunu yeniden yapılandırmıştır.

```
// Kod 13

public class Ford extends Araba{

    private List<Aksesuar> akseruarlar;

    public Ford(String model, List<Aksesuar> akseruarlar){
        super("Ford", model);
        this.aksesuarlar = aksesuarlar;
    }

    @Override
    public String getMarka(){
        System.out.println("Marka: " + super.getMarka());
```

```
        return super.getMarka();
    }
}
```

getMarka() metodu her iki sınıfda da bulunmaktadır. Hangi metodun koşturulacağına oluşturulan nesneye işaret eden referans veri tipi aracılığı ile karar verilmekedir.

```
// Kod 14

Araba araba = new Araba();
araba.getMarka();
```

Kod 14 de Araba sınıfı bünyesindeki getMarka() metodu kullanılmaktadır.

```
// Kod 15

Araba araba = new Ford();
araba.getMarka();
```

Kod 15 de Ford bünyesindeki getMarka() metodu kullanılmaktadır. Eğer kod 13 deki gibi getMarka() metodu Ford bünyesinde reimplemente edilmemiş olsaydı, bu durumda Araba bünyesindeki getMarka() koşturulmuş olurdu. Nesneye yönelik programlama dillerinde bu işleve polimorfizm ismi verilmektedir. Polimorfizm önceden hangi metodun koşturulması gerektigi bilgisine sahip olmadan, kullanılan veri tipine bağımlı metot seçimi yöntemidir.

Java gibi Nesneye yönelik programlama dillerinde sınıflar soyut olarak tanımlanabilmektedir. Bunun bir örneği aşağıda yer almaktadır:

```
public abstract class Araba{
}
```

Soyut olan sınıflardan nesneler oluşturmak mümkün değildir. Bu sınıflar sadece genişletilebilirler ve sınıfların ortak yanlarını soyut olan üst sınıflarda toplamak için kullanılırlar.

Soyut sınıflar yanı sıra soyut metotlar da tanımlamak mümkündür. Bunun bir örneğini kod 16 görmekteyiz:

```
// Kod 16

public abstract class Araba{

    public abstract void startEngine();
```

```
}
```

Üstsiniflarda soyut olarak tanımlanan sınıfların altsınıflarca implemente edilme zorunluluğu mevcuttur. Bu şekilde belli davranış biçimlerini üstsınıflarda soyut olarak tanımlamak ve altsınıfda somut olarak implemente etmek mümkündür. Kod 16 da yer alan örnekte üstsınıf sadece startEngine() ile motor çalıştırma işlemini soyut olarak tanımlamıştır. Altsınıflar bu metodu kendilerine has bir şekilde implemente ederler.

Bazı nesneye yönelik programlama dillerinde çoklu kalıtım mümkün iken, Java'da bu örneğin mümkün değildir. Java'da her sınıf sadece bir üstsınıfa sahip olabilirken birden fazla interface sınıfı implemente edebilir.

Interface sınıflar soyut sınıflar gibidirler. Gerçek soyut sınıflar soyut metotlar yanı sıra somut metotlar da ihtiva edebilirken, interface sınıflar bünyesinde sadece soyut metotlar yer alabilir. Java 8 ile bu da değişmiş ve interface sınıflar bünyesinde default method olarak tanımlanan somut metotlar oluşturmak mümkün hale gelmiştir. Kod 17 de bir interface sınıfı örneği yer almaktadır.

```
// Kod 17

public interface Araba{

    public void start();
    public void stop();
}
```

Kod 17 de Araba sınıfını bir interface olarak tanımladık. Bu sınıfın start() ve stop() isminde iki soyut metodu bulunmaktadır. Kod 18 de yer alan Ford sınıfı bu interface sınıfı implemente etmektedir.

```
// Kod 18

public class Ford implements Araba{

    public void start(){
       System.out.println("started");
    }

    public void stop(){
       System.out.println("stopped");
    }
}
```

Ford kendi bünyesinde Araba interface sınıfında yer alan metotları implemente etmek zorundadır. Bu durumda bir Ford nesnesi bir Araba nesnesi haline gelmektedir ve bu nesneyi şu şekilde de tanımlamak mümkündür:

```
// Kod 19

Araba araba = new Ford();
araba.start();
```

Java dilinde bir interface sınıfın başka bir interface sınıfı genişletmesi de mümkündür. Bunun bir örneğini kod 20 de görmekteyiz:

```
// Kod 20

public interface Arac{
}

public interface Araba extends Arac{
}

public class Ford implements Araba{
}
```

Kod 20 de yer alan Ford sınıfı hem Araba hem de Arac bünyesindeki tüm metotları implemente etmek zorundadır. Eğer Arac ve Araba kod 21 de ki gibi birbirlerinden bağımsız olsalardı, Ford sınıfı bu iki interface sınıfı bu şekilde implemente edebilirdi:

```
// Kod 21

public interface Arac{
}

public interface Araba {
}

public class Ford implements Arac, Araba{
}
```

Bu şekilde çoklu kalıtım olmasa da Java'da birden fazla interface sınıf implemente edilebilmektedir.

Şimdi interface ve soyut sınıfları daha yakından inceleyelim.

Interface Nedir?

Java dilinde interface adını taşıyan, tasarım şablonlarında ve modellemede kullanılan sınıflar tanımlamak mümkündür. Bir interface normal bir Java sınıfından farksız bir şekilde tanımlanır. Sınıf tanımlanırken class yerine interface terimi kullanılır. Bir interface sınıf örneğini kod 22 de görmekteyiz.

```
// Kod 22

public interface Tasit {

    public String getMarka();

}
```

Java'da tanımlanmış bir interface sınıfdan normal bir sınıfta olduğu gibi new() operatörü ile bir nesne oluşturulamaz! Bu neden mümkün değildir?

Bunu gerçek hayattan bir örnek vererek açıklamaya çalışalım.

Çalıştığım firma tarafından bana mesai saatlarinde kullanılmak üzere bir araç tahsis edildi. Bu aracı kullanarak, müşteri görüşmelerine gidiyor ve firmanın diğer işlerini takip ediyorum. Araç bana verilmeden önce ehliyetim olduğunu firmaya bildirdim. Şimdi geriye dönelim ve seneler önce sürücü kursunda aracı kullanmak için neler öğrendiğimi gözden geçirelim.

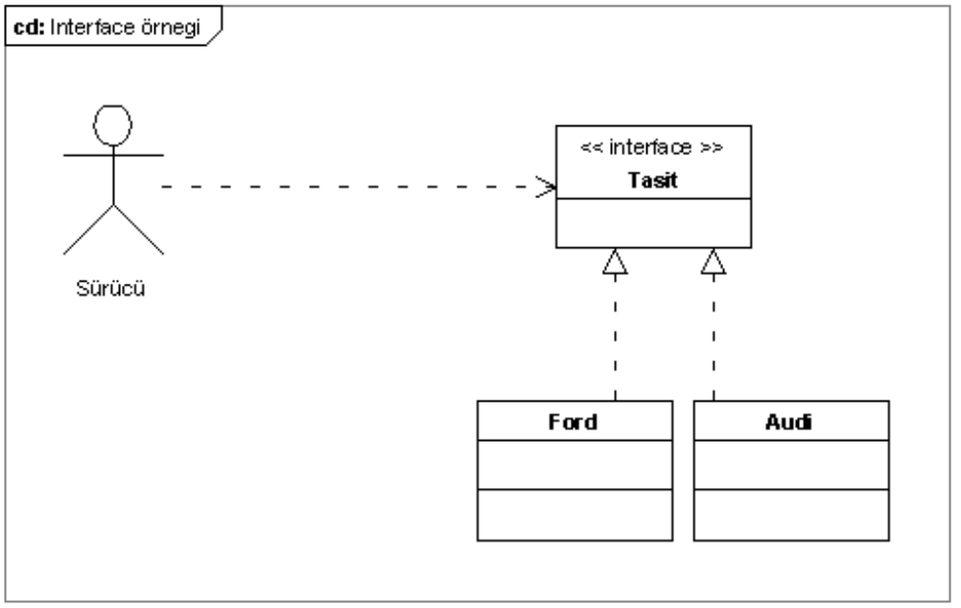
Sürücü kursunda ilk öğrendiğim şey bir aracı kullanmak için fren, debriyaj, vites vb. parçalarının olduğuydu. Aracı hareket ettirmek için vitesi 1 e takıp, hafif gaz vererek, yavaş yavaş sol ayağımı debriyajdan çekmem gerektiğini öğrendim. Araç hereket ettikten ve belirli bir hıza ulaştıktan sonra, ayağımı gazdan çekip, sol ayağımla debrijaya basarak, ikinci vitese geçilmesi gerektiğini öğrendim. Aracı durdurmak için frene basman gerektiği gösterildi. Zaman içinde aracın nasıl hareket ettirilip, durdurulacağını iyice anladım ve sürücü sınavını kazanarak, ehliyetimi aldım.

Ehliyet sahibi olduğumda bildiğim bir şey vardı, oda vitesli araçların hepsinin aynı şekilde kullanıldığı! Hangi araç olursa olsun, vitesi, freni, debriyajı, gaz pedalı olduğu sürece kullanabilirdim, çünkü her araç üreticisi mutlaka ve mutlaka, ürettiği her vitesli araçta bu parçaları benim bildiğim şekilde yerleştirecekti. Arabanın markasının benim için bir önemi yoktu, Ford'da olabilirdi, Fiat'da.

Firmam tarafından bana verilen ilk araç Ford marka bir binek oto idi. Hiç zorluk çekmeden bu aracı kullanarak, firmanın işlerini takip ettim. Kısa bir zaman sonra Audi A4 tipinde bir araca

geçtim. Ford ve Audi arasında kullanım açısından bir farklılık olmadığından, zorluk çekmeden Audi'yi kullanmaya başladım.

Bu örnekte dikkatimizi çeken bir nokta bulunuyor. Bunu daha sonra Java interface sınıfları ile ilişkilendirecegiz. Ben araç sürülmesi için gerekli işlemleri ve kullanılacak metotları biliyorum. Her tip vitesli binek otoyu kullanabilirim. Sürücü kursunda öğrendiklerim her vitesli binek oto için geçerli. Buradan şu sonucu çıkarıyoruz: binek oto kullananlar ve üreten firmalar arasında, bir nevi görünmez ama bilinen bir anlaşma yapılmış. Bu anlaşmaya göre sürücü kursunda ehliyet edinmek isteyenlere, aracın nasıl kullanılacağı öğretiliyor. Sürücü kursunda sürücü adayına dolaylı verilen bir söz var: „Bu aracı kullanmayı öğrendiğin taktirde, diğer araçlarıda bu şekilde, başka bir şey öğrenmene gerek kalmadan kullanabileceksin!"



Resim 2

Binek oto üreticileri, sürücülere verilen bu sözü bildiklerinden (bu bir nevi anlaşma), üretilen araçlar üzerinde durup, dururken bir değişiklik yapıp, örneğin vitesi ya da fren pedalını bagaja koymuyorlar :) Bu böyle olsa idi, ehliyeti olan bir şahıs bu aracı nasıl kullanırdı? Kullanabilse, trafik emniyeti açısından bu iyi olur muydu? Araç yapısı değistirildiğinde bunun gibi sorulacak birçok soru var. Hiçbir binek oto üreten firma böyle bir değişiklik yapmayacağından, sürücüye verilen söz (bu interface sınıfının metotlarıdır) de hiçbir zaman bozulmamış oluyor.

Bu açıklamanın ardından tekrar başa dönelim ve neden bir interface sınıfından new operatörü ile bir nesne oluşturulamıyacağına göz atalım. Bir interface sınıfında sadece metotlar deklare edilir. Bu metotların gövdeleri boştur. Altsınıflar bu metotların gövdeleri için gerekli kodu oluştururlar. Metot gövdesi olmayan bir interface sınıftan nesne oluşturulamaz, çünkü sadece metot

deklarasyonlarına sahip bir nesnenin hiçbir işlevsel görevi olamaz!

Bu örnekten sonra bir Interface sınıfın ne işe yaradığına bakalım. UML diyagramında görüldüğü gibi sürücü sadece Tasit sınıfını tanıyor. Tasit interface olarak tanımlanmış.

```
// Kod 23

public interface Tasit  {
    public String getMarka();
}
```

Ford ve Audi isminde iki sınıf bulunuyor. Bu sınıflar Tasit interface sınıfını implemente ediyorlar. Bu iki implementasyon sınıfı bünyelerinde getMarka() isminde bir metot oluşturup, bu metodu implemente etmek (metot gövdesi için gerekli kodu oluşturmak) zorundalar.

```
// Kod 24

public class Audi implements Tasit{

    public String getMarka() {
        return "Audi A4";
    }
}
```

Herhangi bir Java sınıfı implements direktifini kullanarak, bir interface sınıfının sahip olduğu metotları implemente edebilir. İmplementasyon ile interface sınıfının tanımladığı (örneğin getMarka()) metot gövdelerinin altsınıfta (yukarıdaki örnekte Audi sınıfı) kodlanmasını kastediyoruz. Buna göre interface sınıflarında sadece metotlar deklare edilebilir ve implemente edilemez. İmplementasyonu altsınıflar üstlenir.

```
// Kod 25

public class TasitTest2{

    public static void main(String args[]){
        Tasit tasit = new Audi();
        System.out.println(tasit.getMarka());
    }
}
```

TasitTest2 bünyesinde Tasit.getMarka() metodunu kullanılıyor. Bu metot Audi ve Ford sınıflarında implemente edildiği için new Audi() ya da new Ford() ile TastTest2 sınıfına Tasit interface sınıfının

verdiği sözü tutan bir sınıftan elde edilmiş bir nesneyi verebiliriz. Burada „verdiği sözü tutan“ kelimelerinin altını çizmek istiyorum. Bir interface aslında dış dünyaya verilen bir söz metnidir. Bir interface içinde metotlar tanımlar. İnterface i kullanan diğer sınıflar ise bu metotları kullanarak, gerekli işlemleri yaparlar. İşlemlerin nasıl gerçekleştirildiği, bunun için hangi implementasyon sınıflarının kullanıldığı, kullanıcı için önemli değildir. Analog bir şekilde sürücü kursu ve firma aracı örneğine dönecek olursak, interaface ve bu örnek arasındakı paralellikleri görmüş oluruz. Ben firmama, „ehliyetim var, binek oto kullanabilirim“ dedim. Bu şu anlama geliyor: ben Tasit interface inde yer alan vitesDegistir(), gazVer(), frenle() gibi metotları tanıyorum ve kullanıyorum. Firmam bana soyut olan Taşıt veremez, 4 tekerleği, vitesi, gaz ve fren pedalı olan bir binek oto vermek zorunda. Bana firmam tarafından verilen araç Ford marka bir binek oto. Ford firması Taşıt interface ini tanıdığı için içinde yer alan vitesDegistir(), gazVer(), frenle() gibi metotların yazılımını yapıyor, yani Tasit interface in dış dünyaya vermiş olduğu sözleri (bunlar sahip olduğu metotlar) yerine getiriyor. O halde benim ihtiyaç duyduğum metotlar Ford marka binek otoda da var olduğu için (çünkü Tasit interface sınıfını implemente etmiştir), bu aracı kullanabiliyorum.

Ford sınıfının, Tasit interface sınıfı tarafından verilen tüm sözleri tutabilmesi için Tasit interface sınıfında bulunan tüm metotları impelemente etmesi gerekiyor. İşte bu sebepten dolayı tasit.getMarka() metodunu kullanarak, aracın markasını ögreniyorum. Bu araç bir Audi olduğu için „Audi A4“ şeklinde aracın markası bana bildiriliyor. Audi sınıfının Tasit interface tarafından sunulan getMarka() metodunu implemente etmesi, yani metot gövdesini oluşturması gerekiyor. Aksı taktirde Audi bir taşıt olamaz.

Java interface sınıflarının kullanıcı sınıflar için bir nevi sözleşme metni oluşturduğunu gördük. Bir interface sadece dış dünyaya sunulacak hizmetleri tanımlar. Bu hizmetlerin nasıl yerine getirileceğine interface sınıfını implemente eden altsınıflar karar verir. Interface kullanıcısı genelde hangi altsınıf üzerinden gereken hizmeti aldığını bilmez, bilmek zorunda değildir. Bu sayede „loose coupling“ olarak tabir edilen, servis sağlayıcı ve kullanıcı arasında gevşek bir bağımlılık oluşturulmuş olur. Bu yazılım mimarisi açısından büyük bir avantaj sağlamaktadır. Interface sınıflar kullanılarak, sisteme esnek ve ilerde servis kullanıcılarını etkilemeden eklemeler yapılabilir. Firma aracı örneğine dönecek olursak: Firmanın Ford ve Audi marka iki binek otosu bulunmaktadır. Kısa bir zaman sonra Fiat marka bir araç daha satın alınır. Burada yapılması gereken sadece Fiat isminde bir sınıf oluşturup, Tasit interface ini implemente etmektir. Bu şekilde sistemi, diğer bölümleri etkilemeden genişletmiş oluyoruz. Sürücüler sadece Tasit inteface ini tanıdıklarından, onlara firma tarafından hangi aracın verildiği önemli değildir.

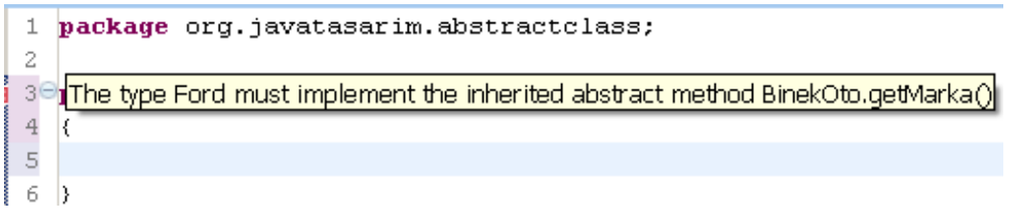
Kısaca interface sınıflar servis kullanıcılarından karmaşık yapıdaki altsınıfları gizli tutmak ve servis sunucusu ve sağlayıcı arasındaki bağımlılığı azaltmak için kullanılırlar. Bu kitapta yer alan bir çok tasarım şablonunda interface sınıflar büyük rol oynamaktadır.

Soyut (Abstract) Sınıf Nedir?

Ortak özellikleri olan nesneleri modellemek için Java dilinde soyut sınflar kullanılırlar. Soyut sınıflardan interface sınıflarında olduğu gibi somut nesneler oluşturulamaz.

```
// Kod 26

public abstract class BinekOto {
    private String marka;
    private int ueretimYili;
    private int vitesAdedi;

    public abstract String getMarka();

    public int getUeretimYili(){
        return this.ueretimYili;
    }
}
```

Kod 26 da yer alan BinekOto.java sınıfı bir soyut sınıftır. getMarka() isminde bir soyut metodu ve getUeretimYili() isminde normal bir metodu bulunmaktadır. getMarka() soyut olduğundan, altsınıfların bu metodu mutlaka implemente etmeleri gerekmektedir. Aksı taktirde Java derleyicisi (compiler) aşağıda görüldüğü şekilde hata mesajı verecektir.



Resim 3

Ford isminde BinekOto sınıfındaki özellikleri devralan bir altsınıf tanımlıyoruz. Bunun için Java dilinde extend direktifi kullanılır.

```
// Kod 27

public class Ford extends BinekOto{

    public String getMarka() {
        return "Ford Focus";
    }
```

```
    public int getUeretimYili(){
        System.out.println(super.getUeretimYili());
        return super.getUeretimYili();
    }
}
```

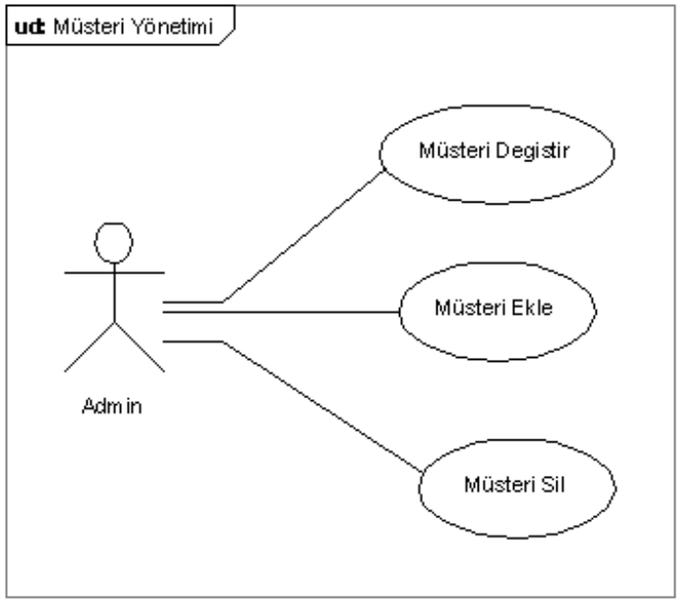
BinekOto sınıfında getMarka() soyut olarak tanımlandığı için altsınıflarda mutlaka getMarka() isminde implementasyonu yapılmıış bir metodun bulunması gerekmektedir. Ford sınıfı tanımlamak zorunda olmadan, BinekOto sınıfından getUeretimYili() metodunu kullanabilir. Java gibi nesneye yönelik programlama dillerinde ortak değerler bir anasınıfta toplanır ve altsınıflarda sadece altsınıf için gerekli ekleme ve değişiklikler yapılarak kullanılır. Altsınıflar istedikleri taktirde üstsınıfta tanımlanmış bir metodu kendi bünyelerinde değiştirebilirler (reimplementasyon).

Soyut sınıflar interface sınıfların aksine tecrübeli Java programcıları tarafından yazılımda tercih edilmezler. Karmaşık sınıf hirarşileri kodun bakımını ve genişletilmesini engeller. Bu problemi çözmek için çeşitli tasarım şablonları oluşturulmuştur. Bunlardan bir tanesi Composite tasarım şablonudur. Composite tasarım şablonu ile soyut sınıflar ve bu sınıfların altsınıflarını oluşturan impelentasyonları birbirinden ayırmak ve bu şekilde sınıflar arası bağımlılığı azaltmak mümkündür. Bunun nasıl yapıldığını ilerleyen bölümlerde göreceğiz.

Interface Örneği

Özellikle yazılım kitaplarında verilen teorik bilgilerin kod örnekleri ile açıklanması okuyucu için büyük önem taşımaktadır. Kendim programcı olduğum için sadece teorilerin yer aldığı bir kitabı okuyup, anlamanın zor olduğunu ve kitabı ilk bölümü okuduktan sonra okumaya devam etmenin ne kadar güç olduğunu biliyorum. Bu nedenle, bu kitapta okuyucuya anlatmak istediğim konuları örnekler ile açıklamaya çalıştım.

Interface sınıflar hakkında okuduklarımızı bir örnek yazılım ile tekrar gözden geçirelim. Bir firmanın müşterilerini veri tabanında tutmak istediğimizi düşünelim. Yapılması gereken işlemler aşağıda yer alan UML usecase diyagramında gösterilmiştir. Programımız yeni bir müşteriyi veri tabanına ekler, mevcut bir müşteriyi silebilir veya müşteri bilgilerini değiştirebilir.



Resim 4

Uygulamamızı gelecekte oluşabilecek teknolojik değişikliklerden etkilenmeyecek bir şekilde tasarlamak istiyoruz. Musteri isminde bir sınıf tanımladık. Müşteri için gerekli tüm bilgiler bu sınıf içinde tanımlandı.

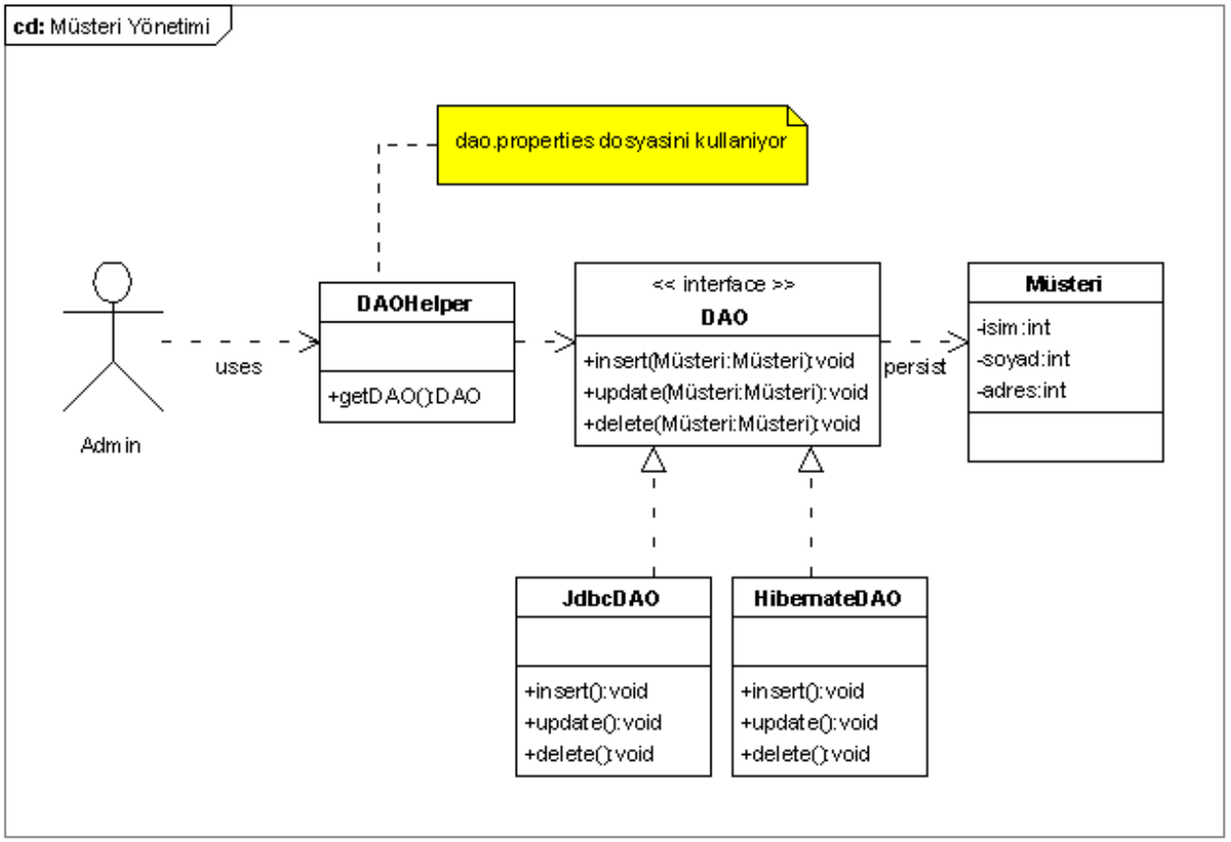
```
// Kod 28

public class Musteri {
    private String isim;
    private String soyad;
    private String adres;

    public String getAdres()
    {
        return adres;
    }
    public void setAdres(String adres)
    {
        this.adres = adres;
    }
    public String getIsim()
    {
        return isim;
    }
    public void setIsim(String isim)
    {
        this.isim = isim;
```

```
    }
    public String getSoyad()
    {
        return soyad;
    }
    public void setSoyad(String soyad)
    {
        this.soyad = soyad;
    }
}
```

Bu sınıf içinde yer alan bilgiler veri tabanında musteri tablosuna eklenecektir. Müşteri bilgilerini veri tabanına kayıt edebilmek için kullanabileceğimiz iki yöntem mevcuttur:

- JDBC – Doğrudan veri tabanı sürücü (JDBC Driver) üzerinden veri tabanına bağlanıp insert, delete, update SQL komutlarını kullanabiliriz
- Hibernate ya da Ibatis gibi bir çatı kullanarak, JDBC kodu yazmaya gerek kalmadan, Musteri Java sınıfından olan nesneleri veri tabanına ekliyebiliriz.

Şu an için Hibernate hakkında yeterli bilgimiz olmadığından dolayı, JDBC yöntemini kullanmayı tercih ediyoruz. Proje yöneticimiz esnek bir mimari tasarlamamızı istiyor. Tasarımını yapacağımız müşteri yönetimi programı gelecekte sorun çıkmadan JDBC yerine Hibernate ve ya Ibatis kullanabilmelidir. Buradan yola çıkarak resim 5 de yer alan tasarımı yapıyoruz.



Resim 5

UML sınıf diyagramında DAO, Musteri, DAOHelper, JdbcDAO, HibernateDAO isimlerinde sınıflar tanımladık. Bu sınıfların görevlerinin ne olduğuna bir göz atalım:

- Musteri: Bu sınıf bünyesinde, bir müşteri için gerekli bilgiler tutulur (isim, soyad, adres).
- DAO (Data Access Object): Bu interface ile müşteri nesneleri tarafından kullanılacak veri tabanı işlemleri tanımlanır (insert, update, delete). DAO tasarım şablonunu diğer bir bölümde detaylı olarak inceliyecegiz.
- DAOHelper: DAOHelper dao.properties dosyasında tanımlanmış DAO sınıfı implementasyonunu edinmemizi sağlar.
- JdbcDAO: Bir müşteri nesnesini JDBC teknolojisi ile veri tabanına ekler, siler, değiştirir.
- HibernateDAO: Müşteri sınıfını Hibernate çatısını kullanarak veri tabanına ekler, siler, değiştirir.
- dao.properties: Program tarafından kullanılacak DAO implementasyonunu tanımlar.

DAO interface sınıfını tanımlayarak, bir müşteri nesnesinin oluşumu ve program içinde kullanımı ile bu nesnenin veri tabanına eklenmesi işlemini birbirinden ayırmış olduk. Programımız bir müşteri nesnesini veri tabanına eklemek istediğinde, DAOHelper sınıfı üzerinden bir DAO

implementasyonu alarak, gerekli işlemi yapacaktır.

Peki böyle bir mimarinin esnekliği nereden kaynaklanıyor? Bu noktaya açıklık getirelim. Programın burada tanıması gereken sadece DAO interface sınıfıdır. DAO interface sınıfı doğrudan kullanılamıyacağı için (çünkü bir interface), implementasyon sınıflarını oluşturmamız gerekiyor. Bunlar ilk etapta JdbcDAO ve HibernateDAO sınıflarıdır. dao.properties dosyası içinde kullandığımız DAO implementasyon sınıfını belirterek, istediğimiz esnekliğe kavuşuyoruz. Programı tekrar derlemek zorunda kalmadan, dao.properties dosyasını değistirerek HibernateDAO sınıfını kullanabiliriz ya da başka bir DAO sınıfı oluşturabiliriz (örneğin FileDAO). dao.properties dosyasının yapısını ve nasıl kullanıldığını biraz sonra görecegiz.

```
// Kod 29

public class Test {

    public static void main(String args[])
    {
        Musteri musteri = new Musteri();
        musteri.setIsim("Ahmet");
        musteri.setSoyad("Yildirim");
        musteri.setAdres("Sisli / İstanbul");

        /**
         * Müşteri nesnesi bilgibankasina
         * insert edilir.
         */
        DAOHelper.getDAO().insert(musteri);
    }
}
```

Yukarıda yer alan Test sınıfında bir müşteri nesnesinin oluşturulması ve DAOHelper üzerinden veri tabanına eklenmesi işlemini görüyoruz. DAOHelper sınıfı hangi DAO implementasyon sınıfının kullanıldığını nereden biliyor?

```
// Kod 30

public class DAOHelper {

    public static DAO getDAO(){
        try {
            /**
             * dao.properties içinde yer alan
             * dao.impl anahtarının değerini
```

```
             * okuyarak, DAO interface
             * sınıfını implemente etmiş bir nesne
             * oluşturur.
             */
            return ((DAO)Class.forName(PropertyHandler.
                    getProperty("dao.impl")).newInstance());
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

DAOHelper.getDAO() statik metodu ile dao.properties içinde yer alan DAO implementasyon sınıfına ulaşıyoruz. Bunun için PropertyHandler isminde bir sınıftan yararlanıyoruz.

```
// Kod 31

import java.util.ResourceBundle;

public class PropertyHandler {

    public  static String getProperty(String key){
        return ResourceBundle.getBundle("dao").getString(key);
    }
}
```

PropertyHandler sınıfı ResourceBundle sınıfını kullanarak, classpath içinde dao.properties isminde bir dosyayı arıyor. Bu dosyanın içeriği bir sonraki tabloda yer almaktadır.

```
// Kod 32

##DAO implementasyon sınıfı
dao.impl = com.pratikprogramci.designpatterns.muesteriyoenetimi.JdbcDAO
```

dao.properties dosyasındaki satırlar anahtar / değer (key/value) değerlerinden oluşmaktadır. dao.impl anahtarı için kullanmak istediğimiz DAO implementasyon sınıfı olan JdbcDAO sınıfının paket ismini yazıyoruz:

```
com.pratikprogramci.designpatterns.muesteriyoenetimi.JdbcDAO
```

DAOHelper.getDAO() metodu içinde kullanmak istediğimiz anahtarı belirterek (dao.impl),

Class.forName() ile yeni bir JdbcDAO nesnesi oluşturuyoruz (newInstance()).

DAOHelper sınıfı içinde aşağıdakı örnekte yer aldığı gibi alternatif bir yazılımda yapılabilirdi. Bu yöntemin bir dezavantajı var: Başka bir DAO implementasyonu kullanmak istediğimizde, DAOHelper.getDAO() metodunu değiştirip, tekrar derlememiz gerekecektir. Oysaki biz yukarıda yer alan örnekte olduğu gibi yapılması gereken değişikliği dao.properties dosyasında tanımlıyarak, programı tekrar derleme ihtiyacı olmadan, başka bir DAO implementasyon sınıfı kullanabiliriz. Yapmamız gereken sadece yeni DAO sınıfını dao.properties dosyasına eklemektir.

```
// Kod 33

public class DAOHelper2 {

    public static DAO getDAO(){
        try {
            return new JdbcDAO();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

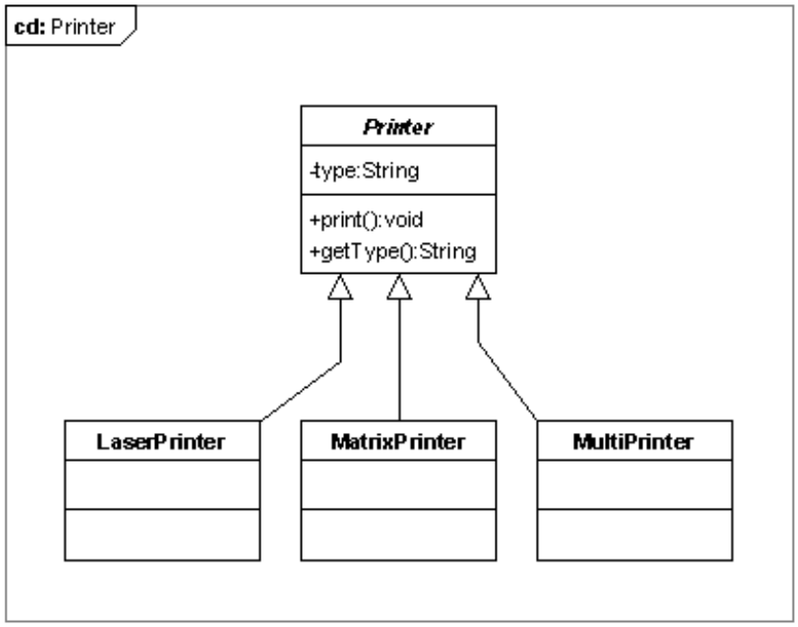
Programın ilk versiyonunda JdbcDAO sınıfını kullanma kararı veriyoruz. İstediğimiz zaman yeni bir DAO implementasyonu oluşturarak, programın diğer bölümlerini değiştirmek zorunda kalmadan kullanabiliriz.

```
// Kod 34

public class JdbcDAO implements DAO{
    public void insert(Musteri musteri) {
        Connection con = null;
        PreparedStatement pstmt = null;
        try {
            con = getConnection();
            pstmt = con.prepareStatement("insert into musteri " +
                    "(isim,soyad, adres) values(?,?,?)");
            pstmt.setString(1, musteri.getIsim());
            pstmt.setString(2, musteri.getSoyad());
            pstmt.setString(3, musteri.getAdres());
            pstmt.executeUpdate();
        }
        catch (Exception e) {
```

```
        }
    }

    public void update(Musteri musteri) {
        //TODO
    }

    public void delete(Musteri musteri) {
        //TODO
    }

    public Connection getConnection(){
        //TODO
        return null;
    }
}
```

Neden Abstract ve Interface Sınıflar Yeterli Değildir?

Önce soyut sınıfların yazılım yaparken sebep oldukları sorunlara bir göz atalım.



Resim 6

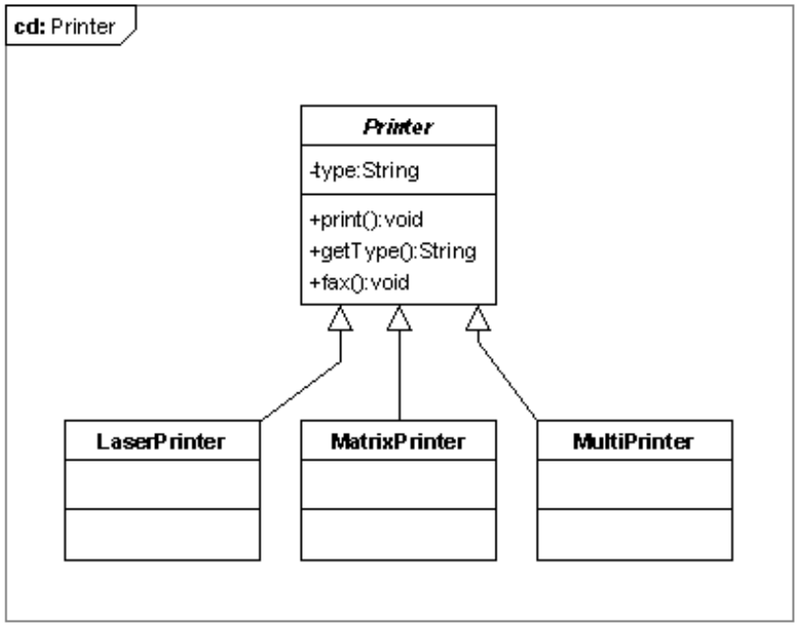
Printer isminde genel anlamda bir yazıcıyı modelleyen bir soyut sınıf oluştuyoruz. Amacımız değişik tipteki yazıcılar için bir üstsınıf oluşturmak ve ortak olan özellikleri bu sınıfın bünyesinde toplamak.

```
// Kod 35

public abstract class Printer {
    private String type;


    public Printer(){
    }

    public Printer(String _type){
        setType(type);
    }

    public void print(){
        System.out.println("Printed with "+getType());
    }

    public void fax() {
        System.out.println("");
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }
}
```

Tüm yazıcıların ortak yanı baskı işlemidir. Bunun için print() isminde bir metot tanımlıyoruz. Oluşturulacak olan tüm altsınıflar Printer sınıfının sahip olduğu tüm değişken ve metotlara sahip olacaktır ve böylece print() metodunu ortak kullanacaklardır.

Yazılım yaparken amacımız, yazdığımız kodun ilerde bakımının kolay ve kolaylıkla değiştirilebilir ve genişletilebilir halde olmasını sağlamaktır. Zamanla değişik yapıda ve modelde yazıcı sistemlerinin piyasaya sürüleceği bir gerçektir. Peki yaptığımız model bu gelişmeleri bünyesine katabilecek yapıda mıdır? Bunu deneyelim ve görelim.

Modelimizde MultiPrinter isminde, faks gönderme fonksiyonuna sahip bir yazıcı bulunmaktadır. Faks gönderebilmek için fax() isminde bir metodun oluşturulması gerekiyor.



Resim 7

Altsınıflara tek tek fax() metodunu eklemek ve implemente etmek istemediğimiz için Printer sınıfına fax() isminde bir metot ekliyoruz. İmplemente ettigimiz fax() metodunu tüm altsınıflar kullanabilir.

Bu noktadan itibaren bir sorunla karşı karşıyayız! Printer sınıfına eklenen bir metot tüm altsınıflar tarafından kullanılır. Bu durumda faks gönderme özelliğine sahip olmayan LaserPrinter ya da MatrixPrinter fax() gönderme metoduna sahip olarak, faks gönderme özelliğine kavuşuyorlar. Lakin fiziksel olarak bu yazıcılar faks gönderemezler, yani farkında olmadan bazı altsınıflar için gereksiz davranışlar (bir nesnenin davranışı sahip olduğu bir metotdur.) oluşturduk. Bu şartlar altında faks gönderme fonksiyonu olmayan bir yazıcıyı faks göndermek için kullanabiliriz ve bu bir hatadır!

Bu sorunu fax() metodunu altsınıflarda, yazıcının yapısına göre tekrar implemente ederek çözebiliriz:

```
// Kod 36

public class LaserPrinter extends Printer{
    @Override
    public void fax()
    {
        // LaserPrinter fax gönderemedigi için bu metot
        // bos kalmak zorundadır.
    }
```

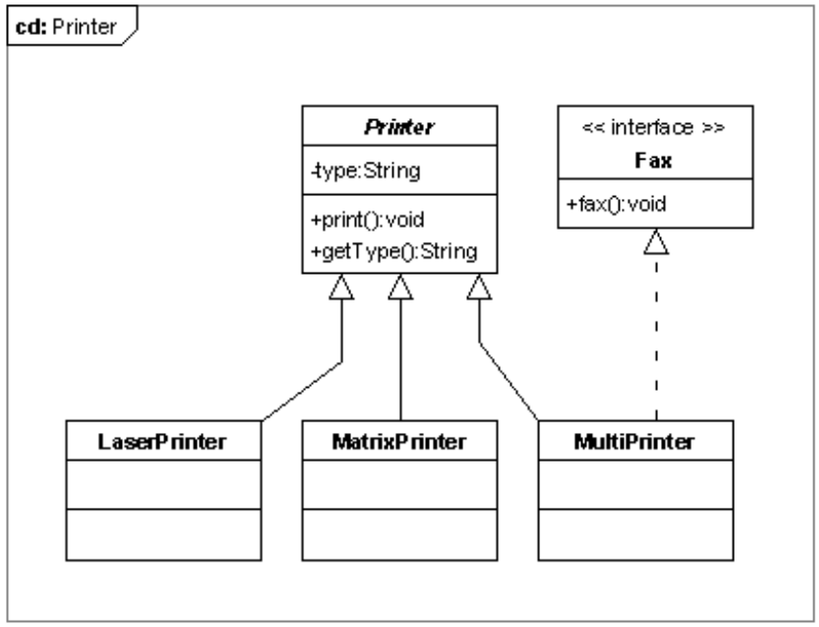
```
}
```

LaserPrinter faks gönderme özelliğine sahip olmadığı için fax() metodunun bu sınıf bünyesinde boş kalması gerekiyor.

```
// Kod 37

public class MultiPrinter extends Printer{
    @Override
    public void fax()
    {
        System.out.println("Fax sent with " + getType());
    }
}
```

MultiPrinter faks gönderebildiği için fax() metodunu implemente etmektedir. Yüzlerce değişik tipte yazıcının bulunduğu bir sistemi düşünürsek, sadece fax() göndermek için her altsınıf içinde değişiklik yapmamız gerekecek. Bu çok yorucu bir işlem olup sistemi kırılgan bir hale sokabilir. Ben şahsen programcı olarak böyle bir işle ugraşmak istemezdim!

Soyut bir sınıf kullanıldığı taktirde, program yapısının yapılacak değişikliklere pek açık olmadığını gördük. Peki bu sorunu bir interface sınıf kullanarak çözebilir miyiz? Aşağıdaki gibi bir çözüm düşünülebilir:



Resim 8

Fax() metodu sadece bazı yazıcılar tarafından desteklendiği için Fax isminde bir interface sınıfı oluşturarak, faks gönderebilecek yazıcıların bu sınıfı implemente etmesini sağlıyoruz. Böylece gerçekten faks gönderebilen yazıcılar bu özelliğe kavuşuyor.

```
// Kod 38

public interface Fax {
    void fax();
}


public class MultiPrinter extends Printer implements Fax{
    public void fax()
    {
        System.out.println("Fax sent with " + getType());
    }
}
```

Peki sizce bu iyi bir çözüm müdür? Bizi bekleyen sorunları kestirebiliyor musunuz? Faks gönderme işlemini değiştirmek zorunda kaldığımızı düşünelim, bu durumda sistemde ne gibi değişiklikler yapılması gerekiyor?

Interface sınıfı kullandığımız zaman karşılaştığımız ilk sorun, her altsınıfın interface sınıfında yer alan metotları implemente etmesi gerçeğidir. Elliye yakın altsınıfın bulunduğu bir sistem düşünün. Fax() metodunu değiştirmek zorunda kaldığımızda, bu elli sınıfın fax() metodunu değiştirmek zorunda kalacağız. Gereksiz yere aynı kodu çoğaltıp, program içinde kullanmış oluyoruz. Bu sistemin bakımını çok zorlaştırır ve mutlaka ve mutlaka sakınılması gereken bir durumdur.

Modelimizi tekrar gözden geçirdigimiz zaman, soyut sınıfların kullanılmasının sorunumuzu çözmekte yeterli olmadığı görüyoruz. Aynı şekilde interface sınıfları kullanarak sorunumuzu çözemiyoruz, çünkü Java'da bulunan interface sınıflarda kod implementasyonu yapmak Java 8 öncesi mümkün değildir. Bu sebepten dolayı her interface metodunun altsınıflarca implemente edilmesi gerekmektedir ve bu da aynı kodların tekrar tekrar başka sınıflarda kullanılmasına ve yapı itibariyle bakımı zor kalıpların oluşmasına sebebiyet vermektedir. Kısaca interface sınıfları hakkında şunu diyebiliriz: "Interface sınıfları kodun tekrar kullanılmasını sağlayamaz". Ama bu bizim amacımız olmalı: "Mümkün mertebe yazılan bir sınıf ya da metot kodu sistemin başka bir yerinde kullanılabilmelidir. Bu en önemli nesneye yönelik prograglama presiplerinden biridir.

3. Bölüm

Unified Modeling Language (UML) Giriş

Bu kitapda tasarım şablonlarının kullanımı açıklamak amacıyla diyagramlar kullandım. Bu diyagramların büyük bir çoğunluğu UML (Unified Modelling Language) diyagramları oluşturmaktadır. UML diyagramları sınıfları ve sınıflar arasındaki ilişki ve interaksiyonu modellemek için kullanılan bir diyagram dilidir.

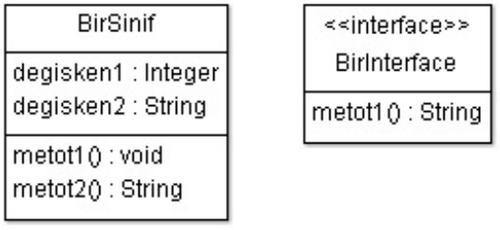
UML bünyesinde değişik diyagram tipleri bulunmaktadır. Bunlar:

- Sınıf diyagramları (class diagram)
- Aktivite diyagramları (activity diagram)
- Komponent diyagramları (component diagram)
- Paket diyagramları (package diagram)
- Nesne diyagramları (object diagram)
- Dizi diyagramları (sequence diagram)
- İletişim diyagramları (communication diagram)
- Profil diyagramları (profile diagram)
- Kullanım senaryo diyagramları (use case diagram)

Bu kitabın ana konusu ne yazık ki UML değil. Bu yüzden UML konusunu detaylı olarak bu bölümde açıkıamam mümkün değil. Daha ziyade bu bölümde UML diyagramlarını okuyabilmek için gerekli olan bilgiyi sunmak istiyorum. Tasarım şablonlarının kullanımında önemli olan iki UML diyagram tipi bulunmaktadır. Bunlar sınıf ve dizi diyagramlarıdır. Şimdi bu diyagram tiplerini yakından tanıyalım.

Sınıf Diyagramları

Sınıf diyagramları aracılığı ile sınıfları ve sınıflar arası ilişkileri modelleyebiliriz. UML bünyesinde sınıfları ve interface sınıfları tanımlamak için kullanilabilecek iki sınıf tipi bulunmaktadır. Bunları resim 1 de görmekteyiz.

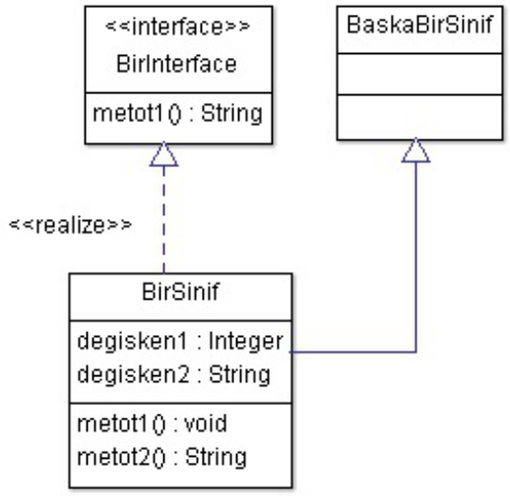


Resim 1

Resim 1 de yer alan sınıfları Java dilinde şu şekilde kodlayabiliriz:

```
// Kod 1

public class BirSinif{

    private Integer degisken1;
    private String degisken2;

    public void metot1(){
    }

    public String metot2(){
    }
}

public interface BirInterface{

    public String metot1();
}
```

Kalıtım resim 2 de yer aldığı şekilde modellenebilir. Bu örnekte BirSinif sınıfı BirInterface sınıfını implemente etmekte ve BaskaBirSinif sınıfını genişletmektedir.



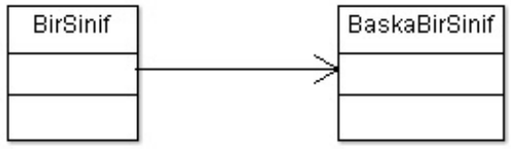
Resim 2

Bu ilişkiyi kod 2 de görmekteyiz.

```
// Kod 2
```

```
public class BirSinif extends BaskaBirSinif implements BirInterface{
}
```

Resim 3 de yer alan örnekte iki sınıf arasındaki ilişki modellenmiştir. Bu örnekte BirSinif sınıfı BaskaBirSinif sınıfını sınıf değişkeni, metot parametresi ya da metot bünyesinde bir değişken olarak kullanmaktadır. Diyagramda ilişkinin yönü belirtilmiştir. Bu durumda BirSinif BaskaBirSinif sınıfını kullanmaktadır ve bunun tersi geçerli değildir. Eğer düz çizgi ok ihtiva etmemiş olsaydı, her iki sınıfın karşılıklı birbirlerini kullandıkları söylenebilirdi.



Resim 3

Sınıflar arasındaki ilişkiyi daha net tanımlamak için resim 4 de yer alan yapılar kullanılabilir.

Resim 4

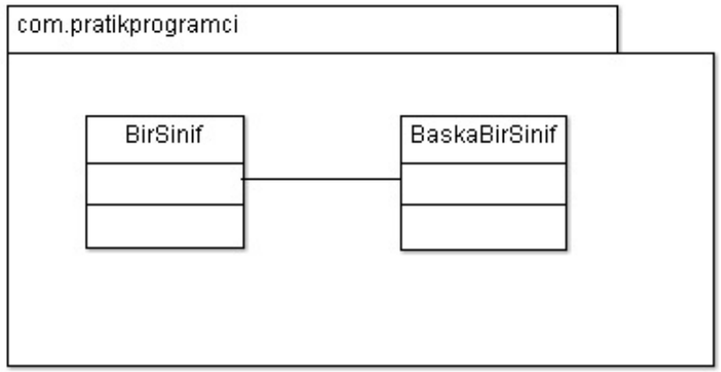
Resim 4 de yer alan diyagramın Java karşılığı kod 3 de yer almaktadır.

```
// Kod 3

public class BirSinif{

    public List<BaskaBirSinif> list;
}
```

BirSinif sınıfı bir BaskaBirSinif nesnelerinden oluşan bir listeyi sınıf değişkeni olarak kullanmaktadır. Hangi nesnenin kaç adet kullanıldığını UML diyagramında rakamlarla tanımlamak mümkündür. Resim 4 de yer alan örnekte BirSinif sıfır ya da birçok BaskaBirSinif nesnesini kullanırken, BaskaBirSinif hiçbir BirSinif nesnesini kullanmamaktadır.

Resim 5 de alan diyagram BirSinif ve BaskaBirSinif sınıflarının aynı paket içinde olduğunu

göstermektedir.

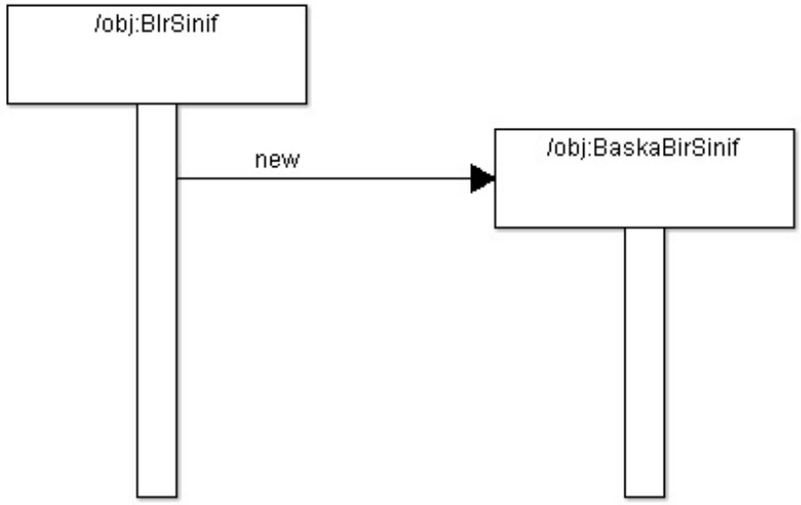


Resim 5

Dizi (Sequence) Diyagramları

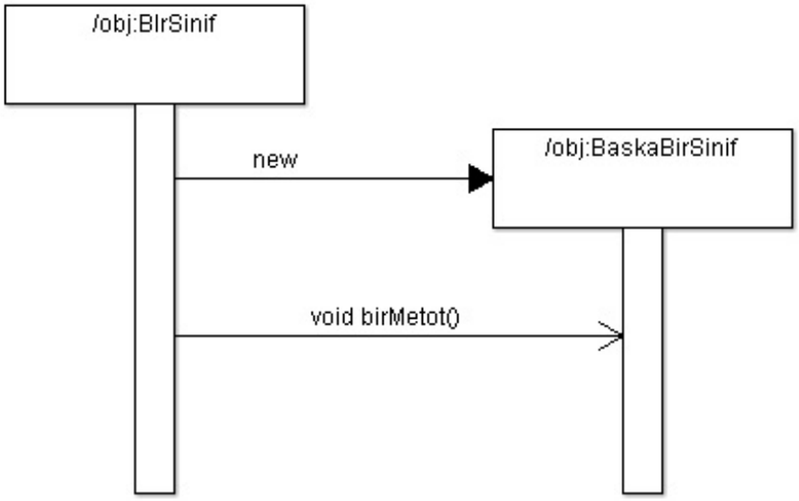
Dizi diyagramları sınıflar arasındaki interaksiyonu modellemek için kullanılan diyagram tipidir. İnteraksiyon metotlar üzerinden gerçekleşir. Dizi diyagramlarında yer alan interaksiyonlar sınıflar arası değil bu sınıflardan oluşturulan nesneler arasında gerçekleşir. Bu yüzden dizi diyagramlarinda nesneler yer alır.

Resim 6 da yer alan örnekte BirSinif sınıfından oluşturulmuş olan nesne BaskaBirSinif sınıfının konstrüktörü kullanarak, yeni bir BaskaBirSinif nesnesi oluşturmaktadır.



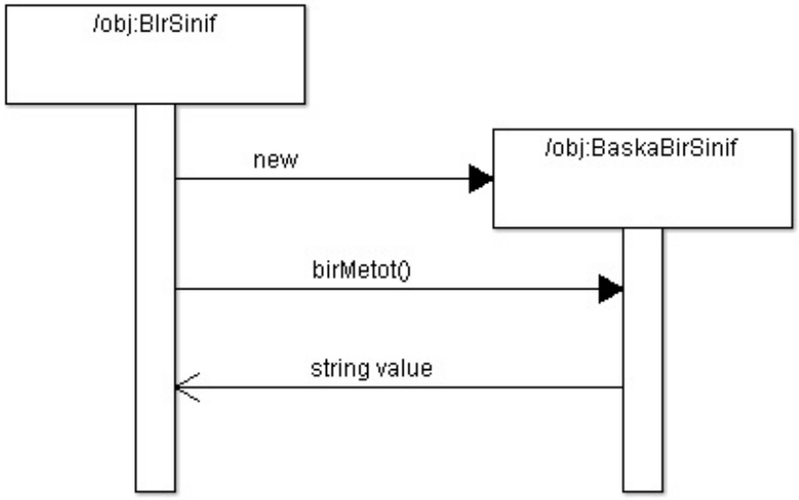
Resim 6

Resim 7 de obj isimli BirSinif nesnesi yine obj ismini taşıyan BaskaBirSinif nesnesinin birMetot() isimli metodunu koşturmaktadır. Bu metot void olduğundan, geriye bir dönüş olmamaktadır.



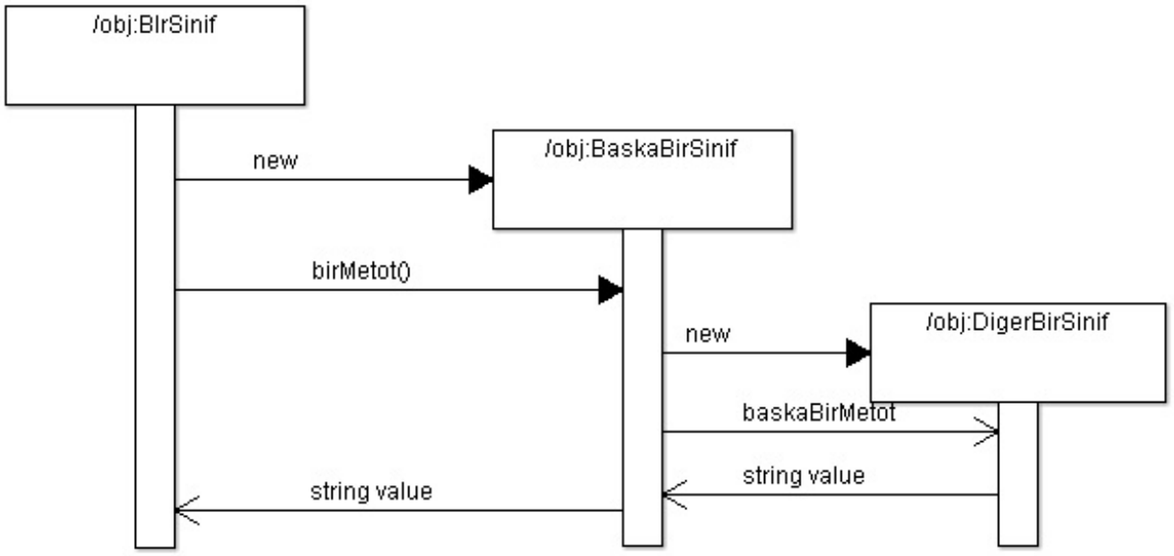
Resim 7

UML dizi diyagramları ile metot geri dönüşlerini modellemek mümkündür. Bunun bir örneğini resim 8 de görmekteyiz. Burada BirSinif nesnesi BaskaBirSinif nesnesinin birMetot() isimli metodunu koşturmakdır ve bir String değeri edinmektedir.



Resim 8

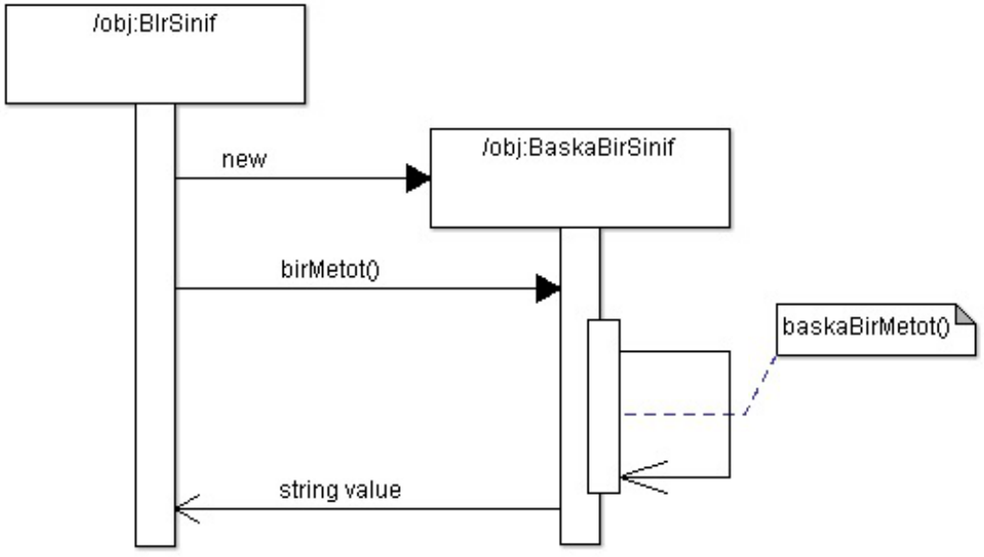
Resim 8 de BaskaBirSinif bünyesindeki birMetot() metodunun koşturulması esnasında hangi işlemlerin yapıldığı görülmemektedir. Örneğin bu metot bünyesinde başka bir sınıfdan bir nesne oluşturularak, bu nesnenin bir metodu koşturulmuş olabilir. Böyle bir yapının modelleniş şekli resim 9 da yer almaktadır.



Resim 9

Resim 9 da görüldüğü gibi BirSinif nesnesinin birMetot() metodunu koştururarak edindiği değer aslında DigerBirSinif sınıfının baskaBirMetot() metodundan gelmektedir.

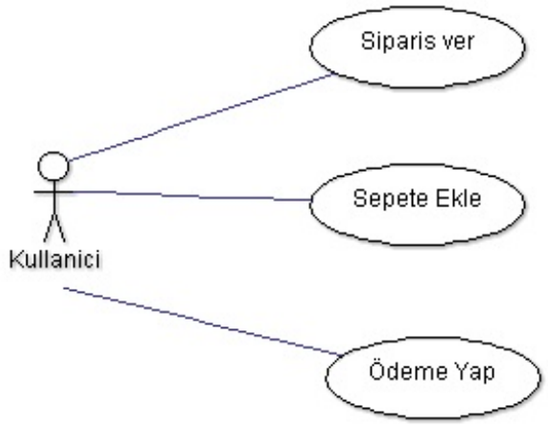
Bir sınıf metodunun kendi sınıfında yer alan başka bir metodu nasıl koşturduğu resim 10 da yer almaktadır.



Resim 10

Kullanım Senaryo (Usecase) Diyagramları

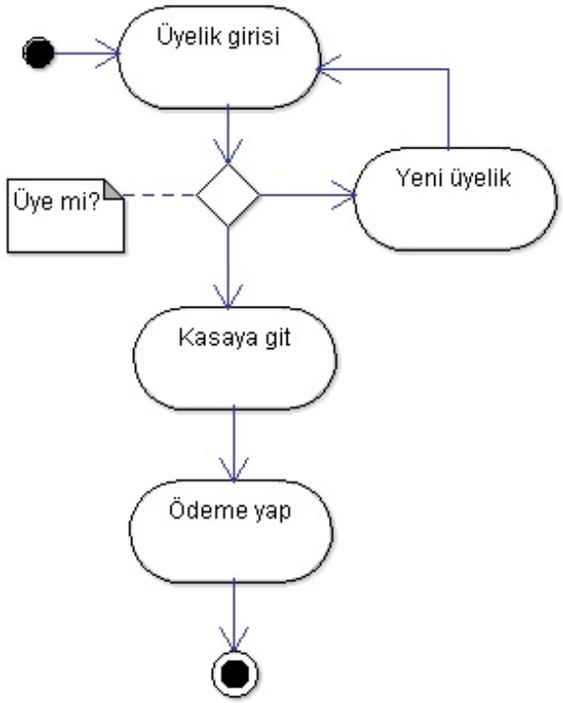
Kullanım senaryo diyagramları uygulama kullanıcılarının kullanım senaryolarını modellemek için kullanılan bir diyagram türüdür. Resim 11 de yer alan örnekte kullanıcı sipariş verme, sepete ürün ekleme ve ödeme yapma gibi işlemleri gerçekleştirebilmektedir. Kullanım senaryo diyagramları aracılığı ile uygulamanın hangi özelliklere sahip olacağı tanımlanmış olur.



Resim 11

Aktivite (Activity) Diyagramları

Aktivite diyagramları bir metot bünyesindeki iş mantığını ya da birden fazla sınıf ve metot aracılığı ile çalışan bir is akışını modellemek etmek için kullanılmaktadir. Resim 12 de yer alan örnekte bir müşterinin sipariş sonrasi ödeme işlemi için atılması gereken adımlar yer almaktadır.



Resim 12

Aktivitäten başlangıç noktası üyelik sorgulamasıdır. Eğer müşteri sisteme üyü ise doğrudan kasaya yönlendirilir. Eğer müşterinin üyeliği yoksa, müşteri üye olmak amacıyla yeni üyelik sayfasına yönlendirilir. Yeni üyelik işlemi tamamlandıktan sonra müşteri tekrar üyelik girişine aktarılır. Buradan yapılan sorgulama neticesinde kasaya, oradan da ödeme sayfasına yönlendirilir. Ödeme işlemi tamamlandıktan sonra, aktivite sonlandırılır.

4. Bölüm

Tasarım Prensipleri

Bu bölümde iyi bir tasarım oluşturabilmek için takip edilmesi gereken prensipleri inceleyeceğiz. Bu prensipler uygulandığı taktirde yapı itibariyle esnek ve geliştirilmesi kolay programlar oluşturabiliriz. Gerekli durumlarda bu prensiplerin takip edilmesi faydalı olacaktır, lakin sadece prensibi uygulamış olmak için yapılan değişiklikler karmaşık bir yapıyı doğuracaktır. Amaç her zaman en basit yöntemler kullanılarak, sade ve esnek yapılar oluşturmak olmalıdır.

İyi bir tasarım oluşturabilmek için implementasyon esnasında uygulanması gereken prensipler şöyledir:

- Loose Coupling (LC)
- Open Closed Principle (OCP)
- Single Responsibility Principle (SRP)
- Liskov Substitution Principle (LSP)
- Dependency Inversion Principle (DIP)
- Interface Segregation Principle (ISP)
- Reuse-Release Equivalence Principle (REP)
- Common Reuse Principle (CRP)
- Common Closure Principle (CCP)
- Acyclic Dependency Principle (ADP)
- Stable Dependencies Principle (SDP)
- Stable Abstractions Principle (SAP)
- Tasarım Şablonları (Design Patterns)

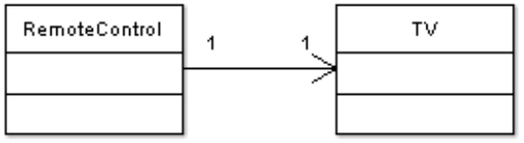
Loose Coupling (LC) - Esnek Bağ

Bir program bünyesinde tanımlanan görevlerin yerine getirilebilmesi için birden fazla nesne görev alır. Bu nesneler birbirlerinin sundukları hizmetlerden faydalanarak, kendi görevlerini yerine getirirler. Bu durumda nesneler arası bağımlılıklar oluşur. Bir nesne kullandığı diğer bir nesne hakkında ne kadar fazla detay bilgiye sahip ise, o nesneye olan bağımlılığı o oranda artar. Oluşan her bağımlılık bir sınıf için dolaylı olarak yapısal değiştirilme rizikosunu artırır, çünkü bağımlı olduğu sınıf üzerinde yapılan her değişiklik kendi yapısında değişikliğe neden olacaktır. Bu durum programın genel olarak kırılgan bir hale gelmesini kolaylaştıracaktır.

Buradan “eğer bağımlılık varsa, sorun var, bu yüzden bağımlılıkların ortadan kaldırılması gerekmektedir” sonucunu çıkartabiliriz. Nesneye yönelik tarzda tasarlanmış bir program içinde bağımlılıkları ortadan kaldırmak imkansızdır, çünkü nesnelerin olduğu yerde interaksiyon ve bağımlılık olmak zorundadır. Bağımlılıkları ortadan kaldıramıyorsak, o zaman onları kontrol altına almamız işimizi kolaylaştıracaktır. Eğer bana soracak olursanız, yazılım disiplininin özü de burada

saklıdır: Yazılımcı olarak kullandığımız tüm metotların temelinde bağımlılıkların kontrolü ve yönetilmesi yatmaktadır. İyi bir tasarım oluşturmak için sarf ettiğimiz efor bağımlılıklara hükmetmek isteyişimizden kaynaklanmaktadır, yani iyi bir tasarım kontrol edilebilir bağımlılıkları beraberinde getirdiği için iyidir, kendimizi programcı olarak iyi hissetmemizi sağladığı için değil! Biliyoruz ki kötü bir tasarım nesneler arası yüksek derecede bağımlılığa sebep vereceği için programcı olarak hayatımızı zorlaştıracaktır. Bu yüzden sahip olduğumuz tüm teknik yeteneklerimizle bağımlılığa karşı bir savaş veririz. Onu yenmemiz mümkün olmasa da, bize zarar vermeyecek şekilde kontrol altına almamız mümkündür.

Anladığımız kadarıyla bağımlılıkları mantıklı bir çerçevede ortadan kaldırmamız imkansız! Peki bağımlılıkları nasıl kontrol altına alabiliriz? Esnek bağımlılıklar oluşturarak! Esnek bağımlılık oluşturmak demek, nesneler arası bağların oluşmasına izin vermek, ama sınıflar üzerinde yapılan yapısal değişikliklerin bağımlı sınıflar üzerinde yapısal değişikliğe sebep vermesini engellemek demektir. Bunu bir örnek vererek açıklayalım.



Resim 1

RemoteControl (tv uzaktan kumanda aleti) ve TV (televizyon) sınıflarının arasında tek yönlü (uni directional) bir bağ oluşmuştur, çünkü bir RemoteControl nesnesi görevini yerine getirebilmek için bir TV nesnesine ihtiyaç duymaktadır. Nesneler arası bağımlılıkların yönü vardır. Resim 1 de bu bağımlılığın yönünün RemoteControl sınıfından TV sınıfına doğru olduğunu görmekteyiz. Bu tek yönlü bir bağdır ve RemoteControl sınıfı bünyesinde TV tipinde bir sınıf değişkeni barındırarak, bu bağı oluşturur. Sınıflar arası bağlar karşılıklı da (bidirectional) olabilir. İki taraflı bağlarda sınıflar bir sınıf değişkeni aracılığıyla karşılıklı olarak birbirlerine işaret ederler. RemoteControl nesnesi bir TV nesnesi olmadığı sürece işe yaramaz. Bu yüzden bu iki sınıf arasında doğrudan bağlantı oluşturulmuştur. Böyle bir bağımlılık aşağıdaki sorunların oluşmasına sebep vermektedir:

- RemoteControl nesnesi bir TV nesnesi olmadığı sürece kendi görevini yerine getiremez, bu yüzden tek başına bir işe yaramaz. Mutlaka ve mutlaka var olabilmesi için bir TV nesnesine ihtiyaç duymaktadır. Bu durumda RemoteControl sınıfını başka bir alanda kullanmak istediğimizde TV sınıfını da yanına koymak zorundayız. Böyle bir bağımlılık RemoteControl sınıfının tek başına başka bir alanda tekrar kullanılmasını engellemektedir. Buradan söyle bir sonucu çıkartıyoruz: Program parçalarının (sınıflar) tekrar kullanılabilir olabilmeleri için diğer sınıflara olan bağımlılıklarının düşük seviyede olması gerekmektedir. Esnek bağımlılık olmadan bir kere yazılan kodun tekrar kullanımı çok güçtür.

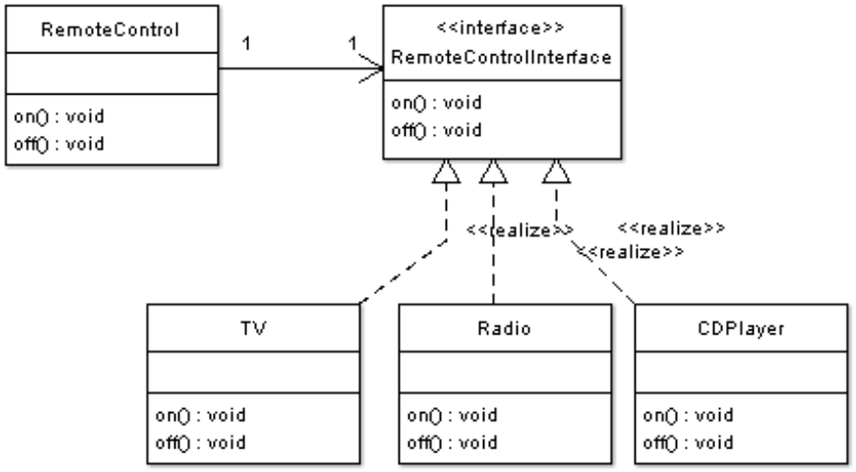
- TV sınıfı bünyesinde meydana gelen yapısal değişiklikler RemoteControl sınıfını doğrudan etkileyecektir. Böyle bir bağımlılık RemoteControl sınıfının yapısal değişikliğe uğrama rizikosunu artırır.
- RemoteControl nesnesi sadece bir TV nesnesini (yani bir televizyonu) kontrol edebilir. Oysaki bir evde uzaktan kumanda edilebilecek başka aletler de olabilir. Böyle bir bağımlılık RemoteControl nesnesini sadece bir TV nesnesiyle beraber çalışmaya mahkum eder. Uzaktan kumanda edilebilen aletler için başka RemoteControl sınıflarının oluşturulması gerekmektedir, örneğin bir CD çalıcıyı kontrol etmek için CDPlayerRemoteControl isminde bir sınıfı oluşturmak zorundayız.

RemoteControl sınıfı kod 1 de yer almaktadır.

```
// Kod 1

public class RemoteControl{
    private TV tv = new TV();

    public void tvOn(){
        tv.on();
    }

    public void tvOff(){
        tv.off();
    }
}
```

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

```
// Kod 2

public class TV{
    public void on(){
        System.out.println("TV açıldı.");
    }

    public void off(){
        System.out.println("TV kapandi");
    }
}
```



Resim 2

Nesneler arası kuvvetli bağların oluşmasının programın bakımı, geliştirilmesi ve kodun tekrar kullanımını negatif yönde etkilediğini gördük. Bu sorunu ortadan kaldırabilmek için sınıfların ilişkileri üzerinde yapısal değişikliğe gitmemiz gerekiyor. Esnek bağ oluşturabilmek için soyut (abstract) ya da interface sınıflardan faydalanabiliriz. Resim 2 de görüldüğü gibi RemoteControlInterface ismini taşıyan bir interface sınıf ile RemoteControl ve bu sınıfın kontrol etmek istediği aletler arasında esnek bir bağ oluşturuyoruz. Böyle bir yapının esnekliği nereden gelmektedir, bunu yakından inceleyelim.

```
// Kod 3

public interface RemoteControlInterface{
    void on();
    void off();
}
```

Alt sınıflar kullanılarak interface sınıfında tanımlanmış olan metotlar implemente edilir. Bu sayede interface sınıfında tanımlanmış metotlar için değişik sınıflarda değişik tarzda implementasyonlar yapmak mümkündür. Herhangi bir sınıf implements direktifini kullanarak bir interface sınıfını implemente edebilir. Interface sınıfında tanımlanmış olan tüm metotların alt sınıflarca implemente edilmesi gerekmektedir.

Resim 2 de görüldüğü gibi RemoteControl sınıfı doğrudan TV sınıfı kullanmak yerine, RemoteControlInterface ismini taşıyan interface sınıfı ile beraber çalışmaktadır. Böyle bir yapılanma ile RemoteControl ve bu sınıfın kullanmak istediği somut sınıflar (örneğin TV) arasına bir set çekmiş olduk. RemoteControl sınıfı bu setin arkasında yer alan sınıfları, bunlar RemoteControlInterface sınıfını implemente eden sınıflardır, tanımamaktadır ve tanımak zorunda değildir. RemoteControl sınıfının tanıması gereken tek sınıf RemoteControlInterface sınıfı ve bu

interface sınıfının dış dünyaya sunduğu metotlardır. RemoteControl sınıfı böylece dolaylı olarak RemoteControlInterface sınıfını implemente eden her sınıfı kullanabilir hale gelmektedir. Böylece RemoteControl sınıfının somut sınıflara olan bağımlılığı bir interface sınıf kullanılarak ortadan kaldırılmıştır.

```
// Kod 4

public class RemoteControl{
    private RemoteControlInterface remote;

    public RemoteControl(RemoteControlInterface _remote){
        this.remote = _remote;
    }

    public void on(){
        remote.on();
    }

    public void off(){
        remote.off();
    }

}
```

RemoteControl sınıfını RemoteControlInterface sınıfını kullanacak şekilde değiştiriyoruz. Bu amaçla RemoteControl sınıfında RemoteControlInterface tipinde bir sınıf değişkeni (remote) tanımlıyoruz. Sınıf konstrüktörü RemoteControlInterface tipinde bir parametre kabul etmektedir. Böylece RemoteControl sınıfından bir nesne oluştururken istediğimiz tipte bir RemoteControlInterface implementasyon sınıfı kullanabiliriz.

RemoteControl sınıfı on() ve off() isminde iki metot tanımlamaktadır. Bu metotlar RemoteControlInterface sınıfında yer alan on() ve off() metotları ile karıştırılmamalıdır. RemoteControl sınıfı sahip olduğu metotlara ac() ve kapat() isimlerini de verebilirdi. Bu metotlar içinde delegasyon yöntemiyle sınıf değişkeni olan remote nesnesi kullanılmaktadır. Bu sayede kullanılan RemoteControlInterface implementasyon sınıfının (örneğin TV) on() ve off() metotları devreye girecektir.

```
// Kod 5

public class TV implements RemoteControlInterface{

    public void on(){
```

```
        System.out.println("TV açıldı.");
    }

    public void off(){
        System.out.println("TV kapandi");
    }
}
```

TV sınıfı RemoteControlInterface sınıfını implemente etmektedir. Bu sebepten dolayı RemoteControlInterface sınıfında tanımlanmış olan on() ve off() metotlarına sahiptir. TV sınıfı RemoteControlInterface sınıfını implemente etmediği sürece RemoteControl sınıfı tarafından kullanılamaz.

```
// Kod 6

public class Test{

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

Test.main() bünyesinde ilk önce kullanmak istediğimiz alet nesnesini (TV) ve akabinde RemoteControl nesnesini oluşturuyoruz. RemoteControl konstrüktör parametresi olarak bir satır önce oluşturduğumuz TV nesnesini almaktadır. RemoteControl bünyesinde yer alan on() ve off() metotları ile televizyonu açıp, kapatabiliriz. Ekran çıktısı şu şekilde olacaktır:

```
TV açıldı.
TV kapandi
```

RemoteControl sınıfının konstrüktörü RemoteControlInterface sınıfını implemente eden her sınıfı kabul ettiği için istediğimiz herhangi bir aleti RemoteControl sınıfı ile kontrol edebilir hale geliyoruz.

Oluşturduğumuz yeni tasarımın bize sağladığı avantajlar şöyledir:

- Bir interface sınıf (RemoteControlInterface) kullanarak RemoteControl ve kontrol etmek istediği aletler (TV, CDPlayer) arasında bir bariyer oluşturduk. RemoteControl sınıfı kontrol etmek istediği aleti tanımak zorunda olmadığı için bu sınıf ve diğerleri arasındaki sıkı bağı

çözmüş ve interface sınıf kullanarak daha esnek bir hale getirmiş oluyoruz.
- Bu tarz bir tasarım ile programı gelecekte oluşacak değişiklikleri taşıyabilecek hale getirdik. RemoteControl sınıfı RemoteControlInterface sınıfını implemente eden her sınıfı kontrol edebilir. RemoteControlInterface sınıfını implemente ederek sisteme uzaktan kumanda edilebilen yeni aletler ekleyebiliriz.
- RemoteControl sınıfı, RemoteControlInterface sınıfının implemente edildiği başka bir ekosistemde tekrar kullanılabilir hale geldi. Esnek bağımlılık oluşturmak kodun tekrar kullanımını kolaylaştırmaktadır.

İncelediğimiz örneklerde nesneler arası bağın ortadan kaldırılamayacağını ama esnek bağ oluşturma prensibini uygulayarak kontrol edilebilir bir hale getirilebileceklerini gördük. Esnek bağlar oluşturabilmek için interface ya da soyut sınıflardan yararlanabiliriz.

Usta yazılımcılar tasarım prensiplerine ve tasarım şablonlarına (design pattern) hakim olup, onları doğru yerde kullanmasını bilirler. Eğer şimdiye kadar tasarım prensipleri hakkında bir çalışmanız olmadıysa, sizin için yazılım disiplininde bir üst boyutun kapısını aralamış olduk. Tasarım prensiplerini uygulayarak ve tasarım şablonlarını kullanarak konseptüel daha yüksek seviyede çalışabilirsiniz. Bu bölümde yer alan tasarım prensipleri yazılım sürecine olan bakış açınızı tamamen değiştirecek niteliktedir.

Open Closed Principle (OCP) - Açık Kapalı Prensibi

Yazılım disiplininde değişmeyen bir şey varsa o da değişikliğin kendisidir. Birçok program müşteri gereksinimleri doğrultusunda ilk sürümden sonra değişikliğe uğrar. Bu doğal bir süreçtir ve müşteri programı kullandıkça ya yeni gereksinimlerini ya da mevcut fonksiyonlar üzerinde adaptasyonları gerekçe göstererek programın değiştirilmesini talep edecektir.

Tanınmış yazılım ustalarından Ivar Jacobson bu konuda şöyle bir açıklamada bulunmuştur:

"All systems change during their life cycles. This must be born in mind when developing systems are excepted to last longer than the first version."

Şu şekilde tercüme edilebilir:

"Her program görev süresince değişikliğe uğrar. Bu ilk sürümden ötesi düşünülen programların yazılımında göz önünde bulundurulmalıdır."

Değişiklik kaçınılmaz olduğuna göre bir programı gelecekte yapılması gereken tüm değişiklikleri göz önünde bulundurarak, şimdiden buna hazır bir şekilde geliştirmek mümkün müdür? Öncelikle

şunu belirtelim ki gelecekte meydana gelecek değişikliklerin hepsini kestirmemiz mümkün değildir. Ayrıca çevik süreçlerde ilerde belki kullanılabileceğini düşündüğümüz fonksiyonların implementasyonu kesinlikle yasaktır. Bu durum bizi programcı olarak gelecekteki değişiklikleri göz önünde bulundurarak bir nevi hazırlık yapmamızı engeller. Çevik süreç sadece müşteri tarafından dile getirilmiş, kullanıcı hikayesi haline dönüştürülmüş ve müşteri tarafından öncelik sırası belirlenmiş gereksinimleri göz önünde bulundurur ve implemente eder. Kısacası çevik süreç geleceği düşünmez ve şimdi kendisinden beklenenleri yerine getirir. Böylece müşteri tarafından kabul görmeyecek bir sistemin oluşması engellenmiş olur.

Çevik süreç büyük çapta bir tasarım hazırlığı ile start almaz. Daha ziyade mümkün olan en basit şekilde yazılıma başlanır. Her iterasyon başlangıcında implemente edilmesi gereken kullanıcı hikayeleri seçildikten sonra mevcut yapının yeni gereksinimlere cevap verip, veremeyeceği incelenir. Büyük bir ihtimalle mevcut yapı yeni kullanıcı hikayelerinin implementasyonu için yeterli olmayacaktır. Bu durumda programcı ekip refactoring yöntemleriyle programın yapısını yeni gereksinimleri kabul edecek şekilde modifike eder. Bu esnada tasarım yapısal değişikliğe uğrar. Bu gerekli bir işlemdir ve yapılmak zorundadır, aksi taktirde bir sonraki iterasyon beraberinde getirdiği değişikliklerle programcı ekibini yazılım esnasında zorlayacaktır.

Çevik süreç gelecekte olabilecek değişikleri göz önünde bulundurmadığına göre, programı bu değişikliklerin olumsuz yan etkilerine karşı nasıl koruyabiliriz? Bunun yolu her zaman olduğu gibi yazılım esnasında uygun tasarım prensiplerini uygulamaktan geçmektedir. Eğer çevik süreç bize gelecekte olabilecek değişikliklere karşı destek sağlamıyorsa, bizimde tedbir alarak çevik süreci, çevik prensiplere ters düşmeden takviye etmemiz gerekiyor. Çevik süreci, çevik tasarım prensiplerini kullanarak istediğimiz bir yapıda, bakımı ve geliştirilmesi kolay program yazılımına destek verecek şekilde takviye edebiliriz.

Tekrar bölüm başında sorduğum soruya dönelim ve sorunun cevabını bulmaya çalışalım. Şu şekilde bir soru sormuştum: "Değişiklik kaçınılmaz olduğuna göre, bir programı gelecekte yapılması gereken tüm değişiklikleri göz önünde bulundurarak, şimdiden buna hazır bir şekilde geliştirmek mümkün müdür?" Bu mümkün değil demiştik. Lakin esnek bir tasarım oluşturarak önü açık ve gelecek korkusu olmayan bir program yapısı oluşturabiliriz. Bunu gerçekleştirmek için kullanabileceğimiz prensiplerin başında Open Closed – Açık Kapalı prensibi (OCP) gelmektedir. Bertrand Meyer tarafından geliştirilen bu prensip kısaca şöyle açıklanabilir:

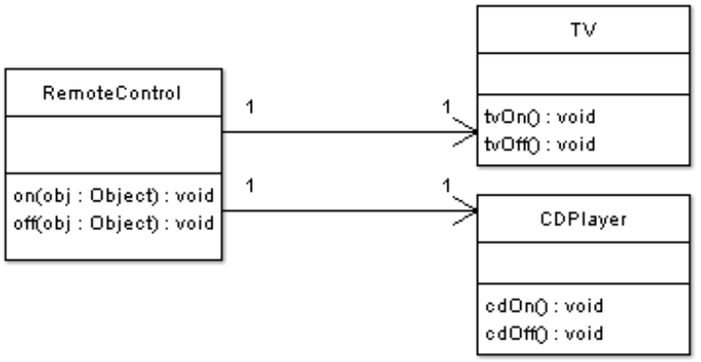
Programlar geliştirilmeye açık ama değiştirilmeye kapalı olmalıdır.

Programı geliştirmek programa yeni bir davranış biçimi eklemek anlamına gelmektedir. OCP ye göre programlar geliştirmeye açık olmalıdır, yani programı oluşturan modüller yeni davranış biçimlerini sergileyecek şekilde genişletilebilmelidirler. Bir modüle yeni bir davranış biçimi

kazandırılarak düşünülen değişiklik sağlanır. Bu yeni kod yazılarak gerçekleştirilir (bu yüzden bu işleme değiştirme değil, genişletme denir), mevcut kodu değiştirerek değil! Eğer kendinizi bir müşteri gereksinimini mevcut kod üzerinde değişiklik yaparken bulursanız, biliniz ki OCP prensibine ters düşüyorsunuz. Kod üzerinde yapılan değişiklik, bir sonraki gereksinimlerinde aynı şekilde implemente edilmesini zorunlu kılacaktır. Bu durum kodun zaman içinde içinden çıkılmaz ve çok karmaşık bir yapıya dönüşmesini çabuklaştırır.

OCP prensibinin nasıl uygulanabileceğini bir önceki bölümde yer alan RemoteControl – TV örneği üzerinde inceleyelim.

Resim 1 de görüldüğü gibi RemoteControl sınıfı TV sınıfını kullanarak işlevini yerine getirmektedir. Eğer RemoteControl sınıfını TV haricinde başka bir aleti kontrol etmek için kullanmak istersek, örneğin CDPlayer Resim 3 deki gibi değişiklik yapmamız gerekebilir. Bu noktada esnek bağ prensibini unutarak, resim 3 de yer alan çözümün bizim için yeterli olduğunu düşünelim.



Resim 3

```
// Kod 7

public class RemoteControl{

    public void on(Object obj)
    {
        if(obj instanceof TV){
            ((TV)obj).tvOn();
        }
        else if(obj instanceof CDPlayer){
            ((CDPlayer)obj).cdOn();
        }
    }
```

```
    public void off(Object obj)
    {
        if(obj instanceof TV)
        {
            ((TV)obj).tvOff();
        }
        else if(obj instanceof CDPlayer)
        {
            ((CDPlayer)obj).cdOff();
        }

    }
}
```

Bu şekilde oluşturulan bir tasarım OCP prensibine ters düşmektedir, çünkü her yeni eklenen alet için on() ve off() metotlarında değişiklik yapmamız gerekmektedir. OCP böyle bir modifikasyonu kabul etmez. OCP ye göre mevcut çalışır kod kesinlikle değiştirilmemelidir. Onlarca aletin bulunduğu bir sistemde on() ve off() metotlarının ne kadar kontrol edilemez ve bakımı zor bir yapıya bürüneceği çok net olarak bu örnekte görülmektedir.

Bunun yanı sıra RemoteControl sınıfı TV ve CDPlayer gibi sınıflara bağımlı kalacak ve başka bir alanda kullanılması mümkün olmayacaktır. TV ve CDPlayer sınıflar üzerinde yapılan tüm değişiklikler RemoteControl sınıfını doğrudan etkileyecek ve yapısal değişikliğe sebep olacaktır.

Resim 2 de yer alan çözüm OCP ye uygun yapıdadır, çünkü kod üzerinde değişiklik yapmadan programa yeni davranışlar eklemek mümkündür. OCP prensibi, esnek bağ prensibi kullanılarak uygulanabılır. OCP ye uygun RemoteControl sınıfının yapısı şu şekilde olmalıdır:

```
// Kod 8

public class RemoteControl
{
    private RemoteControlInterface remote;

    public RemoteControl(RemoteControlInterface _remote)
    {
        this.remote = _remote;
    }

    public void on()
    {
        remote.on();
    }
```

```
    public void off()
    {
        remote.off();
    }

}
```

on() ve off() metotları sadece RemoteControlInterface tipinde olan bir sınıf değişkeni üzerinde işlem yapmaktadır. Bu sayede if/else yapısı kullanmadan RemoteControlInterface sınıfını implemente etmiş herhangi bir alet üzerinde gerekli işlem yapılabilmektedir.

Bu örnekte on() ve off() metotları değişikliğe kapalı ve tüm program geliştirmeye açıktır, çünkü RemoteControlInterface interface sınıfını implemente ederek sisteme yeni aletleri eklemek mümkündür. Sisteme eklediğimiz her alet için on() ve off() metotları üzerinde değişiklik yapmak zorunluluğu ortadan kalkmaktadır. Uygulanan OCP ve esnek bağ prensibi ile RemoteControl başka bir alanda her tip aleti kontrol edebilecek şekilde kullanılır hale gelmiştir.

Stratejik Kapama (Strategic Closure)

Ne yazık ki bir yazılım sistemini OCP prensibini uygulayarak %100 değişikliklere karşı korumamız imkansızdır. OCP ye uygun olan bir metot müşterinin yeni istekleri doğrultusunda OCP ye uygun olmayan bir hale gelebilir. Programcı metodu ilk implemente ettiği zaman gelecekte olabilecek değişiklikler hakkında fikir sahibi olmayabilir. Metot OCP uyumlu implemente edilmiş olsa bile, bu metodun her zaman OCP uyumlu kalabileceği anlamına gelmez.

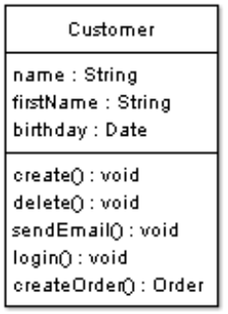
Eğer kapama tam sağlanamıyorsa, kapamanın stratejik olarak implemente edilmesi gerekir. Programcı implementasyon öncesi meydana gelebilecek değişiklikleri kestirerek, implemente ettiği metotların kapalılık oranını yükseltmelidir. Bu tecrübe gerektiren stratejik bir karardır.

Programcı her zaman ne gibi değişikliklerin olabileceğini kestiremeyebilir. Bu durumda konu hakkında araştırma yaparak, oluşabilecek değişiklikleri tespit edebilir. Eğer olabilecek değişikliklerin tespiti mümkün değilse, beklenen değişiklikler meydana gelene kadar beklenir ve implementasyon yeni değişiklikleri de yansıtacak şekilde OCP uyumlu hale getirilir.

Single Responsibility Principle (SRP) – Tek Sorumluk Prensibi

Resim 4 deki gibi bir sınıfa daha önce bir yerlerde rastlamışsınızdır. Bu sınıf kendisini veri tabanına ekleme, silme, müşteriye email gönderme, login yapma (shop sistemi olabilir) ve sipariş oluşturma

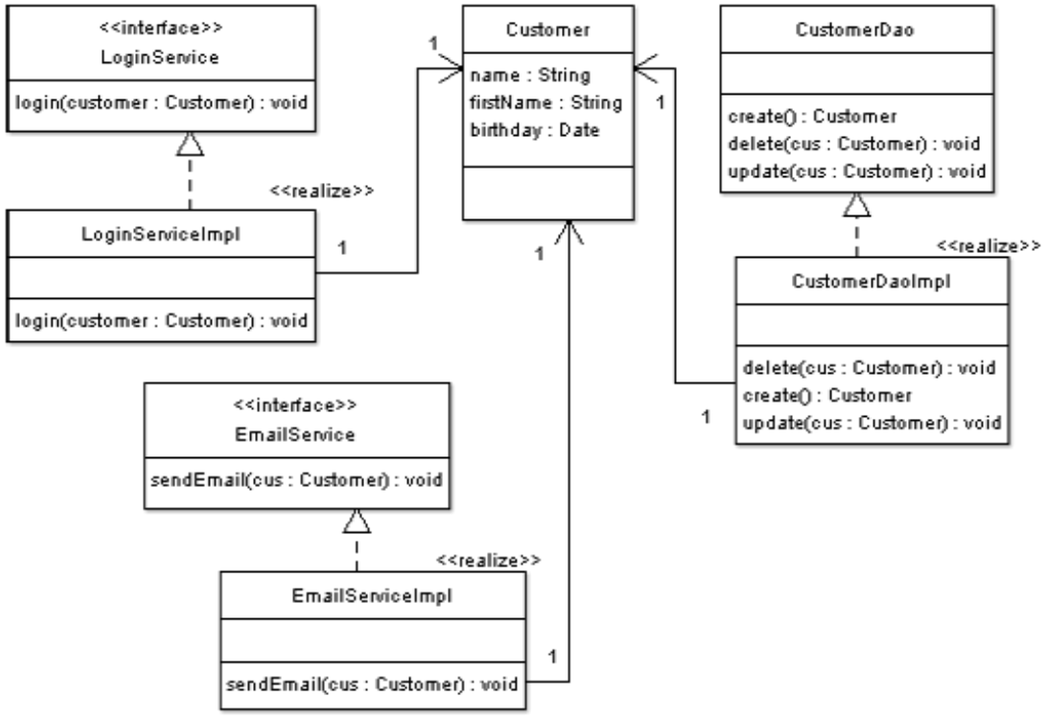
işlemlerini yapabilmektedir.



Resim 4

Büyük bir ihtimalle bu sınıfı programlayan programcı kendisi ile gurur duyuyor olmalıdır. Aslında böyle bir sınıfın ve programcının küçümsenecek hiçbir tarafı yok. Bu sınıf büyük emek harcanarak oluşturulmuş olabilir, çünkü ihtiva ettiği metotlar basit değildir. Lakin bu sınıf saatli bir bombadır, çünkü içinde bulunduğu ortamdaki her değişiklik, sınıfın yapısal değişikliğe uğramasına sebep olabilir. Bunun yanı sıra Customer sınıfında implemente edilen kod tekrar kullanılmak istendiğinde, Customer sınıfının bağımlı olduğu sınıf ve API lerin bu sınıfla beraber kullanılma zorunluluğu vardır, yani Customer sınıfı gittiği yere beraberinde kullandığı diğer sınıflar ve API leri götürür. Bu kodun tekrar kullanımını sağlamak için istenmeyen bir durumdur.

Customer sınıfını test edilebilir ve bakımı kolay bir hale getirmek için sahip olduğu sorumlulukların başka sınıflara yüklenmesi gerekmektedir. Bunun bir örneğini resim 5 de görmekteyiz.



Resim 5

Resim 5 de yer alan Customer sınıfı sahip olduğu sorumlulukların başka sınıflara yüklenmesiyle hafiflemiş ve değişikliklere karşı daha dayanıklı bir hale gelmiştir. Bunun yanı sıra bu sınıfın test edilebilirliği yükselmiştir. Bu ve diğer sınıfların değişmek için artık sadece bir sebebi vardır. Bunun sebebi sadece bir sorumluluk sahibi olmalarında yatmaktadır.

Bir sınıfın sorumluluğunu sadece bir sebepten dolayı değişebilir olması olarak tanımlayabiliriz. Eğer bir sınıfın değişikliğe uğraması için birden fazla sebep varsa, bu sınıf birden fazla sorumluluk taşıyor demektir. Bu durumda sınıfta sadece bir sorumluluk kalacak şekilde sorumlukların diğer sınıflarla paylaşılması yoluna gidilmelidir.

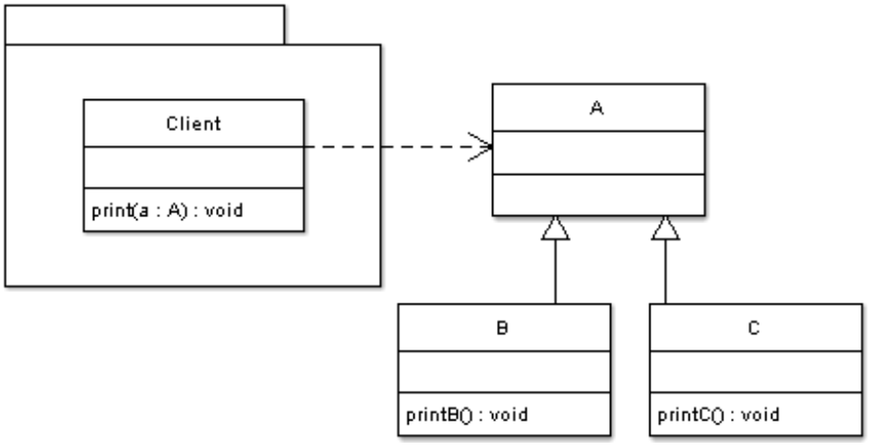
Liskov Substitution Principle (LSP) – Liskov Yerine Geçme Prensibi

Barbara Liskov tarafından geliştirilen bu prensip kısaca şöyle açıklanabilir:

"Alt sınıflardan oluşturulan nesneler üst sınıfların nesneleriyle yer değiştirdiklerinde aynı davranışı göstermek zorundadırlar."

LSP ye göre herhangi bir sınıf kullanıcısı bu sınıfın alt sınıfları kullanmak için özel bir efor sarf etmek zorunda kalmamalıdır. Onun bakış açısından üst sınıf ve alt sınıf arasında farklılık yoktur.

Üst sınıf nesnelerinin kullanıldığı metotlar içinde alt sınıftan olan nesneler aynı davranışı sergilemek zorundadır, çünkü oluşturulan metotlar üst sınıf davranışları örnek alınarak programlanmıştır. Alt sınıflarda meydana gelen davranış değişiklikleri, bu metotların hatalı çalışmasına sebep verebilir. Özellikte bu metotlarda instanceof gibi nesnelerin tipleri arasında kıyaslama yapılmak zorunda kalındığı zaman, LSP prensibi çiğnenmiş olur ki bu alt sınıfların varlığından haberdar olunduğu anlamına gelir. Kullanıcı sınıflar ideal durumda alt sınıfların varlığından haberdar bile olmamalıdır.



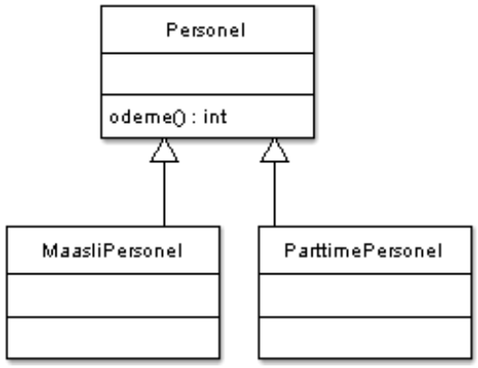
Resim 6

Resim 6 da yer alan Client sınıfındaki print() metodunun nasıl LSP prensibine ters düştüğünü yakından inceleyelim. Bu metot A sınıfından olan nesneler üzerinde işlem yapmaktadır. A sınıfı B ve C sınıfları tarafından genişletilmiştir, yani A sınıfından olan bir parametre nesnesi aynı zamanda B ve C sınıfında da olabilir.

```
// Kod 9

public class Client
{
    public void print(A a)
    {
        if(a instanceof B)
        {
            ((B)a).printB();
        }
        else if(a instanceof C)
        {
            ((C)a).printC();
        }
    }

    public static void main(String[] args)
```

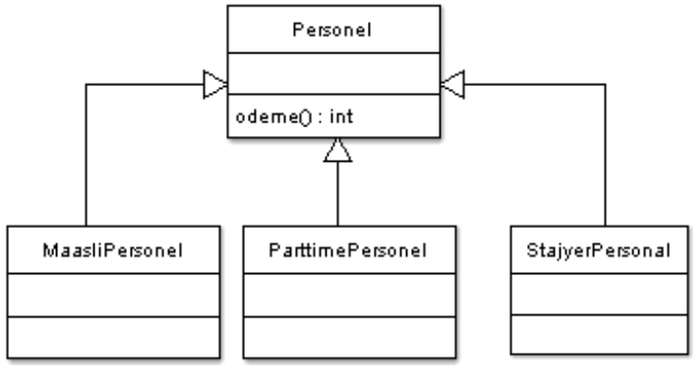
```
    {
        Client client = new Client();
        B b = new B();
        C c = new C();

        client.print(b);
        client.print(c);
    }
}
```

Kod 9 da yer alan print() metodu LSP ile uyumlu değildir. Bu metodun A sınıfına bağımlılığı vardır ve bu sınıfın nesneleri üzerinde işlem yapacak şekilde implemente edilmiş olması gerekir. Lakin print() metodu instanceof komutuyla parametre olarak verilen nesnenin hangi tipte olduğunu tespit etmeye çalışmakta ve tipe bağımlı olarak nesne üzerinde işlemi gerçekleştirmektedir. Bu durum hem OCP ye hemde LSP ye ters düşmektedir. OCP uyumlu değildir, çünkü print() metodu A nin her yeni alt sınıfı için değişikliğe uğramak zorundadır. LSP ye uyumlu değildir, çünkü B ya da C sınıfından olan nesneler A sınıfından olan bir nesnenin yerini alamadığı için print() metodu instanceof ile nesnenin tipini tespit etmek zorunda bırakılmaktadır. Buradan genel bir sonuç çıkartabiliriz: LSP ye ters düşen bir implementasyon aynı zamanda OCP ye de ters düşer.



Resim 7

LSP nin nasıl uygulanabileceğini diğer bir örnekte görelim. Resim 7 de bir firma çalışanları Personel sınıfını genişleten MaasliPersonel ve ParttimePersonel sınıfları ile temsil edilmektedirler. Personel sınıfı soyuttur (abstract). Çalışanların maaşlarının ödenebilmesi için Personel sınıfında bulunan soyut odeme() metodunun alt sınıflarca implemente edilmesi gerekmektedir. Firmaya yeni stajyerler alındığı için modelin resim 8 deki gibi adapte edildiğini farz edelim.



Resim 8

Staj yapan elemanlar için maaş ödenmez, bu yüzden odeme() metodu StajyerPersonel sınıfında boş implemente edilmesi gerekir. İmplementasyon şu şekilde olabilir:

```
// Kod 10

public int odeme(){
    return 0;
}
```

Sıfır değerine geri vererek, odeme() metodunun geçerli ve kullanılabilir bir metot olduğunu ifade etmiş oluyoruz. Bu yanlıştır! Bu metodun StajyerPersonel sınıfında implemente edilmemesi gerekir, çünkü staj yapanlara maaş ödenmez.

Bu sorunu kod 11 de yer aldığı gibi bir Exception oluşturarak çözebiliriz. Eğer bir stajyer için odeme() metodu kullanılacak olursa, oluşan PersonelException, bir stajyer için maaş ödenemeyeceğini metot kullanıcısına bildirir. Exception nesneleri olağan dışı durumları ifade etmek için kullanılır.

```
// Kod 11  - StajyerPersonel.java - odeme() metodu

public int odeme()throws PersonelException
{
    throw new PersonelException("Stajyer maaşlı calismaz!");
}
```

Bu implementasyon ilk örnekte olduğu gibi LSP ile uyumlu değildir. Neden uyumlu olmadığını inceleyelim. Kod 12 de çalışanlara ödenen toplam maaş miktarı hesaplanmaktadır.

```
Kod 12  - Toplam maaş miktarı hesaplanıyor

List<Personel> personel = getPersonelList();
```

```
int total = 0;
for(int i=0; i < personel.size(); i++)
{
    total+=personel.get(i).odeme();
}
```

StajyerPersonal sınıfının sisteme eklenmesiyle kod 12 de yer alan kodun değiştirilmesi gerekmektedir. Ya try/catch bloğu kullanılarak oluşabilecek bir PersonelException ın işleme tabi tutulması gerekir ya da metot imzasında throws kullanılarak PersonelException ın bir üst katmana iletilmesi gerekir. StajyerPersonel ismini taşıyan yeni bir sınıf oluştuğu için mevcut Personel sınıfı kullanıcıları (client) yapısal değişikliğe uğramıştır.

```
// Kod 13 -  try/catch...

try {
    List<Personel> personel = getPersonelList();
    int total = 0;
    for(int i=0; i < personel.size(); i++)
    {
        total+=personel.get(i).odeme();
    }
}
catch (Exception e) {
    // handle exception
}


Kod 14  - instanceof

List<Personel> personel = getPersonelList();
int total = 0;
for(int i=0; i < personel.size(); i++)
{
    if(! (personel.get(i) instanceof StajyerPersonal))
    {
        total+=personel.get(i).odeme();
    }
}
```

Try/catch blokları komplike yapılardır ve kodun okunulabilirliğini azaltırlar. Try/catch bloğundan kurtulmak için kod 14 de yer alan implementasyon düşünülebilir. Burada instanceof ile sırada hangi tip bir nesnenin olduğunu tespit edebilir ve StajyerPersonel tipi nesneleri işlem dışı bırakabiliriz. Daha öncede gördüğümüz gibi bu implementasyon LSP ile uyumlu değildir, çünkü üst sınıf ile çalışan bir metot instanceof ile alt sınıfları tanımak zorunda bırakılmaktadır. Bunun tek

sebebi StajyerPersonel sınıfından olan bir nesnenin, Personel sınıfından bir nesne ile yer değiştiremez durumda olmasıdır. Personel sınıfını kullanan diğer sınıflar alt sınıf olan StajyerPersonel sınıfı sisteme eklendikten sonra bu değişiklikten etkilenmiştir. LSP ye göre bu olmaması gereken bir durumdur. Alt sınıfların nesneleri, üst sınıflardan olan nesnelerin kullanıldığı metotlar içinde üst sınıf nesneleriyle aynı davranışı göstermek zorundadırlar, aksi taktirde kullanıcı sınıflar bu durumdan etkilenirler.

Sorun staj yapan bir elemanın Personel sınıf hirarşisinde yer alan bir sınıf aracılığıyla modellenmiş olmasında yatmaktadır. Staj yapan bir eleman maaş almadığı için personele ait değildir, bu yüzden StajyerPersonel sınıfının Personel sınıfını genişletmesi doğru değildir. Sorunu çözmek ve LSP uyumlu hale gelmek için StajyerPersonel sınıfının Personel sınıf hirarşisinden ayrılması gerekmektedir.

Dependency Inversion Principle (DIP) – Bağımlılıkların Tersine Çevrilmesi Prensibi

Bu prensibe göre somut sınıflara olan bağımlılıklar soyut sınıflar ve interface sınıflar kullanılarak ortadan kaldırılmalıdır, çünkü somut sınıflar sık sık değişikliğe uğrarlar ve bu sınıflara bağımlı olan sınıflarında yapısal değişikliğe uğramalarına sebep olurlar.

Resim 1 de görülen yapı DIP prensibine ters düşmektedir, çünkü RemoteControl sınıfı somut bir sınıf olan TV sınıfına bağımlıdır. TV bünyesinde meydana gelen her değişiklik doğrudan RemoteControl sınıfını etkileyecektir. Ayrıca RemoteControl sınıfını TV sınıfı olmadan başka bir yerde kullanılması mümkün değildir.

Resim 2 de yer alan tasarım DIP ile uyumludur, çünkü RemoteControl sınıfı somut olan TV yerine soyut olan bir interface sınıfına bağımlıdır. Bu tasarım RemoteControl sınıfı ile TV sınıfı arasındaki bağımlılığı yok eder. Ayrıca RemoteControl ve bağımlı olduğu RemoteControlInterface sınıfı beraber başka bir yerde tekrar kullanılabilir.

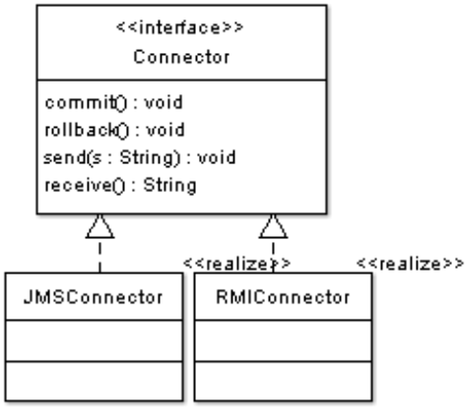
Bu prensibin uygulanması, somut sınıfların kullanımından doğan değişikliklerin azaltılmasını sağlar.

Interface Segregation Principle (ISP) – Arayüz Ayırma Prensibi

Birbiriyle ilişkili olmayan birçok metodu ihtiva eden büyük bir interface sınıf yerine, birbiriyle

ilişkili (cohesive) metotların yer aldığı birden fazla interface sınıfı daha makbuldür.

ISP uygulanmadığı taktirde birden fazla sorumluluğu olan interface sınıflar oluşur. Zaman içinde yüklenen yeni sorumluluklarla bu interface sınıflar daha da büyür ve kontrol edilemez bir hale gelebilir. Bunun bir örneğini resim 9 da görmekteyiz.



Resim 9

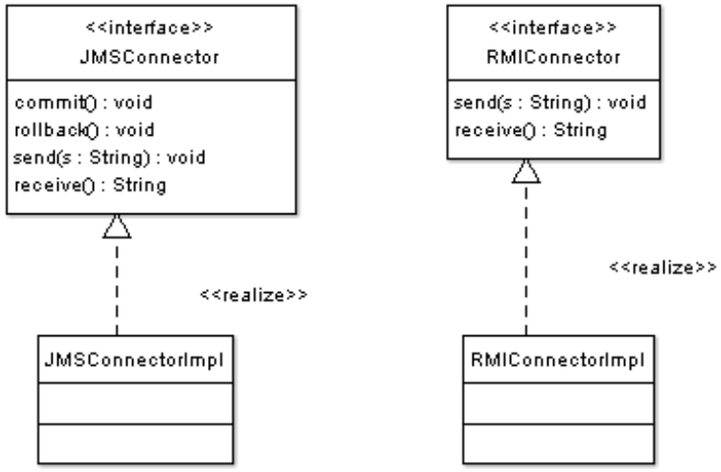
Resim 9 da yer alan Connector interface sınıfı bünyesinde JMS (Java Message Service) için gerekli iki metot bulunmaktadır: commit ve rollback. Bu metotların RMIConnector implementasyonunda implemente edilmeleri anlamsız olur. Büyük bir ihtimalle RMIConnector sınıfında bu metotların implementasyonu kod 15 de yer aldığı şekilde olacaktır.

```
// Kod 15

public class RMIConnector implements Connector
{

    public void commit()
    {
        throw new RuntimeException("not implemented");
    }

    public void rollback()
    {
        throw new RuntimeException("not implemented");
    }
}
```

Kod 15 de yer alan yapı LSP ile uyumlu değildir. Connector sınıfını kullananlar RuntimeException oluşabileceğini göz önünde bulundurmak zorunda bırakılmaktadırlar.

ISP uygulandığı taktirde resim 9 da yer alan Connector interface sınıfını yok ederek, yerine

sorumluluk alanları belli iki yeni interface sınıf oluşturabiliriz. Resim 10 da bu çözüm yer almaktadır.



Resim 10

Programcı olarak bir interface sınıfa birden fazla sorumluluk yüklememek için interface sınıfın sistemdeki görevini iyi anlamamız gerekmektedir. Bu sadece bir sorumluluk alanı olan bir interface sınıf oluşturmamızı kolaylaştıracaktır.

Paket Tasarım Prensipleri (Principles of Package Design)

Bu bölüme kadar tanıştığımız LC, OCP, SRP, LSP, DIP ve ISP sınıflar bazında uygulayabileceğimiz tasarım prensipleridir. Yazılım sistemleri zaman içinde büyüyerek, karmaşık bir yapıya dönüşebilir. Sistemin organizasyonu için paketler (Java'da package) kullanılır. Sınıflarda olduğu gibi paketler arası ilişkiler ve bağımlıklar oluşur. Bunları yönetebilmek için kullanabileceğimiz tasarım prensipleri mevcuttur. Bunlar:

- Reuse-Release Equivalence Principle (REP)
- Common Reuse Principle (CRP)
- Common Closure Principle (CCP)
- Acyclic Dependency Principle (ADP)
- Stable Dependencies Principle (SDP)
- Stable Abstractions Principle (SAP)

Reuse-Release Equivalence Principle (REP) – Tekrar Kullanım ve Sürüm Eşitliği

Program modülleri paketler (packages) kullanılarak organize edilir. Paketler arasında sınıfların birbirlerini kullanmalarıyla bağımlılıklar oluşur. Bunun bir örneği resim 11 de yer almaktadır. Eğer paket B bünyesindeki bir sınıf paket A bünyesinde bulunan bir sınıf tarafından kullanılıyorsa, bu paket A nin paket B ye bağımlılığı olduğu anlamına gelir. Bu tür bağımlılıkların oluşması doğaldır. Amaç bu bağımlılıkları ortadan kaldırmak değil, kontrol edilebilir hale getirmek olmalıdır. Bu amaçla paket bazında uygulanabilecek tasarım prensipleri oluşturulmuştur. Bunlardan birisi Reuse-Release Equivalence (tekrar kullanım ve sürüm eşitliği) prensibidir.

Resim 11 Paket A (package) paket B ye bağımlıdır, çünkü paket A içindeki sınıf ya da sınıflar paket B den bir veya birden fazla sınıfı kullanıyorlar

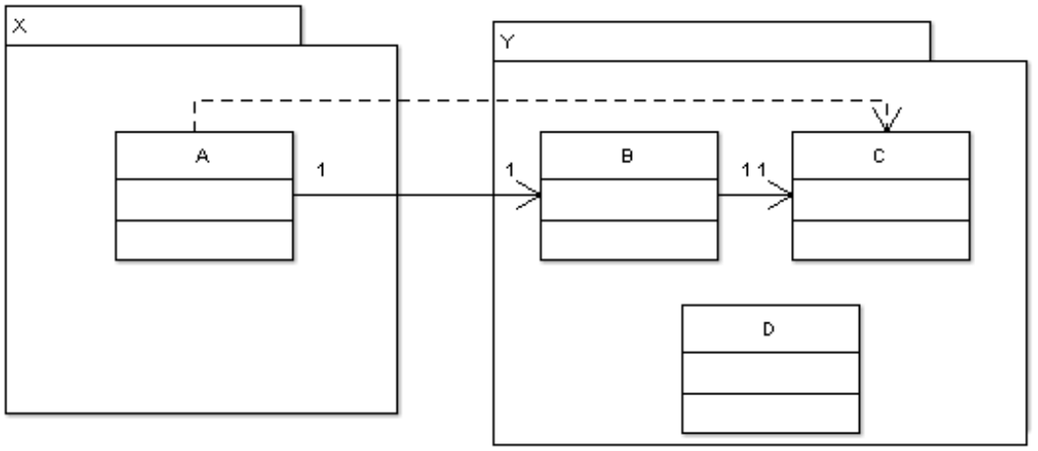
Bir proje bünyesinde değişik modüllerden oluşan bir yazılım sistemini implemente etmek için çalıştığımızı düşünelim. Her modülden ayrı bir ekip sorumlu olsun. Üzerinde çalıştığımız modülün implementasyonunu yapabilmek için büyük bir ihtimalle diğer modülleri kullanmamız gerekecektir. Örneğin veri tabanı üzerinde işlem yapmamızı sağlayacak bir modülü başka bir ekip implemente etmiş olabilir. O ekip veri tabanı üzerindeki işlemler için gerekli tüm sınıfları dao isimli bir paket içine yerleştirmiş olabilir. Bizim bu paket içindeki sınıfları tekrar kullanabilmemiz (reuse) için belirli şartların yerine gelmesi gerekmektedir. Bunlar:

- Veri tabanı ekibi veri tabanı üzerinde işlem yapmak için kullanılan tüm sınıfları, bu sınıflar birbirleriyle ilişkili olduğu için aynı paket içine koymalıdır. Bu tekrar kullanımı kolaylaştırır.
- Tekrar kullanımı desteklemek için oluşturulan paketin bir versiyon numarasıyla yeni sürümünün oluşturulması gerekmektedir.
- Paket kullanıcıları paket üzerinde yapılan değişikliklerden haberdar edilmelidir. Onlar için kullanabilecekleri yeni bir paket sürümünün oluşturulması yanı sıra mevcut kodun kırılmasını önlemek için paketin eski versiyonlarının da paralel kullanıma açık tutulması gerekir. Sadece bu durumda paket kullanıcıları paket üzerinde yapılan değişiklerinden etkilenmeden eski versiyonlarla çalışmaya devam edebilirler. Zaman içinde yeni paket versiyonuna geçerek son değişiklikleri entegre ederler.

Tekrar kullanımı kolaylaştırmak için paket sürümlerinin oluşturulması şarttır. REP e göre tekrar kullanılabilirlik (reuse) sürüm (release) ile doğrudan orantılıdır. Sürüm ne ihtiva ediyorsa, o tekrar kullanılabilir.

Common Reuse Principle (CRP) – Ortak Yeniden Kullanım Prensibi

Bu prensip hangi sınıfların aynı paket içinde yer alması gerektiği konusuna açıklık getirir. CRP ye göre beraberce tekrar kullanılabilir yapıda olan sınıfların aynı paket içinde olması gerekir.



Resim 12 Sınıf A dolaylı olarak sınıf C ye bağımlıdır.

Resim 12 de paket X ve paket Y arasındaki bağımlılık yer almaktadır. Sınıf A paket Y de bulunan B sınıfını kullanmaktadır. B sınıfı aynı paket içindeki C sınıfını kullanmaktadır. Bu durumda X paketinde bulunan A sınıfı dolaylı olarak Y paketinde bulunan C sınıfına bağımlı hale gelmektedir. C sınıfı üzerinde yapılan her değişiklik A sınıfını doğrudan etkileyecektir. B ve C sınıfları beraberce kullanıldıkları için aynı paket içinde yer almaları gerekir.

CRP aynı zaman hangi sınıfların paket içine konmaması gerektiğine de açıklık getirir. Birbirine bağımlılıkları olmayan, birbirini kullanmayan sınıfların aynı paket içinde olmaları sakıncalıdır. Örneğin resim 12 da yer alan Y paketi içindeki D sınıfı, Y paketinde bulunan hiçbir sınıf tarafından kullanılmamaktadır. Bu durumda D sınıfının Y paketinde olmaması gerekir. Eğer D sınıfı değişikliğe uğrar ve Y paketinin yeni bir sürümü (release) oluşursa, bu aslında D sınıfı ile hiçbir ilgisi olmayan X paketindeki A sınıfını etkileyecektir, çünkü Y paketinin yeni sürümü X paketinin ve dolaylı olarak A sınıfının tekrar gözden geçirilmesini zorunlu kılacaktır.

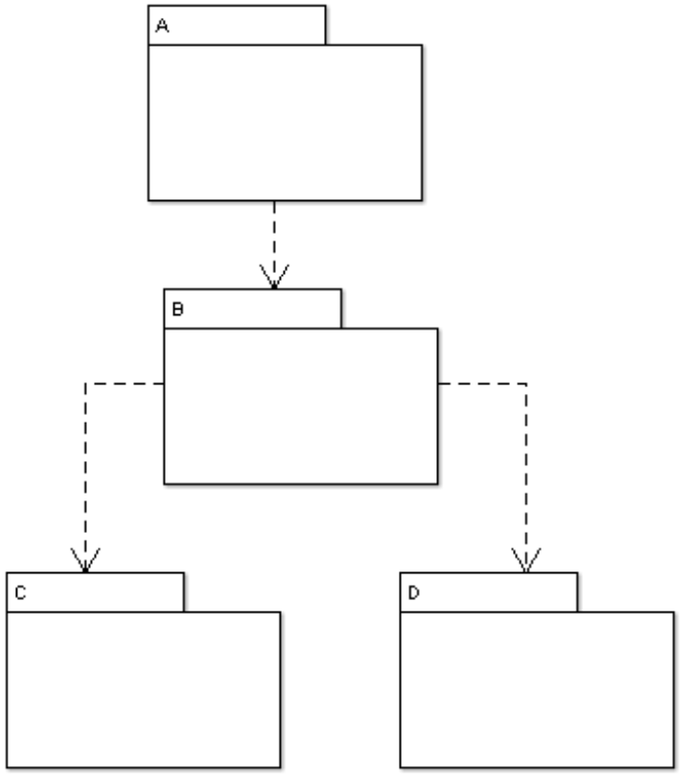
Common Closure Principle (CCP) – Ortak Kapama Prensibi

Yazılım sistemi müşteri gereksinimleri doğrultusunda zaman içinde değişikliğe uğrar. Meydana gelen değişiklerin sistemde bulunan birçok paketi etkilemesi, sistemin bakılabilirliğini olumsuz etkiler. CCP ye göre yapılan değişikliklerin sistemin büyük bir bölümünü etkilemesini önlemek için aynı sebepten dolayı değişikliğe uğrayabilecek sınıfların aynı paket içinde yer alması gerekir. CCP daha önce incelediğimiz sınıflar için uygulanan Single Responsibility (SRP) prensibinin paketler için uygulanan halidir. Her paketin değişmek için sadece bir sebebi olmalıdır. CCP uygulandığı taktirde sistemin bakılabilirliği artırılır ve test ve yeni sürüm için harcanan zaman ve emek

azaltılır.

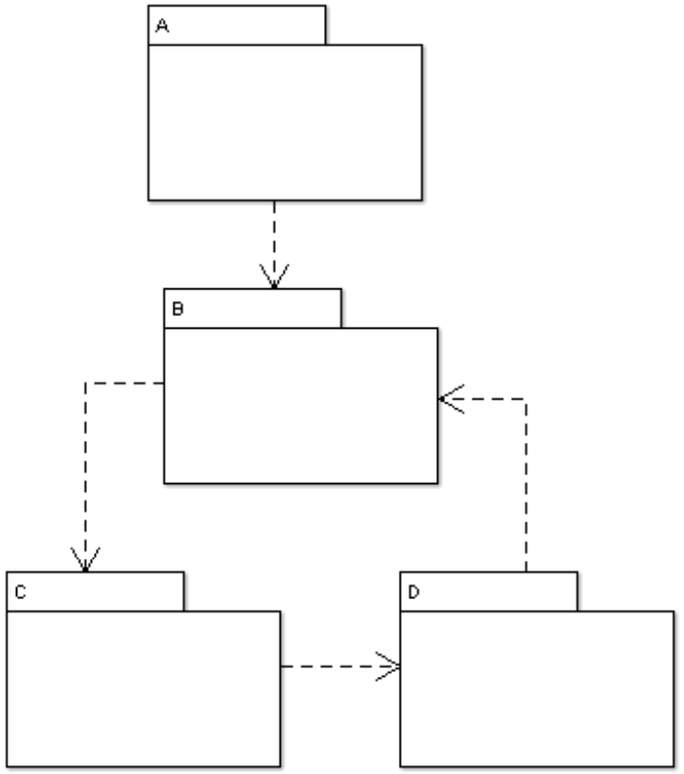
Acyclic Dependency Principle (ADP) – Çevrimsiz Bağımlılık Prensibi

Paketler arasında, çevrim olacak şekilde bağımlılık oluşması sakıncalıdır.



Resim 13 Çevrim olmayan paket yapısı

Resim 13 de yer alan paket yapısında paketlere arası bağımlılıklar mevcuttur, ama çevrim yoktur. Paket A paket B ve dolaylı olarak Paket C ve Paket D ye bağımlıdır. Paket A da bulunan bir sınıfı test etmek istediğimiz zaman Paket B, C ve D yi teste dahil etmemiz gerekmektedir. Bu durum Paket C ve D için geçerli değildir. Bu paketler diğer paketlerden izole edilmiş bir şekilde test edilebilir, çünkü diğer paketlere hiçbir bağımlılıkları yoktur.

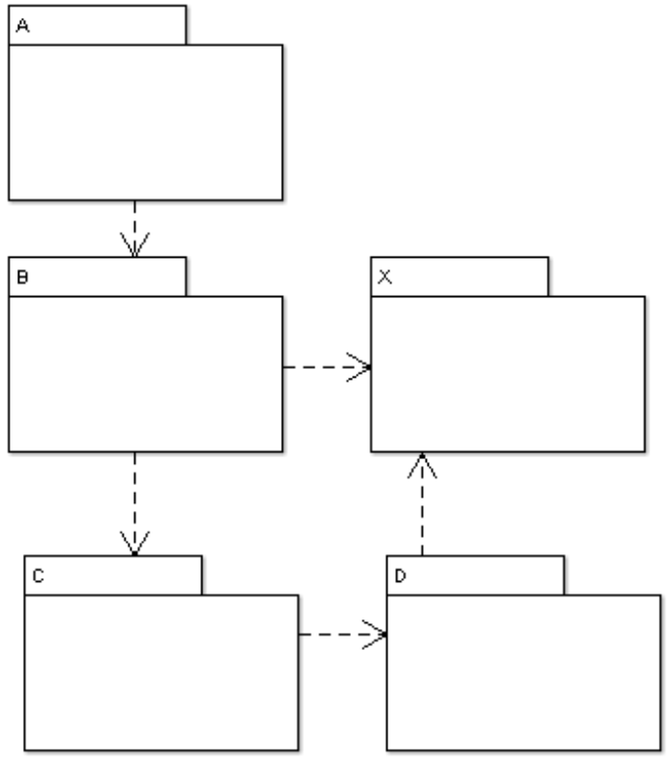


Resim 14 Çevrim olan paket yapısı

Resim 14 de yer alan paket yapısında anomalik bir durum vardır. Eğer B paketinden yola çıkarak bağımlılık yönünde ilerlersek, tekrar bu pakete D paketi üzerinden ulaşabiliriz, yani burada bir çevrim söz konusudur. Bu durumda D paketini test edebilmek için B ve C paketlerine ihtiyacımız vardır.

Çevrim test edilebilirliği zorlaştırdığı gibi dolaylı olarak bağımlılıkları beraberinde getirdiği için yapılan her değişiklikle dolaylı olarak bağımlı olan paketlerin de etkilenmesini sağlar. Bu durum projenin gidişatını zora sokabilir.

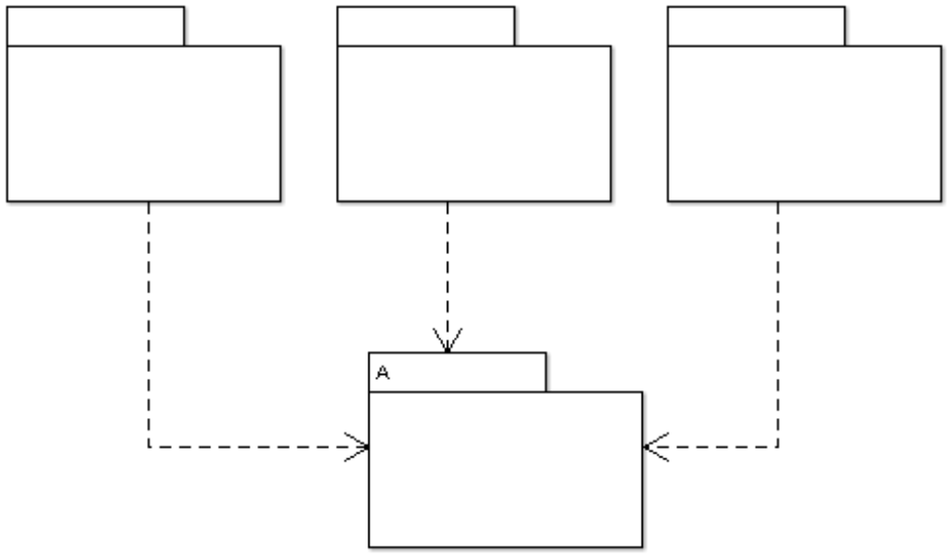
Paketler arası çevrimi yok etmek için paket B ve D nin bağımlı olacağı yeni bir paket oluşturulabilir. Bunun bir örneğini resim 15 de görmekteyiz.



Resim 15 Çevrim olmayan paket yapısı

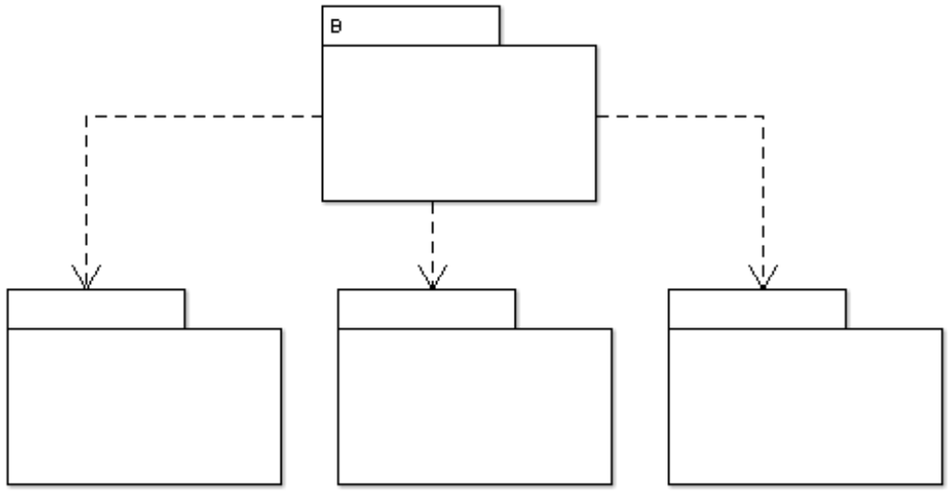
Stable Dependencies Principle (SDP) – Stabil Bağımlılıklar Prensibi

SDP prensini anlayabilmek ve uygulayabilmek için önce paket stabilitesinin ne olduğunu incelememiz gerekiyor.



Resim 16

Resim 16 de yer alan Paket A ya üç bağımlı paket vardır. Bu durumda paket A nın değişikliğe uğramaması için üç sebebi vardır. Paket A bu üç pakete karşı sorumludur (responsible). Bu üç paketi rahatsız etmemek için sık sık değişmemesi gerekir. Bu yüzden paket A stabil ve detiştirilmesi kolay olmayan bir pakettir. Bunun yanı sıra paket A bağımsızdır (independent), çünkü hiçbir dış bağımlılığı olmadığı için değiştirilmek zorunda kalmayabilir.



Resim 17 B stabil olmayan bir pakettir

Resim 17 de durum farklıdır. Üç değişik pakete bağımlılığı bulunan paket B yüksek derecede kırılgan bir yapıdadır. Paket B ye bağımlı olan başka bir paket bulunmamaktadır. Bu yüzden paket B sorumluluğu olmayan (irresponsible) bir pakettir. Bağımlı olduğu her üç paket bünyesinde oluşacak bir değişiklik paket B yi doğrudan etkiler. Bu yüzden paket B bağımlı (dependent) bir pakettir.

SDP paketler arası bağımlılık yönünün stabil paketlere doğru olması gerektiğini söyler. Stabil paketler yapı itibariyle diğer paketlere oranla daha az değişikliğe uğrarlar. Bir paketin stabillik oranını bağımlılık yönü tayin eder. Stabil olmayan bir pakete bağımlılık duyan bir paket doğal olarak değişikliklere maruz kalacaktır ve stabil kalamayacaktır.

Stabilite ölçülebilir bir yazılım metriğidir. Bu metriği elde etmek için kullanabileceğimiz iki değer vardır:

- Afferent Couplings (Ca): Paket içinde bulunan sınıflara bağımlı olan diğer paketlerin adedi
- Efferent Couplings (Ce): Paket içinde bulunan sınıfların bağımlı olduğu diğer paketlerin adedi

Bu metriği testpit etmek için kullanabileceğimiz formül şu şekildedir:

$$I = \frac{Ce}{Ca + Ce}$$

I (instability) paketin stabilite değerini gösteren 0 ile 1 arasında bir değerdir. 0 değerine yaklaşıldıkça paketin stabilitesi artar. 1 değerine yaklaşıldıkça paketin stabilitesi düşer.

SDP prensibine göre bağımlılığı olan bir paketin I metriğinin bağımlı olunan paketin I metriğinden daha yüksek olması gerekir. Bu, bağımlılık yönünün stabil olmayan bir paketten stabil olan bir pakete doğru olması gerektiği anlamına gelmektedir. SDP ile uyumlu paket yapılarında I metriğinin değeri bağımlılık istikametine gidildikçe azalır.

Stable Abstractions Principle (SAP) – Stabil Soyutluk Prensibi

SDP uygulandığı taktirde bağımlılıkların stabil paketlere doğru oluşturulması gerektiğini gördük. Sistemin temelini oluşturan bu paketlerin değişikliklere karşı dirençli olması gerekmektedir. Aksı taktirde sistemin temelinde oluşan bir değişiklik, sistemin tavanına kadar uzanabilecek bir değişiklik zincirini tetikleyebilir.

Paketlerin stabil olmaları sisteme yeni davranış biçimlerinin kazandırılmasının önünde bir engel olmamalıdır. Sisteme yeni davranış biçimlerini abstract ve interface sınıflar kullanarak kazandırabiliriz. SAP, stabil olan paketlerin aynı zamanda soyut sınıflara dayandırılarak, yeniliklere açık olmaları gerektiğini söyler. Sadece soyut olan yapılar değişikliğe karşı direnç gösterebilir. SAP ayrıca stabil olmayan paketlerde somut sınıfların bulunması gerektiğini söyler, çünkü bu özellik somut sınıflarda gerekli olan değişikliklerin yapılabilmesini kolaylaştırmaktadır.

SDP ve SAP nin kombinasyonu paketler için DIP olarak düşünülebilir. SDP stabil olan paketlere doğru bağımlılıklar oluşturmamız gerektiğini, SAP ise stabilliğin paketler için soyutluğu beraberinde getirdiğini söylemekte. Buradan şu sonucu çıkartabiliriz: “ Bağımlılıklar soyutluğa doğru gitmelidir”

Soyutluk ölçülebilir bir metriktir. Bu metriği tespit etmek için kullanabileceğimiz formül şu şekildedir:

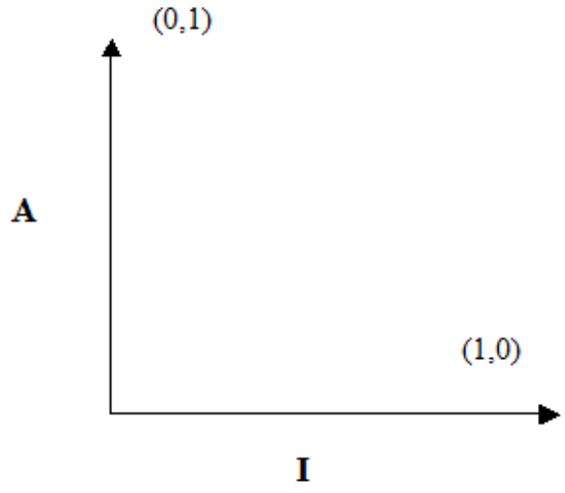
$$A = \frac{Na}{Nc}$$

- A (Abstractness): Soyutluk oranı
- Ne: Paket içinde bulunan sınıfların adedi
- Na: Paket içinde bulunan soyut sınıfların adedi

Bu metrik 0 ile 1 arasında bir değer sahibi olabilir. 0 değeri paket içinde hiçbir soyut sınıfın olmadığının göstergesidir. 1 değeri paketin sadece soyut sınıflardan oluştuğunun göstergesidir. 0 rakamına yaklaşıldıkça somutluk oranı, 1 rakamına yaklaşıldıkça soyutluk oranı artar.

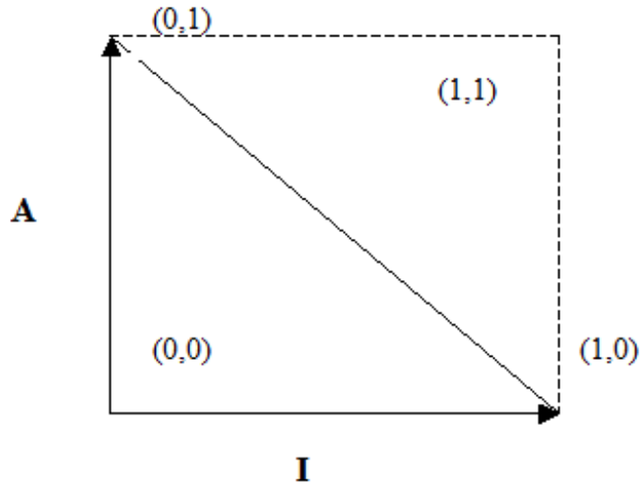
Soyutluk (A) ve Instability (I) Arasındaki İlişki

A ve I metriği arasındaki ilişkiyi incelemek için resim 18 de yer alan koordinat sistemini düşünebiliriz.



Resim 18 A ve I metrikleri

Yatay eksende I metriği, dikey eksende A metriği yer almaktadır. Buna göre 0,1 koordinatında bulunan paketler tamamen soyut ve stabildirler. 1,0 koordinatında bulunan paketler tamamen somut ve instabil yapıdadırlar. Şartlar gereği sistemde bulunan paketlerin tanımladığımız iki kutuptan birinde olması mümkün olmayabilir, örneğin soyut bir sınıfı genişleten bir soyut sınıf soyut (A) olmasına rağmen, genişlettiği sınıfa bağımlılığı olduğu için stabiletisini kaybedebilir. Böyle bir sınıfın 0,1 koordinatında olması mümkün değildir.



Resim 19 A ve I metrikleri

Paketler bazen uç noktalarda (0,1 – 1,0) yer alamayacağına göre paketler için en uygun koordinatların hangileri olduğunu nasıl tespit edebiliriz? Bunun için diğer uç koordinatları da gözden geçirmemiz gerekiyor. 1,1 koordinatında (resim 19) bulunan paketler tamamen soyuttur ve bu paketlere başka paketler tarafından bağımlılıklar mevcut değildir. Böyle paketler kullanılmayan, faydasız paketlerdir. 0,0 koordinatında bulunan paketler tamamen somut ve stabil paketlerdir. Bu paketlerin genişletilmeleri mümkün değildir, çünkü paket içinde soyut sınıflar bulunmamaktadır. Ayrıca bu koordinattaki paketler stabil olduğundan değiştirilmeleri güçtür.

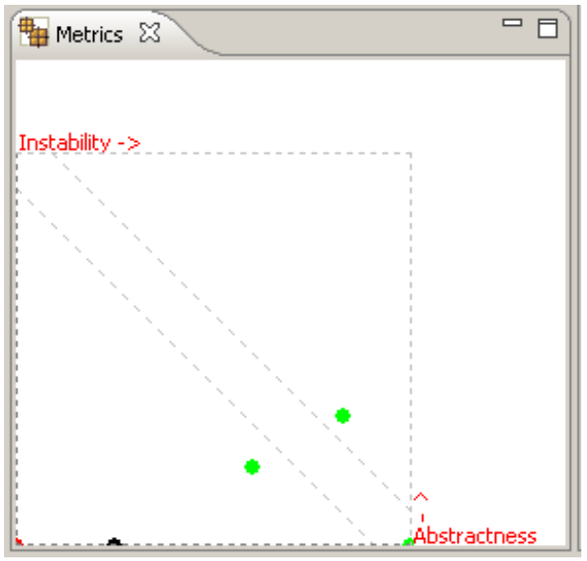
Görüldüğü gibi 1,1 ve 0,0 koordinatlarında ya da bu koordinatlara yakın bir alanda bulunan paketler tasarım eksikliklerine sahiptir. Paketler için en ideal yer 0,1 ve 1,0 koordinatları yanı sıra, bu koordinatları birleştiren çizginin yakınlarında bir yerdir. Bu çizgiye main sequence ismi verilmektedir. Bu çizgiye olan mesafe (D = Distance) şu şekilde hesaplanır.

$$D = |A + I - 1|$$

D metriği 0 ile 1 arasında bir değere sahiptir. 0 değeri paketin doğrudan main sequence çizgisi üzerinde oturduğunu gösterir. 1 değeri paketin bu çizgiden olabildiğince uzakta olduğunu gösterir.

Bu metrik kullanılarak paket yapıları analiz edilebilir. Bir paketin D metrik değeri sıfır yakınlarında değilse, bu paket tekrar gözden geçirilmelidir.

On beşinci bölümde yakından inceleyeceğimiz JDepend ile A, I ve D metriklerini tespit etmek mümkündür. Resim 20 de onuncu bölümde implementasyonunu gerçekleştireceğimiz login modülü için JDepend tarafından oluşturulan main sequence grafiği yer almaktadır.



Resim 20 JDepend

Iyi Bir Tasarım

Iyi bir tasarım oluşturmak için takip edilmesi gereken bazı tasarım prensipleri şunlardir:

- Kalıtım (inheritance) yerine kompozisyon kullanılmalıdır.
- Statik metot ve Singleton yapılar kullanılmamalıdır.
- Bağımlılıkların izole edilmesi gerekir.
- Bağımlılıkların enjekte edilmesi testleri kolaylaştırır.

Kalıtım yerine kompozisyon kullanılmalıdır

Java gibi nesneye yönelik programlama (object oriented = OO) dillerinde kalıtım (inheritance) yöntemi kullanılarak bir sınıf bünyesinde tanımlanan metotlar ve değişkenler değişiklik yapmak zorunda kalmadan alt sınıflara kazandırılır. Bu özelliğin kullanılması avantajlı gibi görüyor olsa da, kalıtım kullanılarak oluşturulan sınıf hiyerarşilerin test edilmesi bakımı ve geliştirilmesi kolay değildir.

Sınıf hirarşileri test için gerekli nesnelerin oluşturulmasını zorlaştırır. Örneğin bir alt sınıftaki herhangi bir metodu test etmek istediğimizi düşünelim. Bu sınıftan bir nesne oluştururken üst sınıfın konstrüktörü geçerli bir parametre kullanılmasını zorunlu kılabilir. Bu durumda üst sınıfı tatmin edecek verilerin oluşturulması gerekmektedir. Bunun karmaşık yapıda bir nesne olabileceğini düşünürsek, bir alt sınıfı test etmek için harcadığımız ekstra efor ortadadır. Bunun yanı sıra oluşturulan testler sınıflar üzerinde değişiklik yapılmasını zorunlu kılabilir. En ufak bir değişiklik bile bir sınıf hirarşisinde ummadık yan etkiler doğurabilir.

Sıkı bir korsa yapısında olan sınıf hirarşileri yerine nesne kompozisyonları tercih edilmelidir. Kompozisyon bir sınıfın bir üst sınıftan miras kalan metotları kullanmak yerine görevi başka bir sınıfta bulunan metoda delege etmesiyle meydana gelen yapıdır. Strateji tasarım şablonunda olduğu gibi kompozisyonda kullanılan sınıfın değişik implementasyonları oluşturulabilir. Alternatif bir implementasyon ile test daha kolay bir hal alabilir. Bunun yanı sıra kompozisyon kodun tekrar kullanımını (reuse) kolaylaştırır.

Statik metot ve tekil yapılar kullanılmamalıdır

Test edilebilirliğin önündeki diğer bir engel statik metotlar ve tekil (singleton) nesnelerdir. Böyle yapıların test esnasında simüle edilmesi çok zor bir hal alabilir. Statik metotlar ve singleton sınıflar test içinde ait oldukları sınıfların isimlerini kullanmayı gerektirirler. Bu durumda bu sınıfa olan bağımlılığı test bünyesinde başka türlü simüle etme şansımız yoktur.

```
// Kod 16

public class PdfCreator
{
    public static boolean create()
    {
        // create a pdf file
    }
}

// Kod 17

import junit.framework.TestCase;

public class PdfCreatorTest extends TestCase
{
    public void testPdfCreator()
    {
        assertTrue(PdfCreator.create());
    }
}
```

Kod 16 da yer alan PdfCreator.create() metodunu PdfCreatorTest.testPdfCreator() metodunda test edebilmemiz için PdCreator sınıf ismini test içinde kullanmamız gerekiyor, çünkü create() statik bir metottur. Bu test metodu ile PdfCreator sınıfı arasında bir bağımlılık oluşturuyor ve bizim create() metodunu örneğin bir mock nesne ile simüle etmemizi engelliyor. Burada PdfCreator sınıfını kalıtım ile genişletip, create() metodunu yeniden implemente etsek bile, PdfCreator ve test metodu arasındaki bağımlılıktan dolayı yeni sınıfı kullanmamız mümkün değildir.

Testleri kolaylaştırmak için statik metotların yok edilmesinde fayda vardır. Çoğu zaman metot ismi önündeki static kelimesini kaldırarak, statik metotları sınıf metotlarına dönüştürebiliriz.

Singleton nesnelerin test edilebilirliklerini yükseltmek için ihtiyaç duydukları verileri dış dünyadan enjekte edebiliriz. Bu durum test esnasında istediğimiz tipte tekil nesnenin oluşmasını kolaylaştıracaktır. Bunu gerçekleştirebilmek için bağımlılıkları izole etmemiz gerekiyor.

Bağımlılıkların izole edilmesi gerekir

Test edilen sınıfın sahip olduğu bağımlılıkları test esnasında başka bir implementasyon ile değiştirebilmek için bu bağımlılıkların izole edilmesi gerekmektedir.

```
// Kod 17

public class PdfCreator
{
    public Pdf create()
    {
        PdfCreatorService service =
            PdfCreatorService.getInstance();
        ...
    }
}
```

Kod 17 de yer alan create() metodu bir Pdf dosya oluşturabilmek için PdfCreatorService singleton sınıfını kullanmaktadır. create() metodunu bu hali ile test etmek çok güçtür, çünkü PdfCreatorService bir singleton sınıf olduğu için bünyesinde olup bitenleri kontrol etme şansımız yoktur. Örneğin bu singleton sınıf private olan konstrüktöründe bir veri tabanına bağlanarak gerekli konfigürasyonları ediniyor olabilir. Bu durumda create() metodunu test etmek için oluşturduğumuz testlerde veri tabanı bağımlılığı oluşacaktır. Bu istenmeyen bir durumdur.

create() metodundaki bağımlılığı kod 18 deki gibi izole ederek, değiştirilebilir hale getirebiliriz.

```
// Kod 18

public class PdfCreator
{
    public Pdf create()
    {
        PdfCreatorService service = getService();
        ...
    }
```

```
    protected PdfCreatorService getService()
    {
        return PdfCreatorService.getInstance();
    }
}
```

getService() isminde yeni bir metot oluşturarak gerekli servis sınıfının edinilme işlemini izole etmiş oluyoruz. Test esnasında bu izole edilen bölümü değiştirerek, create() metodunu test edilebilir hale getirebiliriz.

```
// Kod 19

import junit.framework.TestCase;

public class PdfCreatorTest extends TestCase
{
    public void testCreatePdf()
    {
        PdfCreator creator = new PdfCreator()
        {
            protected PdfCreatorService getService()
            {
                return new DummyPdfCreatorService();
            }
        };

        assertNotNull(creator.create());
    }
}
```

PdfCreator sınıfından yeni bir nesne oluştururken getService() metodunu yeniden yapılandırabiliriz. Bu bize istediğimiz service implementasyonunu kullanma fırsatı verir. Bunun bir örneği kod 19 da yer almaktadır. DummyPdfCreatorService test için oluşturduğumuz bir mock sınıftır. Bu sınıfın kullanımı create() metodunun test edilmesini kolaylaştırmaktadır.

Bağımlılıkların enjekte edilmesi testleri kolaylaştırır

Kod 18 de PdfCreatorService sınıfına doğrudan bağımlı olan PdfCreator sınıfı yer almaktadır. Bu bağı test esnasında kod 19 de yer aldığı gibi yeni bir implementasyon ile çözebiliriz. Bu işlemi daha kolay hale getirmek için başka bir yöntem daha mevcuttur.

İki sınıf arasındaki ilişkinin daha esnek bir hale gelmesini sağlamak ve testleri kolaylaştırmak için

bağımlılıkların enjekte edilmesi (dependency injection) metodunu kullanabiliriz. Bunun bir örneğini kod 20 de görmekteyiz.

```
// Kod 20

public class PdfCreator
{

    private PdfCreatorService service;

    public Pdf create()
    {
        PdfCreatorService service = getService();
        ...
    }

    public PdfCreatorService getService()
      {
        return service;
    }

    public void setService(PdfCreatorService service)
      {
        this.service = service;
    }
}
```

PdfCreatorService sınıfından olan bir değişkeni sınıf değişkeni (service) olarak tanımlıyoruz. setService() metodu ile bu değişkenin değerini değiştirebiliriz. Bu bize test esnasında istediğimiz bir implementasyonu kullanma imkanı tanıyacaktır. Bunun nasıl yapıldığı kod 21 de yer almaktadır.

```
// Kod 21

import junit.framework.TestCase;

public class PdfCreatorTest extends TestCase
{
    public void testCreatePdf()
    {
        PdfCreator creator = new PdfCreator();
        DummyPdfCreatorService service =
            new DummyPdfCreatorService();
        creator.setService(service);
        assertNotNull(creator.create());
    }
```

```
}
```

Test metodunda istediğimiz türde bir PdfCreatorService implementasyonu oluşturarak, setService() metoduyla bu bağımlılığı creator nesnesine enjekte ediyoruz. Böylece creator nesnesi iç dünyasında getService() metodu aracılığıyla setService() ile enjekte ettiğimiz implementasyonu kullanır hale geliyor.

Dependency injection metodunun kullanımı testlerin sınıflar üzerindeki kontrolünü güçlendirir. Bu sınıfların test edilebilirliğini artırır.

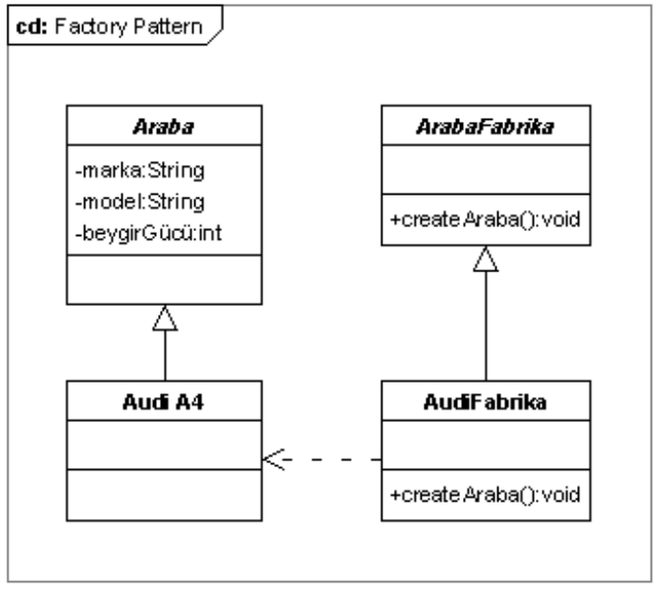
5. Bölüm

Oluşturucu Tasarım Şablonları - Creational Patterns

Fabrika (Factory)

Bir sınıftan yeni bir nesne oluşturulduğunda kullanıcı sınıf hangi sınfı kullandığını bilir ve nesne oluşturma sürecini yönetmiş olur. Bu bağımlılığı ortadan kaldırmak için fabrika tasarım şablonu kullanılabilir. Fabrika sınıfı kullanılan nesneyi oluşturmakla sorumludur. Soyut sınıflar kullanıldığı taktirde, kullancı sınıf kendisine fabrika tarafından hangi implementasyonunun verildiğini bile bilmez. Bu şekilde kullanıcı sınıfa değişik türde implementasyonlar verilerek, uygulamanın esnekliği artırılabilir.

Fabrika tasarım şablonunun nasıl kullanıldığını ilk örneğimizde görelim. Araba üretiminde kullanılmak üzere ufak bir çatı (framework) oluşturacağız. Amacımız araba üretim sürecini soyut bir şekilde tanımlamak ve oluşturduğumuz çatıyı araba üretimi yapan firmalara satmak. Araba üreticisi firma çatı içinde tanımlanmış olan soyut sınıflar yardımı ile üretimi yapılacak olan modelleri oluşturacaktır. Çatının yapısı resim 1 de yer alan UML (Unified Modelling Language) diyagramında görülmektedir.



Resim 1

Araba ve ArabaFabrika isimlerinde iki soyut sınıf tanımlıyoruz.

Kod 1 de yer alan Araba sınıfı üretilecek arabanın özelliklerini ihtiva etmektedir. Sınıf soyut (bknz. public abstract class Araba) olarak tanımlanmıştır. Daha öncede belirttiğim gibi, çatı araba üretim sürecini sadece soyut olarak tanımlamak ya da başka bir terimle modellemek için kullanılmaktadır. Kod 1 de yer alan Araba sınıfını genişleterek, somut araba sınıfları oluşturabliriz. Fabrika tasarım şablonunu bu somut araba nesnelerini oluşturmak için kullanacağız.

```
// Kod 1

package com.pratikprogramci.designpatterns.bolum5.factory;

public abstract class Araba {
    private String marka = null;
    private String model = null;
    private int beygirGucu = 0;

    public Araba(final String marka, final String model, final int beygirGucu) {
        setMarka(marka);
        setModel(model);
        setBeygirGucu(beygirGucu);
    }

    public String getMarka() {
        return marka;
    }

    public void setMarka(final String marka) {
        this.marka = marka;
    }

    public String getModel() {
        return model;
    }

    public void setModel(final String model) {
        this.model = model;
    }

    public int getBeygirGucu() {
        return beygirGucu;
    }

    public void setBeygirGucu(final int beygirGucu) {
        this.beygirGucu = beygirGucu;
    }
}
```

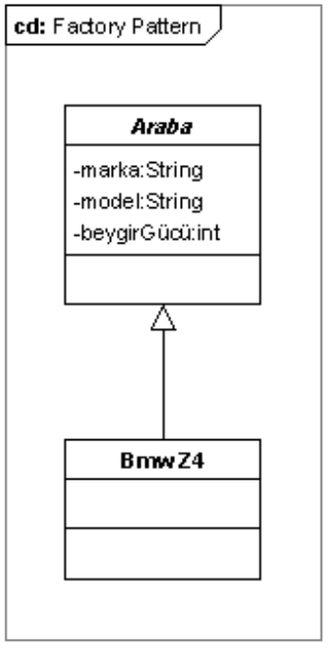
ArabaFabrika sınıfı, Araba sınıfı gibi soyut olarak tanımlanmıştır. Araba üretim süreci bu sınıf tarafından kontrol edilmektedir. Belirli bir marka ve model araba üretebilmek için bu sınıfın altsınıflarının oluşturulması gerekmektedir.

```
// Kod 2
```

```
package com.pratikprogramci.designpatterns.bolum5.factory;

import java.util.ArrayList;

/**
 * Genel bir araba fabrikasini tanımlar. Soyut olduğu için bu sınıftan nesneler
 * oluşturulamaz. Belirli bir araba marka ve modelini üretebilmek için bu
 * sınıfın altsınıfı oluşturulması gerekmektedir
 *
 */
public abstract class ArabaFabrika {

    /**
     * Bir araba fabrikasının ürettiği değişik modeldeki arabalarin içinde
     * yer aldığı liste.
     */
    private ArrayList<Araba> arabaListesi = new ArrayList<Araba>();

    /**
     * Altsınıflarda bir fabrikanın (nesnenin) oluşturulması ile beraber,
     * createAuto() metodu işleme girer, yani fabrika araba üretime başlamış
     * olur.
     */
    public ArabaFabrika() {
        createAuto();
    }

    /**
     * Alt sınıflar tarafından implemente edilir. Belirli bir marka ve modelin
     * olusturulmasinda kullanılır.
     */
    public abstract void createAuto();

    public ArrayList<Araba> getArabaListesi() {
        return arabaListesi;
    }

    public void setArabaListesi(final ArrayList<Araba> arabaListesi) {
        this.arabaListesi = arabaListesi;
    }
}
```

Araba ve ArabaFabrika sınıfları ile çatımız için gerekli iki sınıfı tanımlamış olduk. Şimdi Bmw ve Audi firmalarının bu çatıyı kullanarak nasıl araba üretimi yaptıklarına göz atalım.

Bmw Z4 marka bir arabayı üretebilmek için Araba sınıfını üstsınıf olarak seçmemiz gerekiyor.



Resim 2

```
// Kod 3

package com.pratikprogramci.designpatterns.bolum5.factory;

/**
 * Bmw Z4 Model araba.
 *
 */
public class Z4 extends Araba {

    public Z4(final int beygirgucu) {
        super("BMW", "Z4", beygirgucu);
    }
}
```

Z4 model arabaların üretimini BmwFabrika ismini taşıyan fabrika yapıyor.

```
// Kod 4

package com.pratikprogramci.designpatterns.bolum5.factory;

/**
 * Bmw marka arabaları üretir. ArabaFabrika sınıfının altsınıfı olduğu için
```

```
 * createAuto metodunu implemente eder.
 *
 */
public class BmwFabrika extends ArabaFabrika {

    /**
     * ArabaFabrika sınıfında yer alan createAuto() motodu BmwFabrika
     * altsınıfında implemente edilir. Bu metot Bmw markasina ait Z4 model
     * arabayı üretmek için kullanılır.
     */
    @Override
    public void createAuto() {
        getArabaListesi().add(new Z4(170));
    }
}
```

Bmw marka arabalar yanı sıra Audi marka araba üretimi içinde bizim çatımız kullanılmakta. A4 ve R8 model araba üretimi için gerekli sınıfları tanımlıyoruz.

```
// Kod 5

package com.pratikprogramci.designpatterns.bolum5.factory;


/**
 * Audi A4 Model araba.
 */
public class A4 extends Araba {

    public A4(final int beygirgucu) {
        super("Audi", "A4", beygirgucu);
    }
}


package com.pratikprogramci.designpatterns.bolum5.factory;

/**
 * Audi R8 Model araba.
 *
 */
public class R8 extends Araba {
    public R8(final int beygirgucu) {
        super("Audi", "R8", beygirgucu);
    }
}
```

```
// Kod 6

package com.pratikprogramci.designpatterns.bolum5.factory;

/**
 * Audi marka arabaları üretir. ArabaFabrika sınıfının altsınıfı olduğu için
 * createAuto metodunu implemente eder.
 *
 */
public class AudiFabrika extends ArabaFabrika {

    /**
     * ArabaFabrika sınıfında yer alan createAuto() motodu Audi altsınıfında
     * implemente edilir. Bu metot Audi markasina ait A4 ve R8 modellerinin
     * üretimi için kullanilmaktadir.
     */
    @Override
    public void createAuto() {
        getArabaListesi().add(new A4(120));
        getArabaListesi().add(new R8(350));
    }
}
```

Çatıyı ve oluşturduğumuz diğer sınıfları test etmek amacıyla kod 7 de yer alan Test sınıfını kullanıyoruz.

```
// Kod 7

package com.pratikprogramci.designpatterns.bolum5.factory;

/**
 * Factory tasarım şablonunu test etmek için kullanılan sınıf.
 *
 */
public class Test {

    public static void main(final String[] args) {

        /*
         * Bmw marka arabalarin uretildiği bir fabrika oluşturuyoruz.
         */
        ArabaFabrika bmw = new BmwFabrika();

        /*
```

```
         * Audi marka arabalarin uretildiği bir fabrika oluşturuyoruz.
         */
        ArabaFabrika audi = new AudiFabrika();

        /*
         * Üretilen Bmw marka arabaları ekranda görüntülüyoruz.
         */
        for (final Araba araba : bmw.getArabaListesi()) {
            System.out.println(araba.getMarka() + " " + araba.getModel() + ", "
                    + araba.getBeygirGucu());
        }

        /*
         * Üretilen Audi marka arabaları ekranda görüntülüyoruz.
         */
        for (final Araba araba : audi.getArabaListesi()) {
            System.out.println(araba.getMarka() + " " + araba.getModel() + ", "
                    + araba.getBeygirGucu());
        }
    }
}
```

Bmw ve Audi marka araba üretimi yapabilmek için BmwFabrika ve AudiFabrika sınıflarından fabrika nesneleri oluşturuyoruz. ArabaFabrika soyut bir sınıf olduğu için bu sınıftan doğrudan nesne oluşturulamaz. Ama BmwFabrika ve AudiFabrika somut sınıflar olduklarından ve ArabaFabrika sınıfından türetildikleri için nesne oluşumunda kullanılabilirler.

Test sınıfı çalıştırıldığında aşağıdakı ekran çıktısı oluşur:

```
BMW Z4, 170
Audi A4, 120
Audi R8, 350
```

New operatörü ile bir ArabaFabrika oluşturulduğunda, örneğin BwmFabrika, otomatik olarak bu marka otomobil üretimine başlanır, çünkü ArabaFabrika soyut sınıfının yapılandırıcı metodu (class constructor) createAuto() metodunu kullanmaktadır. createAuto() fabrika metodu olarak isimlendirilir, çünkü bu metot bünyesinde gerekli araba nesneleri üretilmektedir.

createAuto() metodunu ArabaFabrika sınıfında soyut olarak tanımlamıştık, yani bu metodun gövdesi yoktu. ArabaFabrika sınıfından türetilecek altsınıfların mutlaka createAuto() metodunu implemente etmeleri gerekiyor. Örneğin BmwFabrika sınıfı bu işlemi aşağıdakı şekilde yapıyor:

```
/**
```

```
 * ArabaFabrika sınıfında yer alan createAuto() motodu BmwFabrika
 * altsınıfında implemente edilir. Bu metot Bmw markasina ait Z4 model
 * arabayı üretmek için kullanılır.
 */
@Override
public void createAuto() {
    getArabaListesi().add(new Z4(170));
}
```

Çatı içinde tanımladığımız soyut sınıflar genel anlamda araba üretim sürecini modellemiş oldu. Belirli bir marka ve model araba üretimi yapabilmek için marka ve modeli temsil eden altsınıfların oluşturulması gerekmektedir. Fabrika tasarım şablonunu kullanarak, Test sınıfında da gördüğümüz gibi, hangi Araba altsınıflarının kullanıldıklarını (örneğin A4, Z4) bilmek zorunda olmadan Bmw ve Audi marka otomobil üretimi yapabildik. Fabrika tasarım şablonu ile somut araba sınıflarının oluşturulma işlemini kullanıcı sınıftan (Test) gizlemiş oluyoruz. Bu kullanıcı sınıf ile kullanılan sınıf arasındaki bağımlılığı azaltmakta ve kullanılan sınıfın kolaylıkla başka bir implementasyon ile değiştirilebilirliğini mümkün kılmaktadır.

Eğer fabrika tasarım şablonunu kullanmamış olsaydık, kullanıcı sınıf (Test) şu şekilde olmak ve tüm araba altsınıflarını tanımak zorunda kalırdı:

```
// Kod 8

package com.pratikprogramci.designpatterns.bolum5.factory;

public class Test2 {

    public static void main(final String[] args) {

        AudiFabrika audiFabrika = new AudiFabrika();
        audiFabrika.getArabaListesi().add(new A4(120));
        audiFabrika.getArabaListesi().add(new R8(350));

        showCar(audiFabrika);

        BmwFabrika bmwFabrika = new BmwFabrika();
        bmwFabrika.getArabaListesi().add(new Z4(120));

        showCar(bmwFabrika);

    }

    private static void showCar(ArabaFabrika arabaFabrika) {
        for (final Araba araba : arabaFabrika.getArabaListesi()) {
```

```
            System.out.println(araba.getMarka() + " " + araba.getModel() + ", "
                    + araba.getBeygirGucu());
        }
    }
}
```

Verilen örneklerde görüldüğü gibi fabrika sınıfı kullanıcı ve kullanılan sınıf arasında set vazifesi görmektedir. Bu altsınıfların gizlenmesini ve hangi altsınıfların var olduğu bilgisinin kullanıcı sınıflarca göz ardı edilmesini mümkün kılmaktadır.

Fabrika tasarım şablonu ne zaman kullanılır?

- Kullanıcı sınıf hangi altsınıfların kullanılması gerektiğini bilmiyorsa; Örneğin Test sınıfı Z4 ya da A4 model arabaları tanımıyor.
- Kullanıcı sınıf altsınıflardan nasıl nesne üretilmesi gerektiğini bilmiyorsa; Örneğin Test sınıfı Z4 model bir araba üretebilmek için Z4 sınıfının kontrüktörünün bir int değerine ihtiyaç duyduğunu bilmiyor.
- Kullanıcı sınıf ihtiyaç duyulan nesnelerin oluşturulmasını altsınıflara delege etmek istiyorsa; Örneğin Test sınıfı A4 ve R8 model Audi marka araba üretimini tamamen AudiFabrika sınıfına bırakıyor.

İlişkili tasarım şablonları

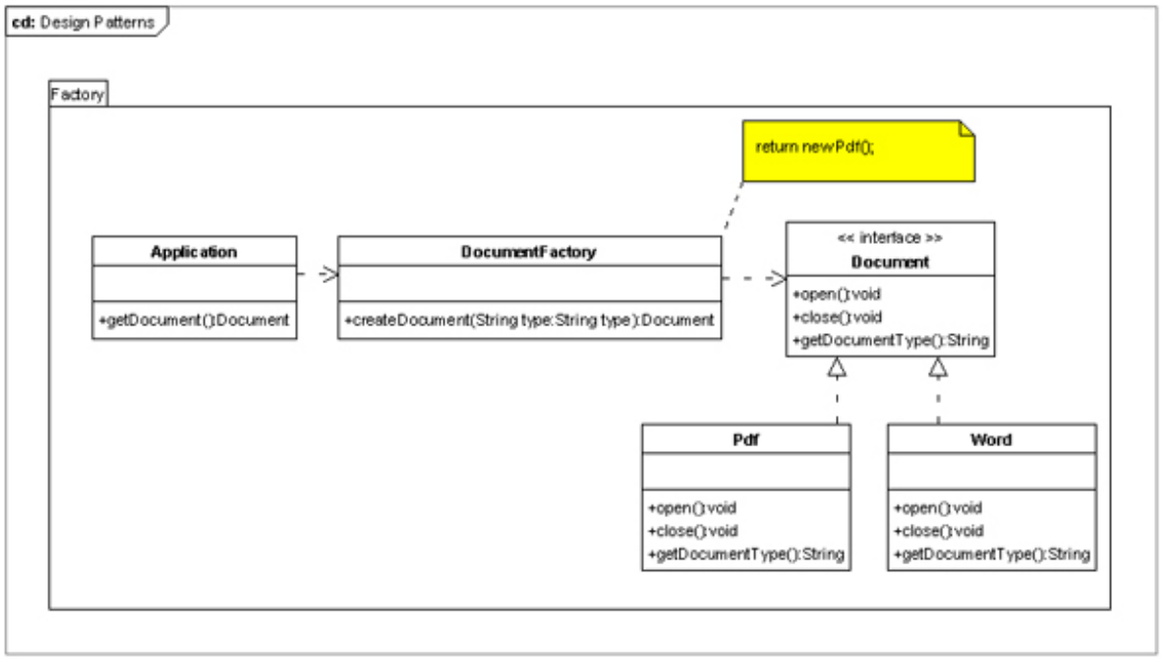
Builder, prototype, abstract factory ve factory method tasarım şablonları fabrika tasarım şablonu ile benzerlik gösterirler ve yeni nesne oluşturmak için kullanılırlar.

Fabrika Metodu (Factory Method)

Yukarıda yer alan örnekler aracılığı ile fabrika tasarım şablonunun ne olduğunu ve nasıl kullanıldığını görmüş olduk. Nesne oluşturma işlemini somut bir sınıfın somut bir metodu aracılığı ile de yapmak mümkün. Fabrika tasarım şablonu denildiğinde aslında akla gelen tasarım şablonu fabrika metodudur. Bu metot bünyesinde nesne oluşturulur ve kullanıcı sınıfa verilir. Kullanıcı sınıfın nesnenin hangi sınıftan olduklarını bilme zorunluluğu yoktur. Bu şekilde kullanıcı sınıfları değiştirmek zorunda kalmadan, fabrika metodu bünyesindeki nesne oluşum süreci ve nesnenin veri tipi tayin edilebilir.

Şimdi fabrika metodu tasarım şablonunu bir örnek üzerinde inceleyelim. Bir projede çeşitli tiplerde dökümanların oluşturulması gerektiğini düşünelim. Bu amaçla Document isminde bir interface tanımlıyoruz. Bu interface sınıfının getDocumentType() isminde bir metodu var. Bu interface

sınıfını implemente edecek olan altsınıfların getDocumentType() metoduna sahip olmaları gerekiyor. İstenilen tipteki dökümanların oluşturulması görevi DocumentFactory ismini taşıyan sınıfa verilmiştir. Sınıflar arası ilişkiyi resim 3 de yer alan UML diyagramında görmekteyiz.



Resim 3

Application sınıfı istediği tipte bir dökümanın oluşturulması için DocumentFactory sınıfının createDocument(String type) metodunu kullanıyor. Application sınıfının bilmesi gereken tek şey, DocumentFactory sınıfı tarafından oluşturulan dökümanın Document tipinde olmasıdır. Document bir interface olduğu için DocumentFactory tarafından oluşturulup, geri verilen tüm döküman nesnelerinde Document interface sınıfında tanımlanmış olan metotlar olacaktır, çünkü bu nesneler Document interface sınıfını implemente etmektedirler.

Sistemde Pdf ve Word isminde Document interface sınıfını implemente eden iki döküman sınıfı bulunmaktadır.

Document interface sınıfının yapısı aşağıdaki gibidir:

```
// Kod 9

package com.pratikprogramci.designpatterns.bolum5.factory.ornek2;

/**
 * Değişik türde döküman türleri oluşturmak için tanimlanan sınıf.
 *
 */
```

```
public interface Document {

    /**
     * Dökümanın tipini verir.
     *
     * @return String döküman tipi (pdf, word, ....)
     */
    public String getDocumentType();
}
```

Değişik tipte döküman nesneleri oluşturmak için DocumentFactory sınıfını kullanıyoruz. Bu sınıf fabrika tasarım şablonunun implementasyonudur.

```
// Kod 10

package com.pratikprogramci.designpatterns.bolum5.factory.ornek2;

/**
 * DocumentFactory sınıfı bir fabrika tasarım şablonu implementasyonudur.
 *
 */
public class DocumentFactory {

    /**
     * Kullanıcının istediği tipte bir döküman oluşturur
     */
    public static Document createDocument(final String type) {
        if (type.equals("PDF")) {
            return new Pdf();
        } else if (type.equals("WORD")) {
            return new Word();
        } else {
            throw new RuntimeException("Doküman tipi belli değil!");
        }
    }
}
```

Gerçek bir dökümanı temsil eden Pdf ve Word sınıfları aşağıdakı yapıya sahiptir:

```
// Kod 11

package com.pratikprogramci.designpatterns.bolum5.factory.ornek2;

/**
 * Bir PDF dosyasini temsil eden sınıf.
```

```
 *
 */
public class Pdf implements Document {

    public String getDocumentType() {
        return "Pdf";
    }
}

package com.pratikprogramci.designpatterns.bolum5.factory.ornek2;

/**
 * Bir Word dosyasini temsil eden sınıf.
 *
 */
public class Word implements Document {

    public String getDocumentType() {
        return "Word";
    }
}
```

Değişik tipte döküman nesneleri üretmek için aşağıdakı sınıfı kullanıyoruz:

```
// Kod 12

package com.pratikprogramci.designpatterns.bolum5.factory.ornek2;


/**
 * Değişik tipte döküman oluşturmak için kullanılan uygulama.
 *
 */
public class Application {

    public static void main(final String[] args) {

        /**
         * Bir pdf dökümanı oluşturuyoruz
         */
        Document document = DocumentFactory.createDocument("PDF");
        System.out.println(document.getDocumentType());

        /**
         * Bir word dökümanı oluşturuyoruz
         */
```

```
        document = DocumentFactory.createDocument("WORD");
        System.out.println(document.getDocumentType());
    }
}
```

Application.main() içinde önce pdf tipinde bir döküman nesnesi oluşturuyoruz. Burada dikkatimizi çeken husus, createDocument() metoduna verilen ve "PDF" değerini taşıyan parametredir. Bu parametre yardımı ile DocumentFactory sınıfına hangi tipte bir döküman nesnesi oluşturması gerektiği bildirilmektedir. Burada kullanıcı sınıfın (Application) "PDF" parametresi aracılığıyla DokumentFactory tarafından yönetilen döküman oluşum sürecini yönlendiriyor olması dikkat çekmektedir. Application sınıfı somut altsınıfların yapısını ve nasıl oluşturulduklarını bilmek zorunda değildir. Application sınıfı String veri tipindeki bir parametre ile hangi bir tip döküman istediğini ifade etmektedir. DocumentFactory sınıfı bu parametreyi değerlendirerek, gerekli somut sınıfı bulmakta ve döküman nesnesini oluşturmaktadır.

Kod 12 de yer alan Application sınıfı çalıştırıldığında, ekran çıktısı aşağıdakı şekilde olacaktır:

```
Pdf
Word
```

Fabrika metodu tasarım şablonu ne zaman kullanılır?

- Kullanıcı sınıf hangi altsınıfların kullanılması gerektiğini bilmiyorsa; Örneğin Application sınıfı Pdf ya da Work döküman tiplerini tanımıyor.
- Kullanıcı sınıf altsınıflardan nasıl nesne üretilmesi gerektiğini bilmiyorsa.
- Kullanıcı sınıf ihtiyaç duyulan nesnelerin oluşturulmasını altsınıflara delege etmek istiyorsa.

İlişkili tasarım şablonları

Builder, prototype, abstract factory ve factory tasarım şablonları fabrika metodu tasarım şablonu ile benzerlik gösterirler ve yeni nesne oluşturmak için kullanılırlar.

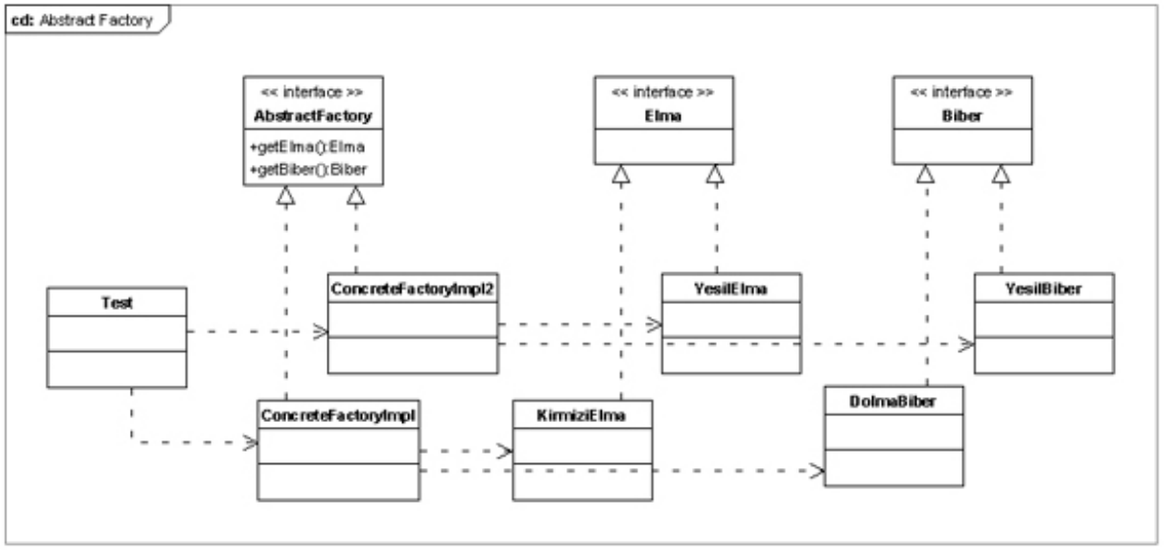
Soyut Fabrika (Abstract Factory)

Birbiriyle ilişkili nesne grupları (nesne ailesi) oluşturmak için soyut fabrika tasarım şablonu kullanılmaktadır. Soyut fabrika sınıfı bir interface sınıf olarak tanımlanır. İnterface bünyesinde aynı aileden nesneler oluşturmak için metotlar tanımlanır. Her nesne ailesi bir interface sınıfı metodu ile temsil edilir.

Kod 13 de yer alan örnekte de görüldüğü gibi Elma ve Biber isminde, çeşitli elma ve biber türlerini

temsil eden interface sınıfları tanımlıyoruz. Elma interface sınıfını implemente eden ve elma ailesinden olan KirmiziElma ve YesilElma isimlerinde sınıflar mevcut. KirmiziElma ve YesilElma aynı ailenin fertleri, çünkü Elma interface sınıfını implemente ediyorlar. Bu ailenin tüm fertleri Elma interface sınıfı ile temsil edilmektedir, yani bir KirmiziElma da elmadır bir YesilElma da.

Aynı şey Biber interface sınıfı için de geçerlidir. Biber ailesinin fertleri YesilBiber ve DolmaBiber sınıfları ile temsil edilmektedir



Resim 4

Önce AbstractFactory sınıfının kodunu inceliyelim. Bu sınıf baz alınarak, sistem içinde kullanılan fabrika sınıfları oluşturulacak.

```
// Kod 13

package com.pratikprogramci.designpatterns.bolum5.abstractfactory;

/**
 * Soyut fabrika sınıfı.
 *
 */
public interface AbstractFactory {

    /**
     * Elma tipinde nesne oluşturmak için kullanılır
     */
    public Elma getElma();

    /**
```

```
     * Biber tipinde bir nesne oluşturmak için kullanılır.
     */
    public Biber getBiber();

}
```

AbstractFactory interface sınıfı bünyesinde getElma() ve getBiber() isminde, Elma ve Biber isimli iki değişik nesne ailesinden nesne oluşturmak üzere iki metot yer almaktadır. Bu sınıf bir interface olduğu için new AbstractFactory() şeklinde kullanılması imkansızdır. Bu sebeple bu sınıfı implemente eden altsınıfların oluşturulması gerekmektedir. Bunu görmeden önce Elma ve Biber sınıflarını inceliyelim:

```
// Kod 14

package com.pratikprogramci.designpatterns.bolum5.abstractfactory;

/**
 * Çeşitli türdeki elmaları tanımlamak için kullanılan interface sınıfı
 *
 * @author Ozcan Acar
 *
 */
public interface Elma {

    /**
     * Hangi tür elma olduğunu gösterir.
     */
    public String getType();
}

package com.pratikprogramci.designpatterns.bolum5.abstractfactory;

/**
 * Çeşitli türdeki biberleri tanımlamak için kullanılan interface sınıfı
 *
 * @author Ozcan Acar
 *
 */
public interface Biber {

    /**
     * Hangi tür biber olduğunu gösterir.
     */
    public String getType();
```

```
}
```

Daha öncede belirttiğim gibi, Elma ve Biber isimlerinde, değişik elma ve biber türlerini ihtiva eden bir nesne ailesi tanımlamış oluyoruz. Elma ailesi içinde KirmiziElma ve YesilElma gibi altsınıflar bulunacaktır. Bu sınıflar Elma interface sınıfını implemente ettikleri andan itibaren, Elma ailesinin birer fertleri haline gelmektedirler.

Çesitli nesne ailelerinden nesneler üretebilmek için AbstractFactory interface sınıfını implemente eden fabrika sınıflarını görelim.

İlk fabrika sınıfımız elma ailesinden kırmızı elma, biber ailesinden dolma biber nesneleri oluşturma yeteneğine sahip olacak. Bu sınıf ConcreteFactoryImpl ismini taşıyor ve AbstractFactory interface sınıfını implemente ediyor.

```
// Kod 15

package com.pratikprogramci.designpatterns.bolum5.abstractfactory;


/**
 * AbstractFactory sınıfını implemete eden gerçek factory sınıfı.
 *
 */
public class ConcreteFactoryImpl implements AbstractFactory {


    /**
     * Elma tipinde nesne oluşturmak için kullanılır
     */
    public Elma getElma() {
        return new KirmiziElma();
    }

    /**
     * Biber tipinde bir nesne oluşturmak için kullanılır.
     */
    public Biber getBiber() {
        return new DolmaBiber();
    }
}
```

getElma() metodu içinde KirmiziElma isminde, elma nesne ailesine ait bir nesne oluşturulmaktadır. Aynı durum getBiber() metodu için de geçerlidir. Burada DolmaBiber isminde, biber nesne ailesinden bir nesne oluşturulmaktadır.

```
// Kod 16

package com.pratikprogramci.designpatterns.bolum5.abstractfactory;


/**
 * Kirmizi elmayı temsil eden sınıf.
 *
 */
public class KirmiziElma implements Elma {

    public String getType() {
        return "Kirmizi Elma";
    }
}

package com.pratikprogramci.designpatterns.bolum5.abstractfactory;

/**
 * Dolma biberi temsil eden sınıf
 *
 */
public class DolmaBiber implements Biber {

    public String getType() {
        return "Dolmalik Biber";
    }
}
```

Kod 15 de yer alan ConcreteFactoryImpl fabrika sınıfının görevi, KirmiziElma ve DolmaBiber nesneleri oluşturmaktır. Peki başka bir tür elma ya da bibere ihtiyaç duyulduğu zaman ne yapılması gerekiyor? Örneğin KirmiziElma yerine YesilElma, DolmaBiber yerine YesilBiber kullanmak istiyoruz. Bu durumda AbstractFactory sınıfını implemente eden ikinci bir factory sınıfına ihtiyacımız olacak:

```
// Kod 17

package com.pratikprogramci.designpatterns.bolum5.abstractfactory;

/**
 * AbstractFactory sınıfını implemete eden gerçek factory sınıfı.
 *
 */
public class ConcreteFactoryImpl2 implements AbstractFactory {
```

```
    /**
     * Elma tipinde nesne oluşturmak için kullanılır
     */
    public Elma getElma() {
        return new YesilElma();
    }

    /**
     * Biber tipinde bir nesne oluşturmak için kullanılır.
     */
    public Biber getBiber() {
        return new YesilBiber();
    }
}
```

İki değişik fabrika sınıfının nasıl kullanıldığını test programında görelim:

```
// Kod 18

package com.pratikprogramci.designpatterns.bolum5.abstractfactory;

/**
 * Abstract Factory test sınıfı
 *
 */
public class Test {

    public static void main(final String args[]) {

        /*
         * AbstractFactory sınıfını implemente eden ConcreteFactoryImpl sınıfı
         * kullanılıyor...
         */

        AbstractFactory factory = new ConcreteFactoryImpl();
        Elma elma = factory.getElma();
        Biber biber = factory.getBiber();
        System.out.println(elma.getType());
        System.out.println(biber.getType());

        /*
         * AbstractFactory sınıfını implemente eden ConcreteFactoryImpl2 sınıfı
         * kullanılıyor...
         */
        factory = new ConcreteFactoryImpl2();
```

```
        elma = factory.getElma();
        biber = factory.getBiber();
        System.out.println(elma.getType());
        System.out.println(biber.getType());
    }
}
```

Ekran çıktısı aşağıdaki şekilde olacaktır:

```
Kirmizi Elma
Dolmalik Biber
Yesil Elma
Yesil Biber
```

Soyut fabrika tasarım şablonu ne zaman kullanılır?

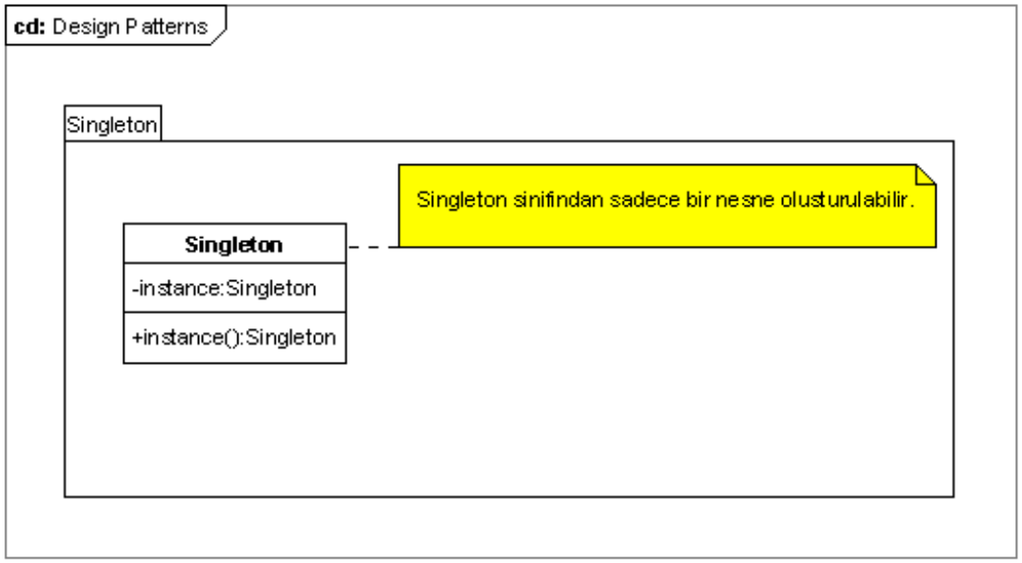
- Kullanıcı sınıf hangi altsınıfların kullanılması gerektiğini bilmiyorsa.
- Kullanıcı sınıf altsınıflardan nasıl nesne üretilmesi gerektiğini bilmiyorsa.
- Kullanıcı sınıf ihtiyaç duyulan nesnelerin oluşturulmasını altsınıflara delege etmek istiyorsa.
- Kullanıcı sınıf birden fazla nesne ailesi ile beraber çalışmak zorunda ise.

İlişkili tasarım şablonları

Soyut fabrika yerine prototip tasarım şablonu kullanılabilir. Bunun yanı sıra soyut fabrika bünyesinde prototip yardımıyla gerçek nesneler de oluşturmak mümkündür. Soyut fabrika sınıflarını fabrika metodu olarak implemente etmek münkündür. Builder tasarım şablonunda nesne adım adım oluşturulup, en son işlem olarak geri verilirken, soyut fabrikada nesne oluşturulur ve akabinde geri verilir.

Tekillik (Singleton Pattern)

Bazı şartlar altında bir sınıftan sadece bir nesnenin oluşturulması ve oluşturulan bu nesnenin tüm sistemde kullanılması gerekebilir. Örneğin veri tabanı için bir bağlantı havuzu nesnesi (connection pool) sadece bir defa oluşturulmalı ve kullanılmalıdır. Bu durumda tekillik tasarım şablonu kullanılarak, bir sınıftan sadece bir nesnenin oluşturulması sağlanabilir.



Resim 5

Tekil yapıya sahip bir sınıfı inceliyelim:

```
// Kod 19

package com.pratikprogramci.designpatterns.bolum5.singleton;

/**
 * Singleton tasarım şablonu implementasyonu
 *
 */
public class Singleton {

    /*
     * Singleton sınıfından olusturulabilecek tek nesne static sınıf değişkeni
     * olarak tanımlanıyor.
     */
    private static volatile Singleton instance = null;

    /*
     * Double check locking yapabilmek için kullanılan nesne.
     */
    private static Object lock = new Object();

    /**
     * Başka sınıfların new Singleton() şeklinde nesne oluşturmaları private
     * ile engellenmiş olmakta.
     *
     */
```

```
    private Singleton() {
        System.out.println("singletion init()");
    }

    /**
     * Singleton sınıfından oluşturulabilen tek nesneye ulaşmak için instance()
     * metodu kullanilmaktadir..
     */
    public static Singleton instance() {
        if (instance == null) {
            // Double checked locking
            synchronized (lock) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }

    /**
     * Singleton sınıfında bulunan bir metot.
     */
    public void printThis() {
        System.out.println(this);
    }
}
```

Bir singleton sınıfın taşıması gereken bazı özellikler vardır. Bunlar:

- Sınıf konstruktörlerinin private olması gerekiyor. Konstrüktörleri private olan bir sınıftan, başka bir sınıf new operatörü ile nesne oluşturamaz.
- Singleton sınıfından sadece bir tane nesne oluşturulması gerektiği için oluşturulması gereken nesneyi sınıfın static değiskeni olarak tanımlamamız gerekiyor. Kod 19 daki örnekte private static Singleton instance = null; şeklinde bu tanımlamayı yapıyoruz.
- Singleton sınıfında instance() isminde static bir metodun olması ve bu metodun static olarak tanımlanmış nesneyi geri vermesi gerekiyor. instance() metodu içinde sınıfın tek nesnesi olacak değişken oluşturulur.
- Sınıf bünyesinde bulunan instance() static metodu büyük önem taşımaktadır.

Singleton sınıflarda genelde aşağıdaki gibi bir instance() metodu görülebilir:

```
public static Singleton instance()  {
    if(instance == null){
```

```
        instance = new Singleton();
    }
    return instance;
}
```

Java multi threaded bir sistem olduğu için yukarıdaki instance() metodunda birden fazla new Singleton() ile Singleton nesnesi oluşturulabilir. Gözümüzde çalışan iki thread canlandırarak, bu sorunun nasıl oluştuğunu inceliyelim: Sistemde T1 ve T2 isimlerinde iki thread mecvut. T1 ilk olarak intance() metoduna girdi ve if(instance == null) satırını geçtikten sonra Java thread scheduler tarafından bloke edildi. Bu noktada T1 instance = new Singleton() yapamadan devre dışı kalmış oldu. Akabinde T2 çalışmaya başladı ve if(instance == null) satırını geçti ve zamanı yeterli olduğu için instance = new Singleton() yaptı. Zamanı dolduğu için T2 thread scheduler tarafından bloke edildi. Kontrol tekrar T1 threadine geçti. T1, T2 nin daha önce new Singleton() yaptığından habersiz olduğu için instance = new Singleton() yaparak tekrar instance değişkenini oluşturdu. Bu durumda aynı nesne arka arkaya iki sefer oluşturulmuş oldu. Bu durum sistem tarafından kullanılan kıymetli kaynakların oluşturulmasında sorun yaratabilir. Singleton sınıfı ve barındırdığı nesne sadece ve sadece bir defa oluşturulmalı. Yukarıda yer alan sorunu ortadan kaldırmak için double checked locking mekanizması kullanılabilir. Bizim örneğimizde yer alan instance() metodu double checked locking kullanmaktadır:

```
public static Singleton instance() {
        if (instance == null) {
            // Double checked locking
            synchronized (lock) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
```

Tekrar T1 ve T2 threadlerimizi kullanarak, nasıl double checked locking yönteminin iki sefer new Singleton() yapılmasını önlediğini görelim:

T1 instance() metoduna girer ve if(instance == null) satırını değerlendirir. Burada T1 thread scheduler tarafından bloke edilebilir. Biz edilmediğini ve T1 in devam ettiğini düşünelim. T1 synchronized bloğuna girdiği andan itibaren, bu bloğa T2 nin girmesi imkansızdır. Şimdi T1 in syncronozed bloğuna girdikten sonra bloke edildigini düşünelim. Bu durumda T2 sadece synchronized metodunun önüne kadar gelip, T1 bu bloktan çıkana kadar beklemek zorundadır, çünkü T1 lock nesnesinin synchronized lock mekanızmasını elinde tutmaktadır. T1 bunu geri

vermedigi sürece, yani synchronized bloğundan çıkmadığı sürece, diğer threadler bu bloğa girmezler. T1 kontrülü tekrar eline aldıktan sonra, emin olmak için bir daha if(instance == null) satırı ile nesnenin oluşturulup, oluşturulmadığını kontrol eder. Eğer T1 synchronized metoduna girmeden bloke olmuşsa, T2 bloğa girmiş ve new Singleton() yapmış olabilir. Tekrar kontrol edildiği için bu mekanizmanın ismi double checked locking dir. Metoda ilk girişte if(instance == null) ile kontrol edilir ve kontrol synchronized bloğunda tekrarlanır. Bu şekilde singleton nesnesinin birden fazla oluşturulması edilmesi önlenmiş olur. Bunun yanı sıra instance degişkeninin volatile olarak tanımlanmış olması gerekmektedir. Günümüzde kullanılan mikro işlemciler birden fazla çekirdek sahibi olduğundan, threadler tarafından değişikliğe uğrayan değişken değerlerinin çekirdekler arası senkronize edilmesi gerekir. Volatile olan değişkenler üzerinde yapılan değişiklikler, bu değişkenlerin sahip olduğu değerlerin tüm çekirdekler tarafından yeniden yüklenmelerini sağlar. Bu konudaki yazıma buradan ulaşabilirsiniz.

Şimdi bir test sınıfı ile Singleton nesnesi kontrol edilebiliriz.

```
// Kod 20

package com.pratikprogramci.designpatterns.bolum5.singleton;

/**
 * Singleton tasarım şablonu test sınıfı
 *
 */
public class Test {

    public static void main(final String[] args) {
        for (int i = 0; i < 10; i++) {
            Singleton.instance().printThis();
        }
    }
}
```

Ekran çıktısı:

```
singletion init()
com.pratikprogramci.designpatterns.bolum5.singleton.Singleton@75da931b
com.pratikprogramci.designpatterns.bolum5.singleton.Singleton@75da931b
com.pratikprogramci.designpatterns.bolum5.singleton.Singleton@75da931b
com.pratikprogramci.designpatterns.bolum5.singleton.Singleton@75da931b
com.pratikprogramci.designpatterns.bolum5.singleton.Singleton@75da931b
com.pratikprogramci.designpatterns.bolum5.singleton.Singleton@75da931b
com.pratikprogramci.designpatterns.bolum5.singleton.Singleton@75da931b
com.pratikprogramci.designpatterns.bolum5.singleton.Singleton@75da931b
```

```
com.pratikprogramci.designpatterns.bolum5.singleton.Singleton@75da931b
com.pratikprogramci.designpatterns.bolum5.singleton.Singleton@75da931b
```

Ekran çıktısında görüldüğü gibi Singleton sınıfından sadece bir nesne oluşturulmaktadır. Bunu singleton init() çıktısı ile görüyoruz. Daha sonraki satırlarda aynı nesne kullanılarak, printThis() metodunun çıktısı ekrana geliyor.

Günümüzde singleton tasarım şablonu anti pattern olarak görülmektedir. Bunun başlıca sebebi singleton nesnelerin zor test edilebilir olmasıdır. Bunun başlıca sebebi sınıf konstruktörünün private olmasıdır.

Tekillik tasarım şablonu ne zaman kullanılır?

Sistem bünyesinde bir sınıftan sadece bir nesne oluşturulması bu nesnenin kullanılması gerektiği durumlarda singleton tasarım şablonu kullanılır.

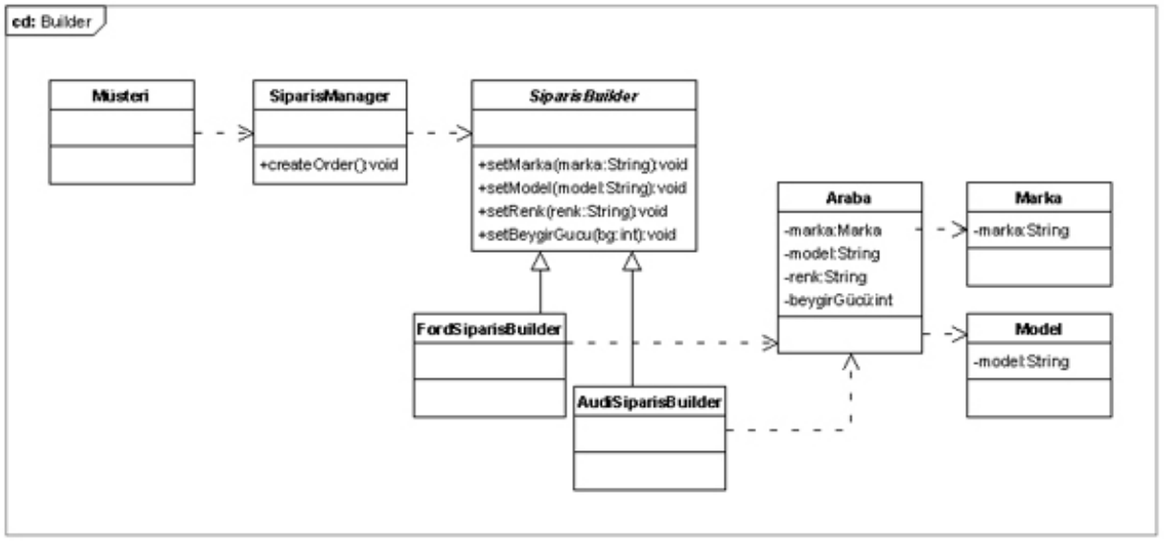
İlişkili tasarım şablonları

Abstract factory, builder ve prototype tasarım şablonlarında olabileceği gibi birçok tasarım şablonu singleton tasarım şablonu kullanılarak implemente edilebilir. Bu nesne oluşturma işleminin tekil ve merkezi bir yerden yapılmasını mümkün kılar.

Yapıcı (Builder)

Daha önce soyut fabrika (abstract factory) tasarım şablonu ile değişik nesne ailelerinden nasıl nesneler üretildiğini incelemiştik. Yapıcı (builder) tasarım şablonu da soyut fabrika tasarım şablonunda olduğu gibi istenilen bir tipte nesne oluşturmak için kullanılmaktadır. İki tasarım şablonu arasındaki fark, yapıcı tasarım şablonunun karmaşık yapıdaki bir nesneyi değişik parçaları bir araya getirerek oluşturmasında yatmaktadır. Birden fazla adım içeren nesne üretim sürecinde, değişik parçalar birleştirilir ve istenilen tipte nesne oluşturulur.

Şimdi bir örnek üzerinde bu tasarım şablonunu yakından inceliyelim.



Resim 6

Araba sınıfından oluşturulacak bir nesne, bünyesinde Marka ve Model sınıflarından birer nesne ihtiva etmektedir yani bir araba nesnesi bereberinde bir marka ve bir model nesnesi getirmektedir. Bu örnekte yapıcı tasarım şablonunu kullanma amacımız, Araba, Marka ve Model sınıflarından oluşan bir nesnenin değişik parametreler kullanarak konfigüre edilmesidir. Sipariş edilen her arabanın markası, modeli, rengi ve beygirgücü değişik olabileceğinden, sipariş için kullanılan Araba nesnesinin bu parametrelere dayalı olarak oluşturulması gerekmektedir. Yapıcı tasarım şablonu bu işlemi sistemin diğer bölümlerinden bağımsız bir şekilde gerçekleştirmek için kullanılmaktadır.

Araba sınıfı aşağıdakı yapıya sahiptir:

```
// Kod 21

package com.pratikprogramci.designpatterns.bolum5.builder;


public class Araba {

    private Marka marka = null;

    private Model model = null;

    private String renk = null;

    private int beygirGucu = 0;

    public Araba() {
```

```
    }

    public Model getModel() {
        return model;
    }

    public void setModel(final Model model) {
        this.model = model;
    }

    public String getRenk() {
        return renk;
    }

    public void setRenk(final String renk) {
        this.renk = renk;
    }

    public int getBeygirGucu() {
        return beygirGucu;
    }

    public void setBeygirGucu(final int beygirGucu) {
        this.beygirGucu = beygirGucu;
    }

    public Marka getMarka() {
        return marka;
    }

    public void setMarka(final Marka marka) {
        this.marka = marka;
    }

}
```

Her sipariş verilen arabanın bir markası, modeli, rengi ve hangi beygir gücüne sahip olduğu bu sınıftan oluşturulan nesne içinde tutulmaktadır.

```
// Kod 22

package com.pratikprogramci.designpatterns.bolum5.builder;

public class Marka {
    private String marka;
```

```
    public Marka(final String marka) {
        setMarka(marka);
    }

    public String getMarka() {
        return marka;
    }

    public void setMarka(final String marka) {
        this.marka = marka;
    }

    @Override
    public String toString() {
        return marka;
    }
}

package com.pratikprogramci.designpatterns.bolum5.builder;

public class Model {
    private String model;

    public Model(final String model) {
        setModel(model);
    }

    public String getModel() {
        return model;
    }

    public void setModel(final String model) {
        this.model = model;
    }

    @Override
    public String toString() {
        return model;
    }
}
```

Değişik marka ve modellerde arabaların sipariş edilebilmesi için soyut olan SiparisBuilder sınıfını tanımlıyoruz.

```
// Kod 23
```

```
package com.pratikprogramci.designpatterns.bolum5.builder;

/**
 * Sipariş sürecinde konfigürasyon için kullanılacak metotlar bu sınıf içinde
 * yer almaktadır. Belirli bir marka arabanın sipariş için altsiniflarin
 * olusturulmsi gerekmektedir.
 *
 */
public abstract class SiparisBuilder {

    private Araba araba = null;

    public SiparisBuilder() {
    }

    public Araba getAraba() {
        if (araba == null) {
            araba = new Araba();
        }
        return araba;
    }

    public abstract void setMarka(String marka);

    public abstract void setModel(String model);

    public abstract void setRenk(String renk);

    public abstract void setBeygirGucu(int bg);

}
```

Ford ve Audi marka arabaların siparişinde kullanılmak üzere SiparisBuilder sınıfının altsınıflarını tanımlıyoruz.

```
// Kod 24

package com.pratikprogramci.designpatterns.bolum5.builder;

/**
 * Audi marka bir arabanın konfigürasyon ve siparisinde kullanilam sınıf.
 *
 */
public class AudiSiparisBuilder extends SiparisBuilder {

    public AudiSiparisBuilder() {
```

```
    }

    @Override
    public void setBeygirGucu(final int bg) {
        getAraba().setBeygirGucu(bg);
    }

    @Override
    public void setMarka(final String marka) {
        getAraba().setMarka(new Marka(marka));
    }

    @Override
    public void setModel(final String model) {
        getAraba().setModel(new Model(model));
    }

    @Override
    public void setRenk(final String renk) {
        getAraba().setRenk(renk);
    }
}

package com.pratikprogramci.designpatterns.bolum5.builder;

/**
 * Ford marka bir arabanın konfigürasyon ve siparisinde kullanılan sınıf.
 */
public class FordSiparisBuilder extends SiparisBuilder {

    public FordSiparisBuilder() {
    }

    @Override
    public void setBeygirGucu(final int bg) {
        getAraba().setBeygirGucu(bg);
    }

    @Override
    public void setMarka(final String marka) {
        getAraba().setMarka(new Marka(marka));
    }

    @Override
    public void setModel(final String model) {
        getAraba().setModel(new Model(model));
    }
```

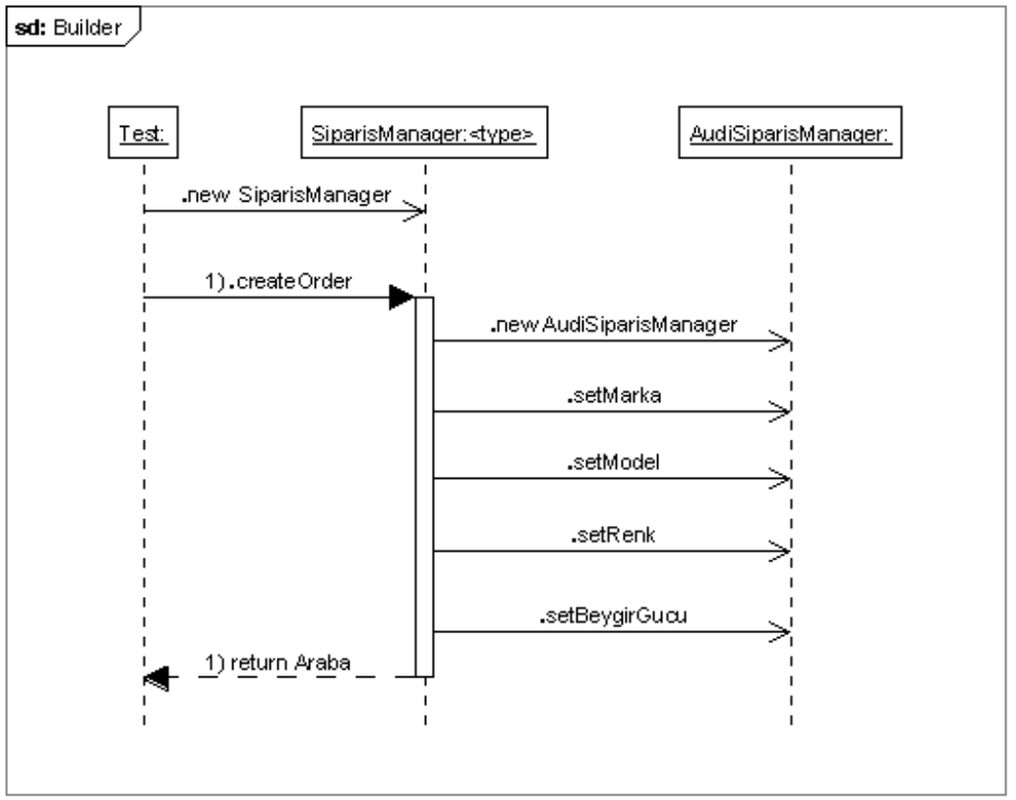
```
    @Override
    public void setRenk(final String renk) {
        getAraba().setRenk(renk);
    }
}
```

Geriye sipariş işlemini gerçekleştirmek için kullanılan SiparisManager sınıfı kalıyor:

```
// Kod 25

package com.pratikprogramci.designpatterns.bolum5.builder;

/**
 * Sipariş işlemini gerçekleştirmek için kullanılan sınıf.
 *
 */
public class SiparisManager {
    private SiparisBuilder builder;

    /**
     * Müşterinin istediği marka, model, renk ve beygir gücüne sahip bir
     * arabanın siparişi için kullanılan metot.
     *
     * @param marka
     *            Sipariş edilen arabanın markası
     * @param model
     *            Sipariş edilen arabanın modeli
     * @param renk
     *            Sipariş edilen arabanın rengi
     * @param beygirgucu
     *            Sipariş edilen arabanın beygir gücü
     *
     * @return Araba Konfigüre edilmiş araba nesnesi
     */
    public Araba createOrder(final String marka, final String model,
            final String renk, final int beygirgucu) {
        if (marka.equals("Ford")) {
            builder = new FordSiparisBuilder();
        } else if (marka.equals("Audi")) {
            builder = new AudiSiparisBuilder();
        }

        builder.setMarka(marka);
        builder.setModel(model);
        builder.setRenk(renk);
```

```
        builder.setBeygirGucu(beygirgucu);
        return builder.getAraba();
    }

    /**
     * Sipariş edilen arabanın özelliklerini ekranda görüntüler.
     */
    public void printOrder() {
        System.out.println("Marka: " + builder.getAraba().getMarka());
        System.out.println("Model: " + builder.getAraba().getModel());
        System.out.println("Renk: " + builder.getAraba().getRenk());
        System.out.println("Beygirgücğ: " +
                    builder.getAraba().getBeygirGucu());
    }
}
```

Kod 25 de yer alan createOrder() metodu ile istenilen tipte bir arabanın siparişi gerçekleştirilmektedir. Bu metot parametre olarak arabanın marka, model, rengi ve beygir gücünü almaktadır. Hangi marka arabanın sipariş edildiğini marka isimli değişkene bakarak bulabilmekteyiz. Eğer marka Ford ise FordSiparisBuilder, Audi ise AudiSiparisBuilder sınıflarından bir nesne oluşturulmaktadır. Akabinde setMarka(), setModel(), setRenk() ve setBeygirGucu() metodları kullanılarak, Araba sınıfından oluşturulan nesnenin konfigürasyonu gerçekleştirilmektedir.



Resim 7

Dizgi (sequence) diyagramında da görüldüğü gibi, Araba nesnesini oluşturmak için dört adım atılması gerekmektedir. Bunlar SiparisBuilder sınıfının soyut olarak tanımlanmış olan set metotlarıdır.

Kod 25.1 de yer alan test sınıfı SiparisManager sınıfının nasıl kullanıldıgını göstermektedir.

```
// Kod 25.1

package com.pratikprogramci.designpatterns.bolum5.builder;

public class Test {
    public static void main(final String[] args) {
        SiparisManager manager = new SiparisManager();
        manager.createOrder("Ford", "Focus", "Beyaz", 70);
        manager.printOrder();

        manager = new SiparisManager();
        manager.createOrder("Audi", "A5", "Siyah", 270);
        manager.printOrder();
    }
}
```

Ekran çıktısı şu şekilde olacaktır:

```
Marka: Ford
Model: Focus
Renk: Beyaz
Beygirgücğ: 70
Marka: Audi
Model: A5
Renk: Siyah
Beygirgücğ: 270
```

Daha öncede belirttiğim gibi karmaşık yapıya sahip olan Araba nesnesini değişik parçaları (Marka, Model, renk, beygirgücü) bir araya getirerek oluşturduk. Nesnenin nasıl oluşturulup, parçalarının biraraya getirileceğini SiparisManager sınıfının createOrder() metodunda tanımlıyoruz. Builder tasarım şablonunu kullanabilmek için createOrder() metodunda olduğu gibi, karmaşık nesnenin oluşumunda kullanılan metotların tanımlanması ve belirli bir sıraya göre kullanılması gerekmektedir.

Yapıcı tasarım şablonu bunun yanı sıra akıcı programlama arayüzleri (fluent API - Application Programming Interface) oluşturmak için kullanılan tasarım şablonudur. Böyle akıcı bir arayüz örneği aşağıdaki kodda yer almaktadır.

```
FluentBuilder.startBuilding().withMarka(new Marka("Fiat"))
            .withModel(new Model("Dublo")).withBeygirGucu(200)
            .withRenk("beyaz").build();
```

FluentBuilder sınıfı bir araba nesnesi oluşturmakla sorumlu olan sınıftır. Arabayı oluşturan özellikleri toplamak için with ile başlayan metotlar tanımlanmıştır. Her bir with() metodu ile arabanın bir özelliği edinilir ve build() metodunda araba nesnesi bu özellikler kullanılarak, oluşturulur. FluentBuilder sınıfı kod 26 da yer almaktadır.

```
// Kod 26

package com.pratikprogramci.designpatterns.bolum5.builder;

public class FluentBuilder {

    private Marka marka;
    private Model model;
    private int beygirGucu;
    private String renk;
```

```
    public static FluentBuilder startBuilding() {
        return new FluentBuilder();
    }

    public FluentBuilder withMarka(final Marka marka) {
        this.marka = marka;
        return this;
    }

    public FluentBuilder withModel(final Model model) {
        this.model = model;
        return this;
    }

    public FluentBuilder withBeygirGucu(final int bg) {
        beygirGucu = bg;
        return this;
    }

    public FluentBuilder withRenk(final String renk) {
        this.renk = renk;
        return this;
    }

    public Araba build() {
        final Araba araba = new Araba();
        araba.setMarka(marka);
        araba.setModel(model);
        araba.setRenk(renk);
        araba.setBeygirGucu(beygirGucu);
        return araba;
    }

    public static void main(final String[] args) {
        FluentBuilder.startBuilding().withMarka(new Marka("Fiat"))
                .withModel(new Model("Dublo")).withBeygirGucu(200)
                .withRenk("beyaz").build();
    }
}
```

Bu yazımda ve bu yazımda yapıcı tasarım şablonunun kullanımını örnekler üzerinde gösterdim.

Yapıcı tasarım şablonu ne zaman kullanılır?

- Değişik parametreler kullanılarak karmaşık yapıda bir nesnenin oluşturulması gerektiğinde.
- Karmaşık yapıya sahip nesnenin oluşturulma sürecinin, sistemin diğer bölümlerinden bağımsız

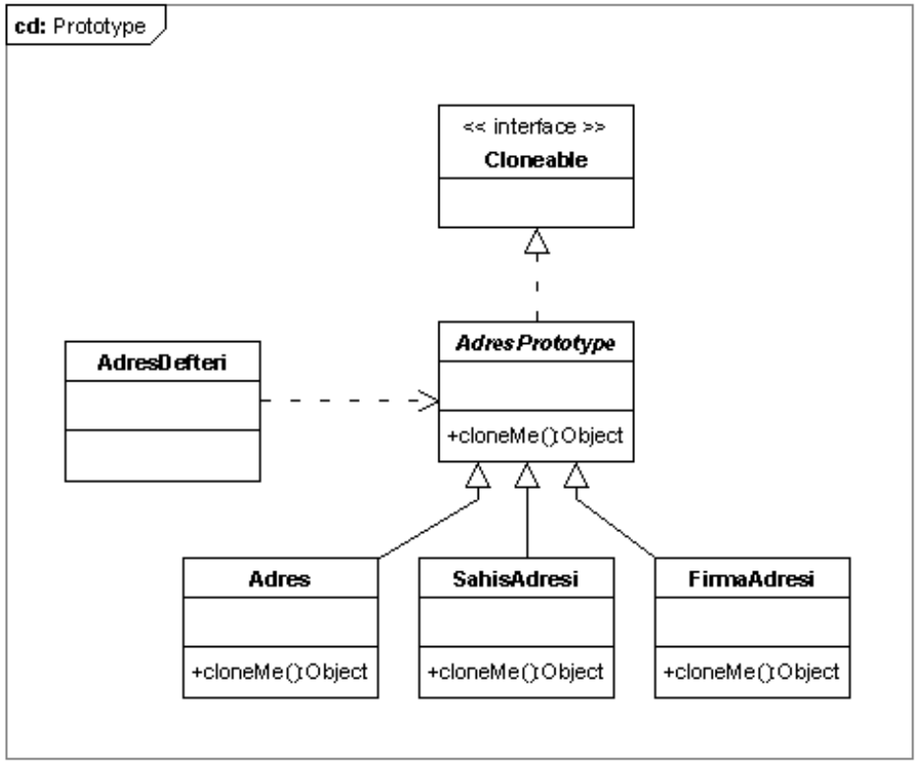
bir şekilde yapılması gerektiği durumlarda.

İlişkili tasarım şablonları

Composite Tasarım şablonunda oluşturulan composite nesne builder tasarım şablonunda oluşturulan karmaşık nesne gibidir. Ayrıca abstract factory kullanılarak karmaşık nesneler oluşturulabilir.

Prototip (Prototype)

Sistem içinde kullanılan bazı nesnelerin oluşturulmaları, değişik kaynakları kullandıklarından dolayı zaman alıcı olabilir. Bu gibi nesneler new operatörü ile yeniden oluşturulmak yerine, prototip tasarım şablonu kullanılarak, mevcut bir nesneden klonlanabilirler. Bu şekilde oluşan nesne bir prototiptir ve set metodları kullanılarak istenilen özelliklere göre yapılandırılabilir.



Resim 8

Şimdi bu tasarım şablonunun nasıl kullanıldığını bir örnek üzerinde inceleyelim. Bir adres defterinde kullanılmak üzere AdresPrototype isminde bir soyut sınıf tanımlıyoruz. Bu sınıf Cloneable interface sınıfını implemente ettiği için altsınıflardan oluşturulacak nesnelerin clone() metodu kullanılarak birebir kopyaları oluşturulabilmektedir.

```
// Kod 27

package com.pratikprogramci.designpatterns.bolum5.prototype;

/**
 * Klonlanabilir nesneler oluşturmak için kullanılan sınıf.
 *
 */
public abstract class AdresPrototype implements Cloneable {

    private String sokak;
    private String no;
    private String semt;
    private String şehir;

    /**
     * Altsınıflardan oluşturulan nesneleri klonlamak için kullanılan metot.
     *
     * @throws CloneNotSupportedException
     */
    @Override
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public String getSokak() {
        return sokak;
    }

    public void setSokak(final String sokak) {
        this.sokak = sokak;
    }

    public String getNo() {
        return no;
    }

    public void setNo(final String no) {
        this.no = no;
    }

    public String getSemt() {
        return semt;
    }

    public void setSemt(final String semt) {
```

```
        this.semt = semt;
    }

    public String getSehir() {
        return şehir;
    }

    public void setSehir(final String şehir) {
        this.şehir = şehir;
    }

    public void printAdres() {
        System.out.println("Sokak: " + getSokak());
        System.out.println("No: " + getNo());
        System.out.println("Semt: " + getSemt());
        System.out.println("Sehir: " + getSehir());
        System.out.println("hashCode: " + hashCode());
    }
}
```

Klonlama işlemini gerçekleştirmek için clone() isminde bir metot tanımlıyoruz.

Adres defteri programı içinde Adres, FirmaAdres ve SahisAdres sınıflarından oluşturulan nesneler kullanılmaktadır. Bu sınıfların ortak özelliği, AdresPrototype sınıfından tüm değişken ve metotları miras olarak almalarıdır. Bu sınıfları kod 28 görmekteyiz.

```
// Kod 28

package com.pratikprogramci.designpatterns.bolum5.prototype;

/**
 * Genel bir adres verisini temsil eden sınıf.
 *
 */
public class Adres extends AdresPrototype {

    public Adres(final String sokak, final String no, final String semt,
            final String şehir) {
        setSokak(sokak);
        setNo(no);
        setSemt(semt);
        setSehir(şehir);
    }
}

// FirmaAdres sınıfı
```

```
package com.pratikprogramci.designpatterns.bolum5.prototype;

/**
 * Bir firma adresini temsil eden sınıf.
 *
 */
public class FirmaAdres extends AdresPrototype {
    private String firma;

    public FirmaAdres(final String sokak, final String no,
            final String semt,
            final String şehir, final String firma) {
        setSokak(sokak);
        setNo(no);
        setSemt(semt);
        setSehir(şehir);
        setFirma(firma);
    }

    public String getFirma() {
        return firma;
    }

    public void setFirma(final String firma) {
        this.firma = firma;
    }
}

// SahisAdres sınıfı

package com.pratikprogramci.designpatterns.bolum5.prototype;

/**
 * Bir şahıs adresini temsil eden sınıf.
 *
 */
public class SahisAdres extends AdresPrototype {
    private String isim;
    private String soyad;

    public SahisAdres(final String sokak, final String no,
            final String semt,
            final String şehir, final String isim,
            final String soyad) {
        setSokak(sokak);
        setNo(no);
```

```
        setSemt(semt);
        setSehir(şehir);
        setIsim(isim);
        setSoyad(soyad);
    }

    public String getIsim() {
        return isim;
    }

    public void setIsim(final String isim) {
        this.isim = isim;
    }

    public String getSoyad() {
        return soyad;
    }

    public void setSoyad(final String soyad) {
        this.soyad = soyad;
    }
}
```

Klonlama işlemini gerçekleştirmek için AdresDefteri.main() metodunu kullanıyoruz.

```
// Kod 29

package com.pratikprogramci.designpatterns.bolum5.prototype;

/**
 * Adres defteri programı.
 *
 */
public class AdresDefteri {
    public static void main(final String[] args)
            throws CloneNotSupportedException {
        final Adres adres =
                new Adres("Ataturk cad.", "10", "Atasehir",
                "İstanbul");
        adres.printAdres();

        /*
         * adres nesnesini clone() metodu ile kopyalıyoruz.
         */
        final Adres adres2 = (Adres) adres.clone();
```

```
        /*
         * set metotlarını kullanarak, klonlanan nesneyi yeniden
         * yapilandiriyoruz.
         */
        adres2.setSokak("Ulus cad.");
        adres2.setNo("120");
        adres2.setSemt("Kadiköy");
        adres2.setSemt("İstanbul");
        adres2.printAdres();
    }
}
```

Main() içinde Adres tipinde bir nesne oluştuyoruz. Nesneyi oluşturmak için sokak, no, semt ve şehir parametrelerinin kullanılması gerekmektedir. Daha sonra adres nesnesini klonlayarak adres2 isminde ikinci bir nesne oluşturuyoruz. Clone() operasyonundan sonra adres2 bir prototip nesnedir ve adres nesnesinin tüm özelliklerine sahiptir. Bu noktada new operatörünü kullanmadan, mevcut bir nesneyi taban alarak ikinci bir nesne oluşturduk. Daha sonra set metotlarını kullanarak, adres2 nesnesini istediğimiz özelliklerde yapılandırıyoruz.

Ekran çıktısı şu şekilde olacaktır:

```
Sokak: Atatürk cad.
No: 10
Semt: Atasehir
Sehir: İstanbul
hashCode: 20392474
Sokak: Ulus cad.
No: 120
Semt: İstanbul
Sehir: İstanbul
hashCode: 11352996
```

İncelediğimiz örnek prototip tasarım şablonu için belki uygun olmayabilir. adres.clone() yerine new Adres(..) operasyonuyla da yeni bir adres nesnesi oluşturabilirdik. Adres oluşturulma bedeli yüksek bir nesne değil. Karmaşık sistemlerde clone() ile klonlanmaya değer nitelikte nesneler mutlaka olacaktır.

Prototip tasarım şablonu, sistemde mevcut altsınıfların adedini azaltmak için de kullanılabilir. Örneğin AdresDefteri programına dernek ve kurum adreslerinde eklemek istiyoruz. Bu durumda DernekAdres ve KurumAdres isminde iki yeni altsınıf oluşturmak zorundayız. Bu iki altsınıfı oluşturmak yerine, AdresPrototype sınıfına bir dernek ve kurum için gerekli değişkenleri ekliyerek, DernekAdres ve KurumAdres sınıfları yerine Adres sınıfını kullanabiliriz.

Prototip tasarım şablonu ne zaman kullanılır?

- Bir sistem bünyesinde büyük ve oluşturulmaları zaman alıcı nesneler bulunduğu taktirde.
- Sistem bünyesinde kullanılan sınıf hirarşilerini küçültmek ve kullanılan sınıf adedinin azaltılması gerektiğinde.

İlişkili tasarım şablonları

Prototip yerine soyut fabrika kullanılabilir. Bunun yanı sıra soyut fabrika bünyesinde prototip yardımıyla gerçek nesneler de oluşturmak mümkündür. Fabrika metodunda kalıtım kullanılırken, prototip bünyesinde yeni nesne oluşturma işlemi delegasyon yöntemi ile yapılmaktadır.

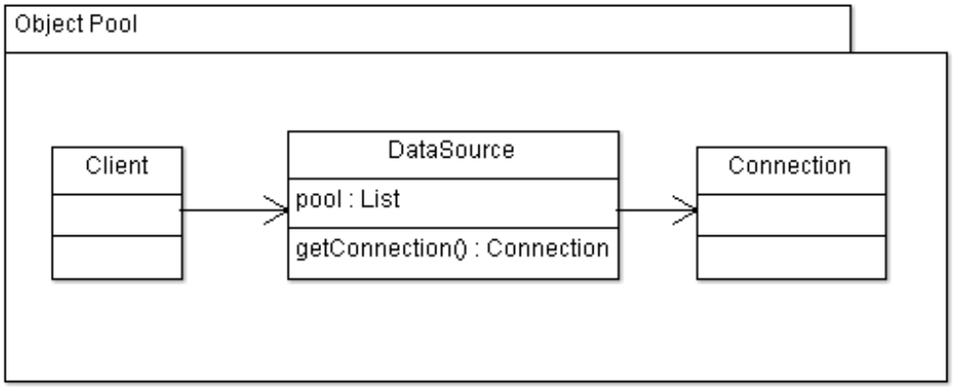
Nesne Havuzu (Object Pool)

Bir nesnenin oluşturulabilmesi için belli bir zamana ihtiyaç duyulmaktadır. Java dilinde new operatörü yardımı ile nesneler oluşturulmaktadır. Birçok dilde new operatörüne rastlamak mümkündür.

Şimdi kullanmak istediğimiz nesneleri new ile oluşturmak yerine, bir nesne havuzundan temin ettiğimizi düşünelim. Böyle bir yapının bize sağladığı avantajlar neler olurdu? Öncelikle nesne edinme sürecimiz hızlanırdı, çünkü biz nesneyi edinmek istediğimiz zaman new işlemenin gerçekleşmesi zorunluluğu ortadan kalmış olurdu. Karmaşık yapıya sahip nesnelerin oluşturulma süreçlerinin uzun olabileceğini düşündüğümüzde, bir nesneyi doğrudan edinmenin ne kadar avantajlı olabileceği görülmektedir.

Nesnelerin bulunduğu havuzu bir bellek olarak düşünebiliriz. Uygulama çalışmaya başladığı andan itibaren, örneğin fabrika ya da fabrika metodu yöntemiyle gerekli nesneleri oluşturarak, havuz doldurulabilir. Bu tür belleklere cache ismi verilmektedir. Uygulama bünyesinde nesnelere hızlı erişimi, kaynakların ölçülü kullanılmasını ve nesnelerin tekrar kullanılmasını (reuse) mümkün kılmak için bu tür bellekler kullanılmaktadır.

Şimdi nesne havuzu tasarım şablonunun nasıl implemente edilebileceğini bir örnek üzerinde inceleyelim. Vereceğim örnek uygulama sunucularında veri tabanı bağlantılarını sınırlı tutmak için kullanılan yapıyı ihtiva etmektedir ve connection pool olarak isimlendirilmektedir. Veri tabanı bağlantısını modellemek için Connection isminde bir sınıf oluşturdum. Uygulama bünyesinde bir veri tabanı bağlantısı edinebilmek için DataSource sınıfını kullanmamız gerekmektedir. Uygulama (örnekte Client sınıfı) bir veri tabanı bağlantısı alabilmek için DataSource sınıfını kullanmaktadır.



Resim 9

Resim 9 görüldüğü gibi havuzda tutmak istediğimiz nesneler Connection sınıfından oluşturduğumuz nesneler olacaktır.

```
// Kod 30

package com.pratikprogramci.designpatterns.bolum5.objectpool;

public class Connection {

}
```

DataSource sınıfı Connection nesnelerinden oluşan havuzu bünyesinde tutan sınıftır. Nesne havuzunu modellemek için ArrayList sınıfını kullandım.

```
// Kod 31

package com.pratikprogramci.designpatterns.bolum5.objectpool;

import java.util.ArrayList;
import java.util.List;

public class DataSource {

    static DataSource instance = new DataSource();
    List<Connection> pool = new ArrayList<Connection>();

    private DataSource() {
        init();
    }

    private void init() {
```

```
        for (int i = 0; i < 3; i++) {
            pool.add(new Connection());
        }
    }

    static Connection getConnection() {
        if (instance.pool.size() == 0) {
            throw new NoConnectionsInPool();
        }

        Connection con = instance.pool.get(0);
        instance.pool.remove(0);

        System.out.println(con);

        return con;
    }

    static void release(Connection con) {
        if (con != null) {
            instance.pool.add(con);
        }
    }

}
```

Kod 31 de görüldüğü gibi DataSource sınıfı singleton yapısındadır. Sınıf konstrüktörü bünyesinde init() metodu aracılığı ile Connection nesne havuzu oluşturulmaktadır. Bu işlem sadece bir sefer gerçekleşmektedir, çünkü instance değişkeni bir singleton nesnesidir. instance = new DataSource() ile bu nesne hayat bulmakta ve akabinde init() metodu koşturulmaktadır. Sınıf konstrüktörü private olduğundan başka hiçbir sınıf DataSource sınıfından bir nesne ve dolaylı olarak bir Connection nesne havuzu oluşturamamaktadır.

Nesne havuzu üç adet Connection nesnesi ihtiva etmektedir. Havuzdan bir nesne almak için getConnection(), havuza bir nesneyi geri vermek için release() metotları kullanılmaktadır. Şimdi bir uygulamanın havuzdan bir nesneyi nasıl alabileceğine göz atalım.

```
// Kod 32

package com.pratikprogramci.designpatterns.bolum5.objectpool;

public class Client {
```

```
    public static void main(String[] args) {

        Connection con1 = DataSource.getConnection();
        Connection con2 = DataSource.getConnection();
        Connection con3 = DataSource.getConnection();
    }
}
```

Main() metodunu koşturduğumuzda, alacağımız ekran çıktısı şu şekilde olacaktır:

```
com.pratikprogramci.designpatterns.bolum5.objectpool.Connection@5058431c
com.pratikprogramci.designpatterns.bolum5.objectpool.Connection@529e0c79
com.pratikprogramci.designpatterns.bolum5.objectpool.Connection@645064f
```

Bu durumda havuzda bulunan tüm nesneleri havuzdan çekip, almış olduk. Eğer dördüncü Connection nesnesini havuzdan almak istersek, o zaman bir hata mesajı almamız gerekir. Deneyelim:

```
// Kod 33

public static void main(String[] args) {

        Connection con1 = DataSource.getConnection();
        Connection con2 = DataSource.getConnection();
        Connection con3 = DataSource.getConnection();
        Connection con4 = DataSource.getConnection();
    }
```

Kod 33 de yer alan main() metodunu koşturduğumuzda, karşılacağımız ekran görüntüsü şu şekilde olacaktır:

```
com.pratikprogramci.designpatterns.bolum5.objectpool.Connection@5058431c
Exception in thread "main"
com.pratikprogramci.designpatterns.bolum5.objectpool.Connection@529e0c79
com.pratikprogramci.designpatterns.bolum5.objectpool.Connection@645064f
com.pratikprogramci.designpatterns.bolum5.objectpool.NoConnectionsInPool
    at
com.pratikprogramci.designpatterns.bolum5.objectpool.DataSource.
        getConnection(DataSource.java:24)
    at com.pratikprogramci.designpatterns.bolum5.objectpool.Client
    .main(Client.java:11)
```

Görüldüğü gibi NoConnectionsInPool hatasını aldık, çünkü havuz içinde kullanabileceğimiz bir nesne kalmadı. Release() metodunu kullanarak daha önce edindiğimiz bir nesneyi havuza şu

şekilde geri verebiliriz:

```
// Kod 34

public static void main(String[] args) {

    Connection con1 = DataSource.getConnection();
    Connection con2 = DataSource.getConnection();
    Connection con3 = DataSource.getConnection();

    DataSource.release(con1);
    Connection con4 = DataSource.getConnection();
}
```

Ekran çıktısı şu şekilde olacaktır:

```
com.pratikprogramci.designpatterns.bolum5.objectpool.Connection@5058431c
com.pratikprogramci.designpatterns.bolum5.objectpool.Connection@529e0c79
com.pratikprogramci.designpatterns.bolum5.objectpool.Connection@645064f
com.pratikprogramci.designpatterns.bolum5.objectpool.Connection@5058431c
```

Nesne havuzu tasarım şablonu ne zaman kullanılır?

- Sistem kaynaklarının kontrollü bir şekilde kullanılması gerektiğinde.
- Kullanılan nesne adedine sınırlama getirilmek istendiğinde.
- Nesne edinme süreci hızlandırılmak istendiğinde.

İlişkili tasarım şablonları

Havuzu gerekli nesnelerle doldurmak için prototip tasarım şablonu kullanılabilir. Factory method tasarım şablonu ile nesnelerin oluşturulma işlemi havuz ve havuzdan nesne edinme kodundan ayrıştırılabilir. Havuzun kendisi singleton tasarım şablonu ile implemente edilir.

6. Bölüm

Yapısal Tasarım Şablonları - Structural Patterns

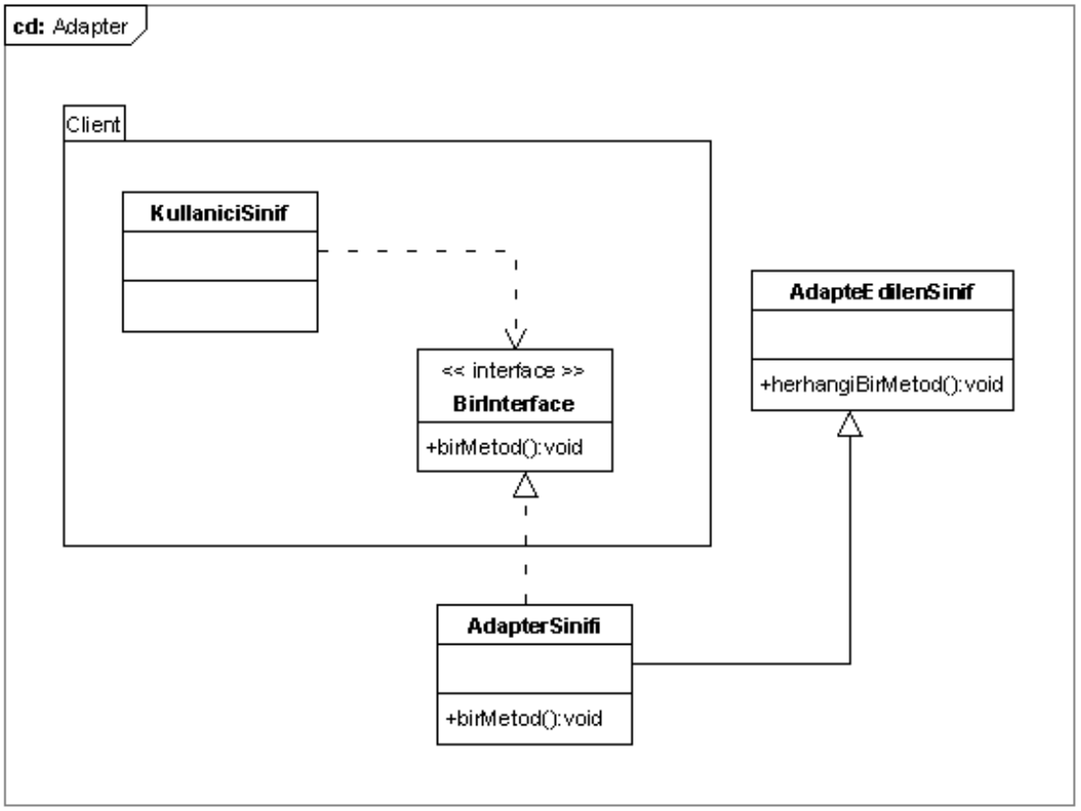
Adaptör (Adapter)

Adaptör tasarım şablonu yardımı ile sistemde mevcut bulunan bir sınıfın sunduğu arayüz (metotları) başka bir sınıf tarafından kullanılabilir şekilde değiştirilir (adapte edilebilir). Bu adaptör yardımı ile birbiriyle beraber çalışamayacak durumda olan sınıflar, birlikte çalışabilir hale getirilirler.

Adaptör tasarım şablonu kendi içinde sınıf ve nesne adaptörü olarak ikiye ayrılmaktadır. Önce sınıf adaptörünü inceliyelim.

Sınıf Adaptörü (Class Adapter)

Kurumsal projelerde sistemin parçaları değişik ekipler tarafından tasarlanır ve implemente edilir. Bu her zaman geçerli olmasada, çoğu zaman daha önceden hazırlanmış sistem komponentlerini kullanmak zorunda kalabiliriz. Aşağıda yer alan UML diyagramında da görüldüğü gibi KullaniciSinif isminde bir program hazırlanmıştır ve bu program BirInterface sınıfı içinde yer alan mototları kullanmaktadır. Bu sınıfın sisteme entegrasyonunu sağlayabilmek için ya BirInterface sınıfını implemente eden yeni bir sınıf oluşturmamız ya da mevcut bir sınıfı kullanarak entegrasyonu sağlamamız gerekmektedir. Ufak bir araştırma yaptıktan sonra, KullaniciSinif ismindeki programın işlevini yerine getirebilmesi için başka bir ekip tarafından hazırlanmış olan AdapteEdilenSinif isminde bir sınıf olduğunu ögreniyoruz.



Resim 1

Adaptör tasarım şablonunu kullanarak, KullaniciSinif sınıfını AdapteEdilenSinif sınıf metotlarını kullanabilecek hale getirebiliriz. KullaniciSinif aşağıdakı yapıya sahiptir:

```
// Kod 1

package com.pratikprogramci.designpatterns.bolum6.adapter.clazz;

/**
 * BirInterface sınıfına bağımlı olan KullaniciSinif.
 *
 */
public class KullaniciSinif {

    public static void main(final String[] args) {

        /*
         * AdapterSinifi, BirInterface interface sınıfını implemente
         * ettiği için BirInterface olarak sistem içinde kullanılabilir.
         */
        BirInterface birInterface = new AdapterSinifi();
        birInterface.birMetod();
```

```
    }
}
```

KullaniciSinif işlevini yerine getirebilmek için mutlaka bir BirInterface nesnesine ihtiyaç duymaktadır.

```
// Kod 2

package com.pratikprogramci.designpatterns.bolum6.adapter.clazz;

public interface BirInterface {

    public void birMetod();

}
```

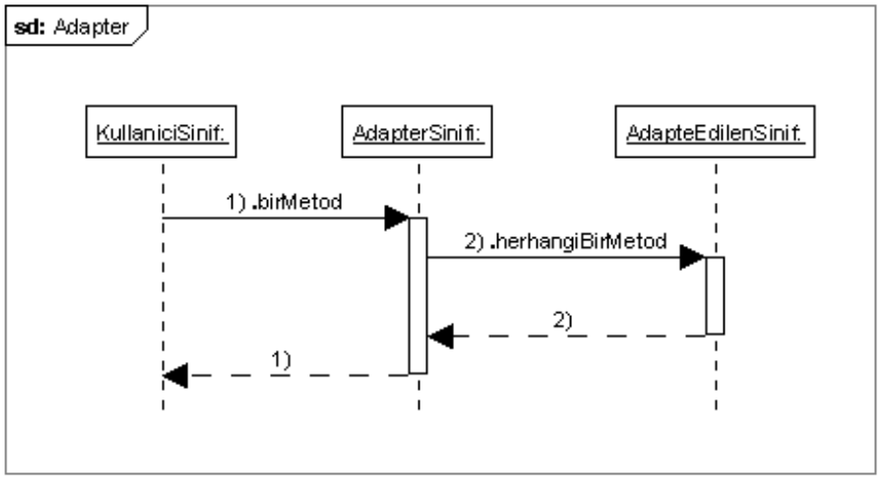
BirInterface sınıfında, KullaniciSinif sınıfı tarafından ihtiyaç duyulan birMetod() metodu yer almaktadır. Daha öncede belirttiğim gibi, ya BirInterface sınıfını implemente eden bir altsınıf oluşturacağız ya da adaptör şablonu ile sistemdeki mevcut bir sınıfı adapte etmeye çalışacağız. Yeni bir sınıf oluşturmak yerine, adaptasyon metodunu seçiyoruz. Bunun için AdapterSinifi isminde, adaptasyonu sağlamak üzere yeni bir sınıf oluşturuyoruz:

```
// Kod 3

package com.pratikprogramci.designpatterns.bolum6.adapter.clazz;

/**
 * Adaptasyonu gerçekleştirmek için kullanılan sınıf. BirInterface sınıfını
 * implemente ederek, KullaniciSinifi için kullanılabilir bir sınıf haline
 * gelir. Extends komutu ile AdapteEdilenSinif sınıfında yer alan özellikleri
 * miras olarak alır.
 *
 */
public class AdapterSinifi extends AdapteEdilenSinif implements BirInterface {

    /**
     * BirInterface interface sınıfında tanımlanmış olan birMetod() metodu
     * AdapterSinifi tarafından implemente edilir.
     */
    @Override
    public void birMetod() {
        /**
         * AdapterEdilenSinif içinde yer alan herhangiBirMetod() metodunu miras
         * olarak alır. KullaniciSinifi tarafından birMetod() metodu
```

```
         * kullanıldığında, bu metot herhangiBirMetod() metoduna delege edilir
         * ve işlem AdapteEdilenSinif sınıfına devredilmiş (adapte) edilmiş
         * olur.
         */
        herhangiBirMetod();
    }
}
```

AdapterSinifi BirInterface interface sınıfını implemente etmektedir. Bu andan itibaren KullaniciSinif sınıfı AdapterSinifi sınıfından oluşturulan bir nesneyi kullanabilir. Ancak bu yeterli değil! AdapteEdilenSinifi sınıfının KullaniciSinif tarafından dolaylı olarak kullanılabilmesi için AdapterSinifi ile AdapteEdilenSinif arasında extends komutu ile baglantı kurmamız gerekiyor.

```
// Kod 4

package com.pratikprogramci.designpatterns.bolum6.adapter.clazz;

/**
 * Adapte edilip  kullanılan bir sınıf.
 *
 */
public class AdapteEdilenSinif {

    public void herhangiBirMetod() {
        System.out.println("Herhangi bir metot");
    }
}
```



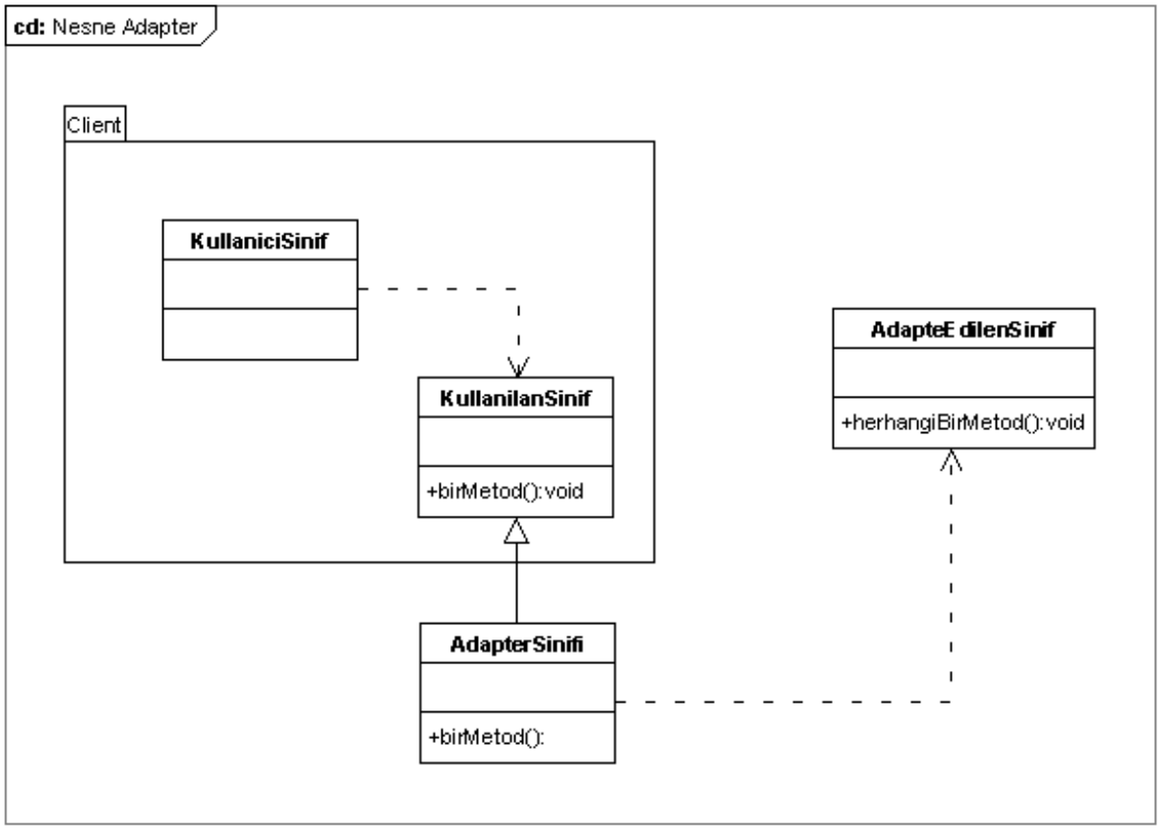
Resim 2

Dizgi diyagramında görüldüğü gibi AdapterSinifi.birMetod() kullanıldığı zaman işlem otomatik

olarak AdapteEdilenSinif.herhangiBirMetod() metoduna delege edilmektedir ve bu şekilde adaptasyon gerçekleşmektedir.

Nesne Adaptörü (Object Adapter)

Bazı durumlarda sınıf adaptör tasarım şablonu kullanılamayabilir. Bu gibi durumlarda nesne adaptör tasarım şablonu alternatif oluşturmaktadır.



Resim 3

Adaptasyonu gerçekleştirmek için KullaniciSinifi sınıfının kullandığı KullanilanSinif AdapterSinifi ile genişletilir. Bu durumda AdapterSinifi ile KullanilanSinif aynı veri tipinden olduklarından, iki sınıfın nesneleri de KullaniciSinif (kod 5) tarafından kullanılabilir hale gelmektedir. İkinci adımda KullaniciSinif (kod 5) tarafından kullanılan, KullanilanSinif sınıfının birMetod() metodu AdapterSinifi bünyesinde tekrar implemente edilir. AdapteEdilenSinif (kod 8) ile AdapterSinifi (kod 7) arasındaki ilişkiyi oluşturmak için AdapteEdilenSinifi sınıfından bir nesne sınıf değişkeni olarak AdapterSinif sınıfına eklenir (adapteEdilenSinif isimli değişken). Kullanılan sınıfların kodları aşağıda yer almaktadır.

```
// Kod 5
```

```
package com.pratikprogramci.designpatterns.bolum6.adapter.object;

public class KullaniciSinif {

    public static void main(final String[] args) {
        final KullanilanSinif kullanilanSinif = new AdapterSinifi();
        kullanilanSinif.birMetod();
    }
}
```

KullaniciSinif.main() metodunda görüldüğü gibi new operatörü ile KullanilanSinif sınıfından bir nesne oluşturmak yerine, AdapterSinifi sınıfından bir nesne oluşturuyoruz. AdapterSinifi (kod 7) sınıfı KullanilanSinif (kod 6) sınıfının altsınıfıdır. Burada hangi adapter sınıfının kullanıldığını saklamak amacıyla fabrika tasarım şablonunu da kullanabiliriz. KullaniciSinif sınıfı örneğin şu şekilde bir KullanilanSinif nesnesi edinebillir:

```
KullanilanSinif kullanilanSinif = KullanilanSinifFactory.instance();
kullanilanSinif.birMetod();
```

KullanilanSinifFactory sınıfı bir fabrika olup, şu yapıda olabilir:

```
    public  KullanilanSinif instance() {
        return new AdapterSinifi();
    }
```

Şimdi bu nesne üzerinde birMetod() metodunu koşturabiliriz:

```
// Kod 6

package com.pratikprogramci.designpatterns.bolum6.adapter.object;

public class KullanilanSinif {

    /**
     * KullaniciSinif sınıfı tarafından kullanılmak istenen asıl metot bu.
     */
    public void birMetod() {
        System.out.println("bir metot");
    }
}


// Kod 7
```

```
package com.pratikprogramci.designpatterns.bolum6.adapter.object;

/**
 * Adaptasyonu gerçekleştirmek için kullanılan sınıf. KullanilanSinif sınıfını
 * genişleterek, birMetod() metodunu yeniden implemente eder.
 *
 */
public class AdapterSinifi extends KullanilanSinif {

    /**
     * Adaptasyonu gerçekleştirmek için AdapteEdilen sınıfından bir nesne
     * oluşturulur.
     */
    private AdapteEdilenSinif adapteEdilenSinif = new AdapteEdilenSinif();

    /**
     * Bu metoda gelen çağrılar, sınıf değişkeni olan adapteEdilenSinif
     * değişkenine gönderilir.
     */
    @Override
    public void birMetod() {
        getAdapteEdilenSinif().herhangiBirMetod();
    }

    public AdapteEdilenSinif getAdapteEdilenSinif() {
        return adapteEdilenSinif;
    }

    public void setAdapteEdilenSinif(final AdapteEdilenSinif adapteEdilenSinif) {
        this.adapteEdilenSinif = adapteEdilenSinif;
    }
}

// Kod 8

package com.pratikprogramci.designpatterns.bolum6.adapter.object;

public class AdapteEdilenSinif {

    public void herhangiBirMetod() {
        System.out.println("Herhangi bir metot");
    }
}
```

AdapterSinifi sınıfında tanımladığımız adapteEdilenSinif değişkeni, AdapteEdilenSinif sınıfına bir nevi köprü oluşturmaktadır. birMetod() kendisine gelen çağrıları bu nesne üzerinden

AdapteEdilenSinif sınıfına iletmektedir ve bu şekilde KullaniciSinif ile AdapteEdilenSinif arasındaki adaptasyon gerçekleşmektedir.

Adaptör tasarım şablonu ne zaman kullanılır?

- Sistemde mevcut sınıflar kullanmak istenildiğinde, lakin sınıfın sunmuş olduğu arayüz (metot ve değişkenler) istenilen cinsten değilse.
- Tekrar kullanılabilir ve sistemin diğer bölümlerinden bağımsız bir sınıf ya da komponent oluşturulmak istendiğinde.

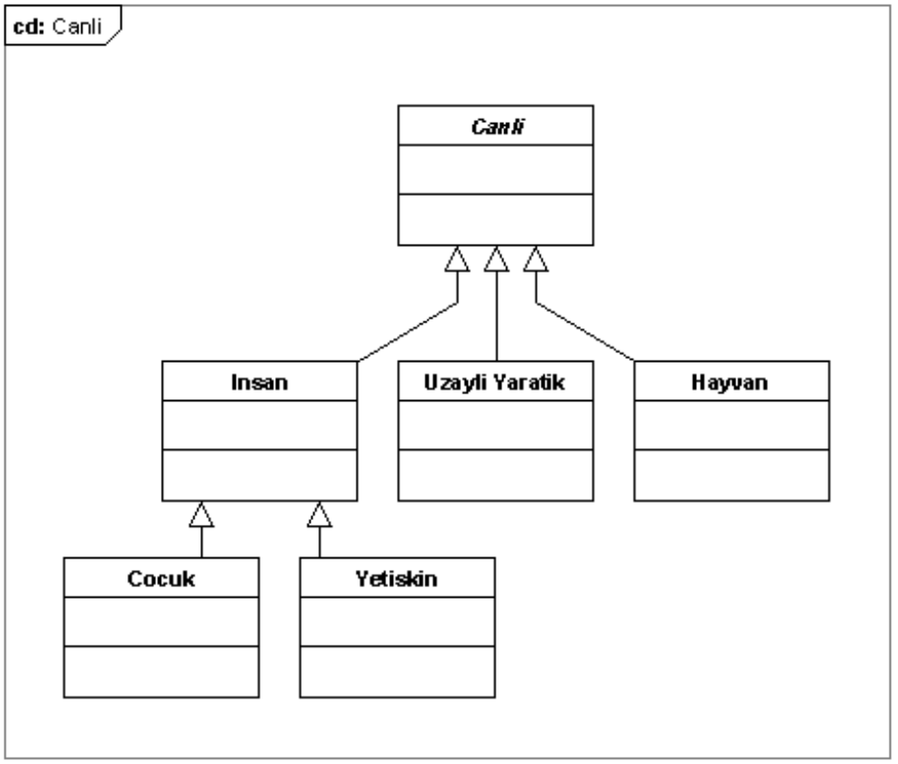
İlişkili tasarım şablonları

Bridge (köprü) tasarım şablonu yapı itibariyle nesne adaptör tasarım şablonuna benzemektedir, lakin iki tasarım şablonunun kullanım amaçları değişiktir. Bridge tasarım şablonu ile soyut sınıflar ve implementasyonları birbirlerinden ayrıştırılırken, adaptör tasarım şablonu bir sınıfın sunmuş olduğu arayüzü (metotlar) değiştirmek için kullanılmaktadır. Proxy tasarım şablonu ile başka bir nesne için temsilci oluşturulur. Proxy temsil ettiği nesnenin arayüzünü değiştirmez.

Köprü (Bridge)

Nesneye yönelik yazılım gerçek hayatta mevcut olan nesneleri modellemek suretiyle gerçekleşmektedir. Java dilinde nesneleri modellemek için sınıfları (class) kullanıyoruz. Her sınıf bu açıdan bakıldığı zaman soyut bir yapıdadır, çünkü gerçek hayattaki nesnenin modelini oluşturmaktadır.

Nesneleri modellerken sınıf hirarşileri oluştururuz. Örneğin Canlı isminde bir soyut sınıf tanımlayarak, tüm canlı yaratıkları temsil eden bir model oluşturabiliriz.



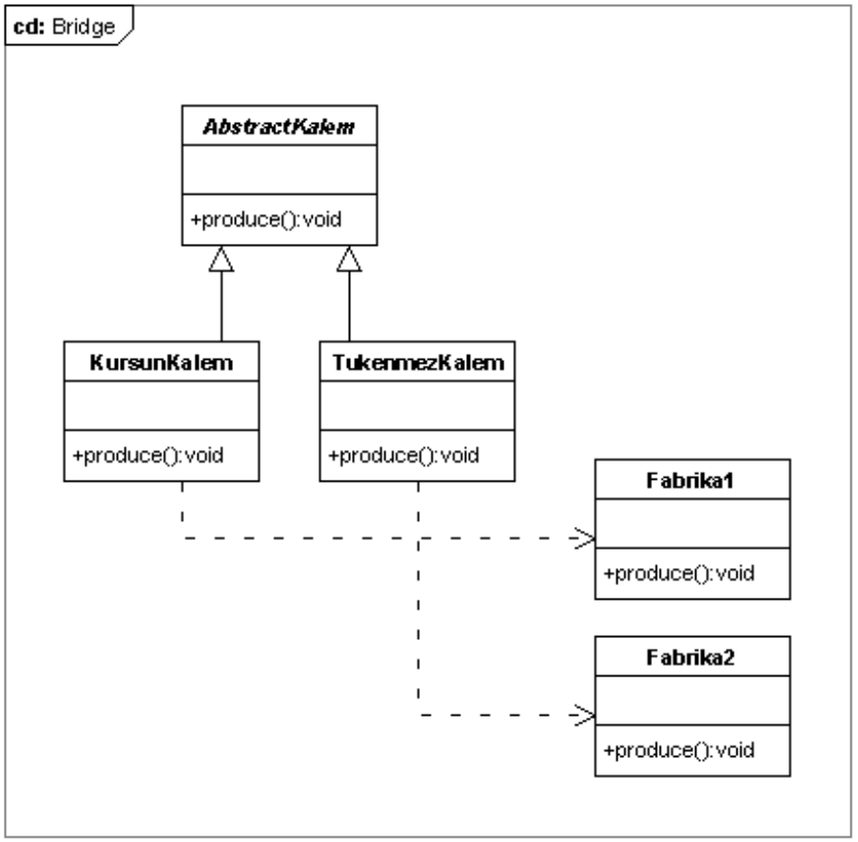
Resim 4

Modelimizi genişletmek için altsınıflar oluşturabiliriz ve bu sayede sınıflar arası hirarşik bir yapı oluşmaktadır. Her altsınıf Canlı sınıfında yer alan tüm değişkenlere sahip olmakla beraber, bu sınıfta tanımlanmış olan soyut metodları implemente etmek durumundadır.

Geniş çaplı yapıdaki sınıfları modellerken karmaşık ve bakımı zor olan sınıf hirarşileri oluşabilir ve bu durum modelleme işlemini çıkmaza sokabilir. Sınıf hirarşileri yerine her zaman nesneler arası birleşme (object associativity) yönetimi kullanılmalıdır. Bunu gerçekleştirmek için köprü tasarım şablonu kullanılabilir.

Köprü tasarım şablonu, modelleme esnasında oluşan soyut oluşumlar ve bunların implementasyonunu ayırmak için kullanılmaktadır. Bu yöntem sayesinde sınıf hirarşileri daha esnek bir hale getirilebilmektedir, çünkü üstsınıflar bünyelerinde barındırdıkarı soyut metodları bir interface sınıfına taşıyarak, altsınıfların istedikleri bir implementasyonu kullanmalarına izin vermektedirler. Köprü tasarım şablonunun nasıl uygulanabileceğini bir örnek üzerinde görelim.

Bir kırtasiye firmasının envanterini yönetmek için bir program yazılımı yapmamız istenmektedir. İlk etapta bize verilen bilgi, sistemde değişik tipte kalemlerin mevcut olduğu ve bu bu kalemlerin iki değişik fabrikada üretiliyor olmalarıdır. Resim 5 deki şekilde ilk modelimizi oluşturuyoruz.



Resim 5

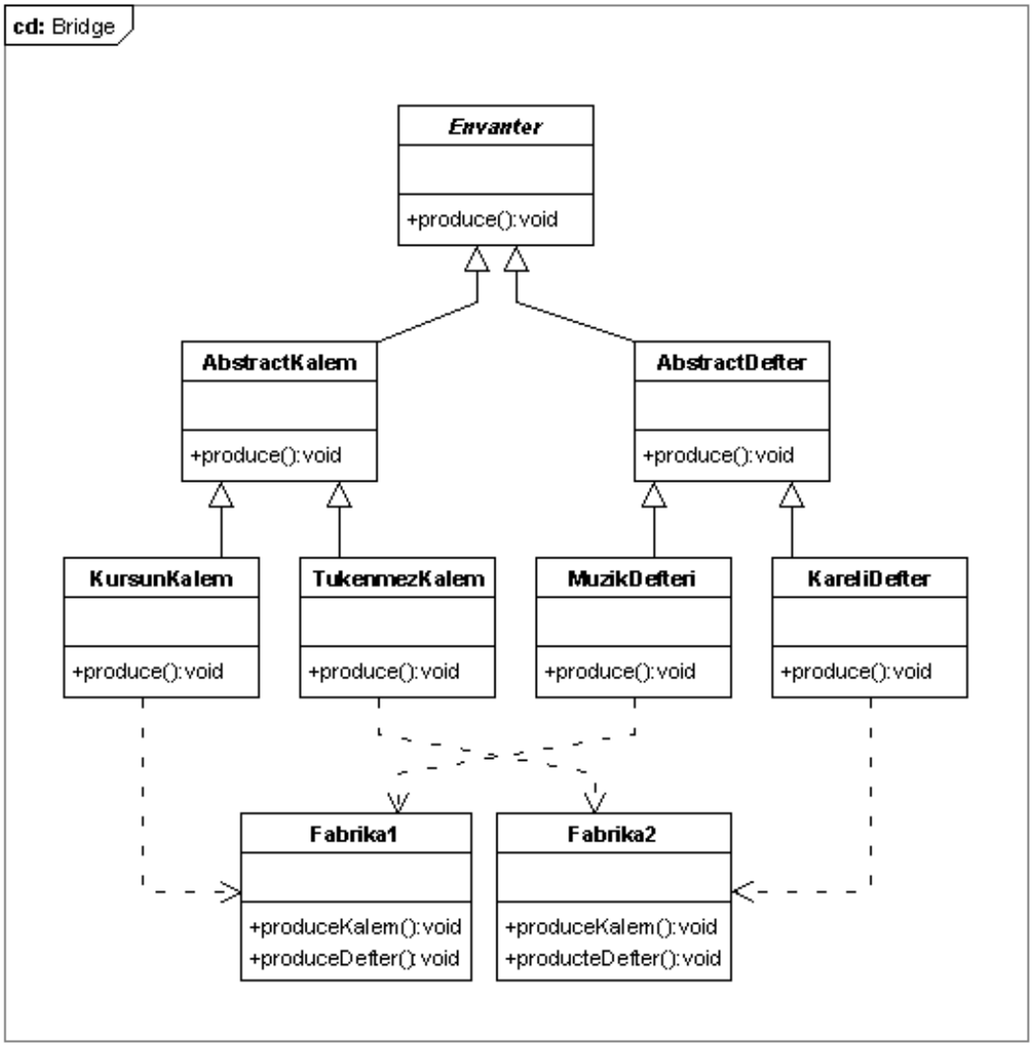
Genel anlamda bir kalemi modellemek için AbstractKalem isminde bir sınıf tanımlıyoruz. Satılan kurşun ve tükenmez kalemler AbstractKalem sınıfının altsınıflarını oluşturmaktadırlar. Kurşun ve tükenmez kalemler değişik fabrikalarda üretilmektedirler. Üretim yapan fabrikaları Fabrika1 ve Fabrika2 isminde iki sınıf ile tanımlıyoruz.

```
// Kod 9

package com.pratikprogramci.designpatterns.bolum6.bridge.ornek1;

public interface AbstractKalem {

    public void produce();

}

// Kod 10

package com.pratikprogramci.designpatterns.bolum6.bridge.ornek1;

public class KursunKalem implements AbstractKalem {
```

```
    private final Fabrika1 fabrika = new Fabrika1();

    @Override
    public void produce() {
        fabrika.produce();
    }
}

// Kod 11

package com.pratikprogramci.designpatterns.bolum6.bridge.ornek1;

public class Fabrika1 {
    public void produce() {
        System.out.println("Fabrika1: Kursun kalem imal edildi.");
    }
}
```

Kurşun kalem üretimi Fabrika1 tarafından gerçekleştirilmektedir. Bu nedenle KursunKalem sınıfında Fabrika1 tipinde bir değişken tanımlıyoruz ve produce() metodu kullanıldığında, bu çağrıyı fabrıkanın produce() metoduna delege ediyoruz.

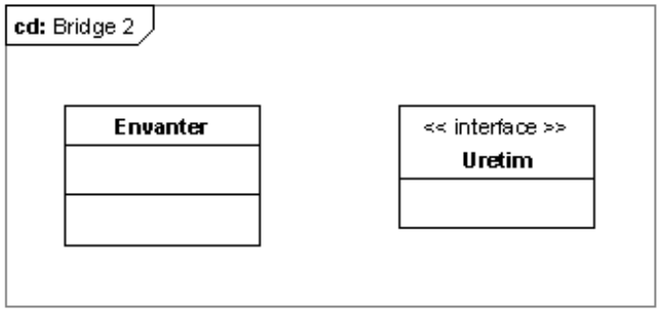
Kısa bir zaman sonra envanter içinde değişik tipte defterlerin de yer alacağı ve bu defterlerin iki değişik fabrikada üretileceği bilgisi bize ulaşıyor. Aşağıdaki şekilde modelimizi değiştiriyoruz.



Resim 6

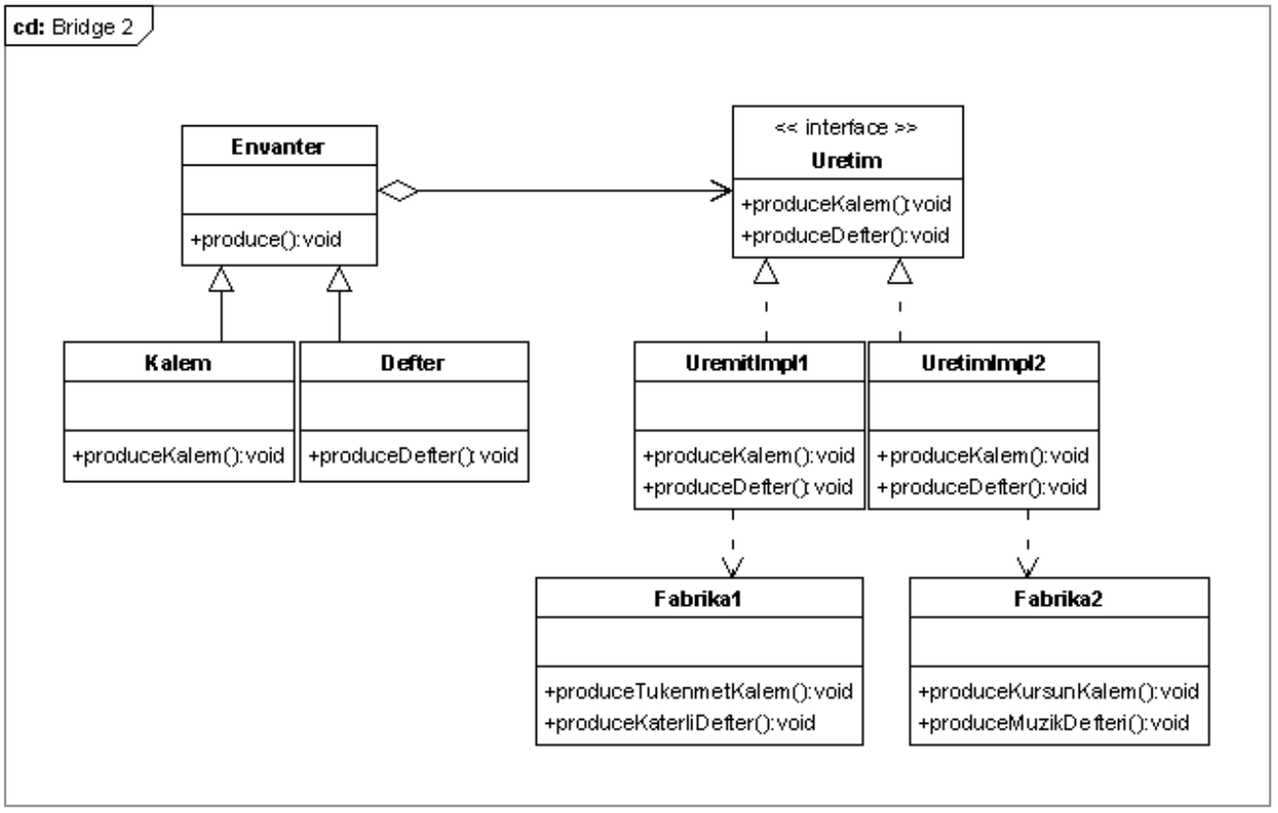
Görüldüğü gibi modelimiz, envanter büyüdükçe karmaşık bir yapıya dönüşmektedir. Envanter içine değişik tipte silgilerin de eklendiğini düşünürsek, modelimiz için kullandığımız sınıf sayısının giderek arttığını göreceğiz. Dört değişik tipte silgi eklemek için dört ayrı altsınıf oluşturmamız gerekmektedir. Bunun yanı sıra her altsınıfın bir fabrika ile bağlantısı oluşturulup, silgilerin üretiminin yapılması gerekmektedir. Görüldüğü gibi bir çıkmazın içine girmiş bulunuyoruz. Sınıf adedinin artmasının bir sebebi de kullandığımız envanter ürünlerinin, üretildikleri fabrikalara bağımlı olmalarıdır. Her envanter üretildiği fabrikayı bilmek zorundadır. Daha sağlıklı bir model oluşturabilmek için envanter ürünleri ile üretildikleri fabrikalar arasındaki bağı azaltmamız ya da ortadan kaldırmamız gerekmektedir.

Modelimizi köprü tasarım şablonunu kullanarak daha kullanışlı bir hale getirebiliriz. Bunun için ilk etapta modelimizde hangi değerlerin değişik türleri olduğunu tespit etmemiz gerekiyor.



Resim 7

Sistemimizde değişik tipte envanter bulunmaktadır (kalem, defter, silgi...). Bunun yanı sıra üretim için birden fazla fabrika mevcut. Tüm envanterleri Envanter, tüm üretim yapan fabrikaları da Uretim sınıfıyla temsil edecek şekilde modelizi değiştiriyoruz. Çeşitli kalem tiplerini temsil eden sınıfları kaldırarak, yerine sadece Envanter sınıfının altsınıfı olan Kalem sınıfını oluşturuyoruz. Aynı işlemi defter içinde gerçekleştiriyoruz.



Resim 8

Köprü tasarım şablonunu uyguladıktan sonra modelimiz daha derli toplu ve bakımı kolay bir hale geldi. Değişik fabrikalarda üretimi sağlamak için UretimImpl1 ve UretimImpl2 isimlerinde Uretim

interface sınıfının altsınıflarını oluşturuyoruz. Bu altsınıflar hangi fabrikalarla beraber çalıştıklarını bildikleri için o fabrikanın üretim metotlarını kullanabilirler.

```
// Kod 12

package com.pratikprogramci.designpatterns.bolum6.bridge.ornek2;

/**
 * Bridge tasarım şablonu bünyesinde gerekli soyut metotların tanımlandığı
 * interface sınıfı.
 *
 */
public interface Uretim {

    public void produceDefter();

    public void produceKalem();
}


// Kod 13

package com.pratikprogramci.designpatterns.bolum6.bridge.ornek2;

/**
 * Fabrika1 deki üretimi yönetmek için kullanılan sınıf.
 *
 */
public class UretimImpl1 implements Uretim {

    private final Fabrika1 fabrika = new Fabrika1();

    @Override
    public void produceDefter() {
        fabrika.produceKareliDefter();
    }

    @Override
    public void produceKalem() {
        fabrika.produceTukenmezKalem();
    }
}


// Kod 14
```

```
package com.pratikprogramci.designpatterns.bolum6.bridge.ornek2;

/**
 * Fabrika2 deki üretimi yönetmek için kullanılan sınıf.
 *
 */
public class UretimImpl2 implements Uretim {

    private final Fabrika2 fabrika = new Fabrika2();

    @Override
    public void produceDefter() {
        fabrika.produceMuzikDefteri();
    }

    @Override
    public void produceKalem() {
        fabrika.produceKursunKalem();
    }
}


// Kod 15

package com.pratikprogramci.designpatterns.bolum6.bridge.ornek2;

/**
 * Üretim yapan fabrika sınıfı
 *
 */
public class Fabrika1 {

    public void produceTukenmezKalem() {
        System.out.println("Fabrika1: Tükenmez kalem imal edildi.");
    }

    public void produceKareliDefter() {
        System.out.println("Fabrika1: Kareli defter imal edildi.");
    }
}


// Kod 16

package com.pratikprogramci.designpatterns.bolum6.bridge.ornek2;

/**
```

```
 * Üretim yapan fabrika sınıfı.
 *
 */
public class Fabrika2 {

    public void produceKursunKalem() {
        System.out.println("Fabrika2: Kursun kalem imal edildi.");
    }

    public void produceMuzikDefteri() {
        System.out.println("Fabrika2: Müzik defteri imal edildi.");
    }
}
```

Köprü tasarım şablonunun püf noktası Uretim interface sınıfında saklıdır. Envanter sınıfında bulunması gereken ve üretim için kullanılan soyut metodları Uretim interface sınıfına taşıdık. UretimImpl1 ve UretimImpl2 bu metodları implemente ederek, bu metotları sistemin başka bölümleri tarafından kullanılır hale getiriyorlar.

Köprü tasarım şablonunu kullanabilmek için soyut metodları ve bu metotların implementasyonlarını ayırmamız gerektiğini daha önce belirtmiştim. Bu ayırma işlemini gerçekleştirmek için gerekli tüm soyut metotlar Uretim interface sınıfına taşındı. Bu metotları ve implementasyonlarını kullanabilmek için Envanter sınıfında Uretim isminde bir değişken tanımlamamız gerekmektedir.

```
// Kod 17

package com.pratikprogramci.designpatterns.bolum6.bridge.ornek2;

/**
 * Değişik tipte envanterleri temsil eden sınıf.
 *
 */
public abstract class Envanter {
    /*
     * Üretim hangi fabrikada gerçekleşecek?
     */
    private Uretim uretim;

    public abstract void produce();

    public Envanter(final Uretim uretim) {
        setUretim(uretim);
    }
```

```
    public Uretim getUretim() {
        return uretim;
    }

    public void setUretim(final Uretim uretim) {
        this.uretim = uretim;
    }
}
```

Bu değişikliklerin ardından Envanter nesneleri ve bu nesnelerin üretildikleri fabrikalar arasındaki ilişkiyi ortadan kaldırmış ve bu ilişkiyi Envanter sınıfında bulunan uretim değişkenine taşımış olduk. Envanter sınıfında tanımladığımız uretim ismindeki değişken ile Envanter ve Uretim arasında köprü oluşturmuş oluyoruz. Bu noktadan itibaren Envanter sınıfı, uretim değişkeninin sahip olduğu değere göre, bu UretimImpl1 ya da UretimImpl2 sınıfından bir nesne olabilir, üretim için gerekli fabrikayı kullanacaktır. Envanter sınıfında, altsınıflar tarafından implemente edilmek üzere produce() isminde soyut bir metot tanımlıyoruz.

Yaptığımız değişiklikle Kalem ve Defter sınıflarını da daha sade hale getiriyoruz:

```
// Kod 18

package com.pratikprogramci.designpatterns.bolum6.bridge.ornek2;

/**
 * Bir kalemi temsil eden sınıf.
 *
 */
public class Kalem extends Envanter {

    public Kalem(final Uretim uretim) {
        super(uretim);
    }

    @Override
    public void produce() {
        produceKalem();
    }

    public void produceKalem() {
        getUretim().produceKalem();
    }
}

// Kod 19
```

```
package com.pratikprogramci.designpatterns.bolum6.bridge.ornek2;

/**
 * Bir defteri temsil eden sınıf.
 *
 */
public class Defter extends Envanter {

    public Defter(final Uretim uretim) {
        super(uretim);
    }

    @Override
    public void produce() {
        produceDefter();
    }

    public void produceDefter() {
        getUretim().produceDefter();
    }
}
```

Kalem ve Defter sınıfları Envanter sınıfını genişlettikleri için Envanter sınıfında yer alan abstract produce() metodunu implemente etmek zorunda kalmışlardır. Sınıf konstrüktörleri Uretim interface sınıfının bir implementasyonunu (UretimImp1 ya da UretimImp2) parametre olarak alarak, üretimin nerede yapılacağını öğrenmiş olmaktadırlar. Bu sayede Envanter nesnelerini ve üretim yapan fabrikaları kod çalışmaya başladığı zaman (at runtime) bir araya getirmiş oluyoruz. Programımız bu şekilde daha esnek bir yapıya sahip olmaktadır, çünkü istediğimiz zaman Uretim interface sınıfının başka bir implementasyon sınıfını kullanarak, üretim yapan fabrikayı değiştirebiliriz. Bunun için kod üzerinde değişiklik yapmamız gerekmemektedir. Eğer modelimizi hirarşik sınıf yapıları olarak oluşturmuş olsaydık, üretim yapan fabrikalar ve envanter nesneleri arasındaki bağı köprü tasarım şablonu örneğinde olduğu gibi esnek bir şekilde tanımlayamazdık.

Modelimizi ve programımızı test etmek için aşağıda yer alan Test sınıfını kullanıyoruz:

```
// Kod 20

package com.pratikprogramci.designpatterns.bolum6.bridge.ornek2;

/**
 * Bridge tasarım şablonu test programı.
 *
 */
```

```
public class Test {

    public static void main(final String[] args) {

        /*
         * Fabrika1 de defter üretimi yapmak üzere bir UretimImpl1 nesnesi
         * kullanıyoruz.
         */
        Envanter defter = new Defter(new UretimImpl1());
        defter.produce();

        /*
         * Fabrika2 de defter üretimi yapmak üzere bir UretimImpl2 nesnesi
         * kullanıyoruz.
         */
        defter = new Defter(new UretimImpl2());
        defter.produce();

        /*
         * Fabrika1 de kalem üretimi yapmak üzere bir UretimImpl1 nesnesi
         * kullanıyoruz.
         */
        Envanter kalem = new Kalem(new UretimImpl1());
        kalem.produce();

        /*
         * Fabrika2 de kalem üretimi yapmak üzere bir UretimImpl2 nesnesi
         * kullanıyoruz.
         */
        kalem = new Kalem(new UretimImpl2());
        kalem.produce();
    }
}
```

Ekran çıktısı aşağıdaki şekilde olacaktır:

```
Fabrika1: Kareli defter imal edildi.
Fabrika2: Müzik defteri imal edildi.
Fabrika1: Tükenmez kalem imal edildi.
Fabrika2: Kursun kalem imal edildi.
```

Köprü tasarım şablonu ne zaman kullanılır?

- Sistemde tanımlanmış olan soyut sınıflar ve bu sınıfların implementasyonu ayırmak isteniyorsa; Böyle bir ayrım, programın çalışma esnasında (runtime) değişik

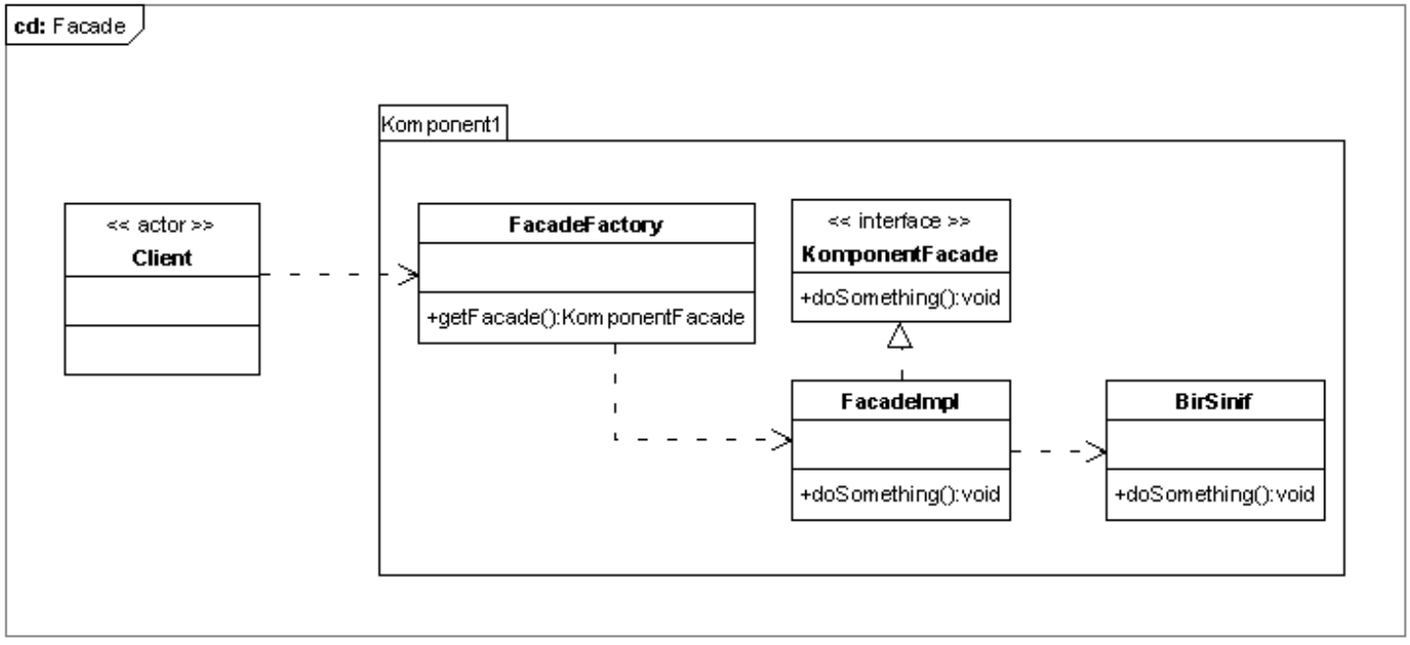
implementasyonların kullanılmasına izin vermektedir.
- Karmaşık sınıf hirarşileri modellemek yerine, köprü tasarım şablonu ile sistem daha esnek ve bakımı kolay hale getirilmek istendiğinde.

İlişkili tasarım şablonları

Abstract Factory köprü oluşumunda kullanılabilir. Adaptör tasarım şablonu ile sistem içinde bulunan sınıflar başka sınıflar tarafından kullanılacak şekilde adapte edilebilir. Adaptör tasarım şablonu, çoğu zaman sistem tasarlandıktan sonra uygulanır. Buna karşın köprü tasarım şablonu, sistemin oluşturulması esnasında, soyut sınıfları ve sahip oldukları implementasyonları ayrıştırmak için kullanılmaktadır.

Cephe (Facade)

Profesyonel yazılım sistemleri birçok komponentin birleşiminden oluşurlar. Yazılım esnasında birçok ekip birbirinden bağımsız, sistemin bütününü oluşturan değişik komponentler üzerinde çalışırlar. Bir komponent, belirli bir işlevi yerine getirmek için hazırlanmış bir ya da birden fazla Java sınıfından oluşmaktadır.



Resim 9

Bir komponentin sunmuş olduğu hizmetten yararlanabilmek için komponentin dış dünya için tanımlamış olduğu giriş/çıkış noktaları (input/output interface) kullanılır. Komponent sadece bu giriş/çıkış noktaları üzerinden dış dünya ile iletişim kurar ve iç dünyasını tamamen gizler. Bu

iletişim noktaları genelde cephe tasarım şablonu kullanılarak programlanır.

Resim 9 da yer alan UML diyagramında görüldüğü gibi Komponent1 isminde bir sistem komponenti dış dünya ile iletişimi KomponentFacade interface sınıfı üzerinden sağlamaktadır. Kullanıcı sınıfın (client) tanıması gereken sınıflar FacadeFactory ve KomponentFacade sınıflarıdır. FacadeFactory ile kullanıcı sınıfın kullanabileceği şekilde bir KomponentFacade nesnesi oluşturulmaktadır. Komponentin sunduğu hizmetlere, KomponentFacade interface sınıfında tanımlanmış metotlar aracılığıyla ulaşılmaktadır. Komponenti kullanmak için tanımlanan giriş noktası KomponentFacade.doSomething() metodudur.

KomponentFacade sınıfını bir interface olarak tanımlıyoruz. Bu sayede komponentin kullanıldığı yere göre sunduğu hizmet, kullanıcı sınıf (client) etkilenmeden değiştirilebilir hale gelmektedir. Bu amaçla komponent içinde değişik KomponentFacade implementasyon sınıfları programlanır. Kullanıcı sınıfın gereksinimleri doğrultusunda FacadeFactory tarafından gerekli görülen interface implementasyon nesnesi oluşturulur ve kullanılmak üzere kullanıcı sınıfa verilir. Örnekte gördüğümüz gibi interface sınıfları kullanarak, sistemin parçaları arasında çok esnek bağlar oluşturabiliyoruz. Esnek ve bakımı kolay sistemler oluşturmak için mutlaka interface sınıfları tercih edilmelidir.

Cephe tasarım şablonunu nasıl implemente edebileceğimizi bir örnekle inceliyelim. Önce KomponentFacade interface sınıfını tanımlıyoruz:

```
// Kod 21

package com.pratikprogramci.designpatterns.bolum6.facade.component;

/**
 * Komponentin sunduğu hizmetlerin kullanılması için oluşturulan metotların
 * bulunduğu facade interface sınıfı.
 *
 */
public interface KomponentFacade {
    /**
     * İmplementasyon sınıfları tarafından implemente edilmesi gereken bir
     * metot.
     */
    public void doSomething();
}
```

Komponentin sunduğu hizmetlerden yararlanabilmek için öncelikle bir KomponentFacade nesnesinin oluşturulması gerekmektedir. Bu görevi FacadeFactory sınıfı üstleniyor:

```
// Kod 22

package com.pratikprogramci.designpatterns.bolum6.facade.component;

/**
 * KomponentFacade oluşturmak için kullanılan singleton factory sınıfı.
 *
 */
public class FacadeFactory {

    /**
     * Singleton tasarım şablonunu kullanmak için bu sınıftan bir değişken
     * tanımlıyoruz.
     */
    private static FacadeFactory instance = new FacadeFactory();

    /**
     * Singleton olabilmesi için sınıf konstruktörünün private olması gerekiyor.
     * Bu durumda başka bir sınıf new FacadeFactory() şeklinde nesne
     * oluşturamaz. Amacımız da bunu engellemek ve bu sınıftan sadece bir
     * nesnenin sistemde bulunmasını sağlamak (singleton tasarım şablonuna
     * bakınız)
     */
    private FacadeFactory() {
    }

    /**
     * Sistemde bulunan tek FacadeFactory nesnesine ulaşmak için instance()
     * metodu kullanılır.
     *
     * @return FacadeFactory singleton nesne
     */
    public static FacadeFactory instance() {
        return instance;
    }

    /**
     * Kullanıcı sınıf tarafından kullanılmak üzere oluşturulan KomponentFacade
     * nesnesi.
     */
    public KomponentFacade getFacade() {
        return new FacadeImpl();
    }
}
```

FacadeFactory sınıfını singleton olarak implemente ediyoruz. Böylece sistem içinde sadece bir tane

FacadeFactory nesnesi bulunmuş oluyor. FacadeFactory sınıfının görevi, komponenti kullanan sınıf için bir KomponentFacade nesnesi oluşturmaktır. Bunu gerçekleştirebilmek için komponent içinde KomponentFacade interface sınıfını implemente eden sınıfların bulunması gerekmektedir. FacadeImpl isminde bir implementasyon sınıfı aşağıdaki yapıya sahiptir:

```
// Kod 23

package com.pratikprogramci.designpatterns.bolum6.facade.component;

/**
 * KomponentFacade interface sınıfının bir implementasyonu.
 *
 */
public class FacadeImpl implements KomponentFacade {

    @Override
    public void doSomething() {
        new BirSinif().doSomething();
    }
}
```

FacadeImpl sınıfı KomponentFacade interface sınıfında bulunan doSomething() metodunu impelente etmek zorundadır. FacadeImpl sınıfı implemente ettiği doSomething() metodu içinde BirSınıf sınıfından bir nesne oluşturarak, bu nesnenin doSomething metodunu kullanıyor. Bu sayade komponenti kullanan sınıfın (Test sınıfı) KomponentFacade interface sınıfında bulunan doSomething() metodunu kullanmış olması, BirSınıf.doSomething() metoduna kadar iletilmiş oluyor.

Komponenti ve sunduğu hizmeti test etmek için aşağıda yer alan Test sınıfı kullanılabilir:

```
// Kod 24

package com.pratikprogramci.designpatterns.bolum6.facade.component;

/**
 * KomponentFacade interface sınıfının bir implementasyonu.
 *
 */
public class FacadeImpl implements KomponentFacade {

    @Override
    public void doSomething() {
        new BirSinif().doSomething();
    }
```

```
}
```

Test sınıfı FacadeFactory.instance() metodunu kullanarak önce bir FacadeFactory nesnesine sahip olmaktadır. FacadeFactory sınıfı singleton olduğu için sınıf içinde oluşturulmuş ve bilgisayarın hafızasında bulunan FacadeFactory nesnesi Test sınıfına verilmektedir. FacadeFactory.getFacade() metodu üzerinden, komponentin sahip olduğu bir KomponentFacade implementasyon nesnesi oluşturulmaktadır. getFacade() metodu içinde new FacadeImpl() ile KomponentFacade interface sınıfını implemente eden bir nesne oluşturulur. Akabinde FacadeImpl.doSomething() metodu kullanılarak, komponentin KomponentFacade.doSomething() metodunda tanımlanmış olan servis Test sınıfı tarafından kullanılmış olur.

Bu şekilde modellenmiş ve implemente edilmiş bir komponentin bir çok avantajı bulunmaktadır. Bunlar:

- Komponenti kullanan sınıf (Test sınıfı) komponentin sunduğu hizmeti KomponentFacade interface sınıfında tanımlanmış metotlar üzerinden alır. Kullanıcı sınıfın tanıması gereken sadece KomponentFacade ve FacadeFactory sınıflarıdır.
- Komponent, bünyesinde barındırdığı sınıfları kullanıcı sınıf kodunun tekrar derlenmesine gerek kalmadan değiştirebilir. Komponent ile kullanıcı sınıf arasındaki tek bağ, KomponentFacade interface sınıfından tanımlanmış olan metotlardır ve bu metotlar değişmediği sürece, kullanıcı sınıf komponent içinde yapılan değişikliklerden etkinlenmez.
- Komponent, bünyesinde barındırdığı ve sunduğu hizmeti implemente eden sınıfları dış dünyadan saklar. Örneğin komponent içinde yer alan BirSınıf sınıfını kullanıcı sınıf (Test) kesinlikle tanımaz ve new BirSınıf() şeklinde kullanamaz. Eğer Test sınıfı BirSınıf sınıfını doğrudan kullanabilseydi, bu Test sınıfı ve komponent arasında sıkı bir bağ oluşturur ve komponentin yapısı değiştirildiğinde, Test sınıfını da negatif etkilerdi.
- Komponent çeşitli KomponentFacade implemenstasyon sınıfları (örneğin FacadeImpl) sunarak, verdiği hizmetin değişik yöntemlerle ve kullanıcı sınıfın ihtiyaçları doğrultusunda oluşturulmasını sağlıyabilir. Bu gibi değişikliklerden kullanıcı sınıf etkilenmez.

Cephe tasarım şablonu daha sonra detaylı olarak inceliyeceğimiz katmanlı mimarilerde, katmanlar arası izolasyonu gerçekleştirmek içinde kullanılmaktadır. Her katmanın belirlenmiş bir görevi vardır ve üst katmanlar kendi görevlerini yerine getirmek için bir alt katmanın sunduğu hizmetlerden faydalanırlar. İki katman arasındaki bağı esnek tutmak için üst katman, alt katmanın sunduğu hizmete, alt katmanda tanımlanmış olan KatmanFacade interface sınıflarından erişir. Bir katman içinde meydana gelen değişiklikler, diğer katmanları etkilemez ve böylece katmanlar arası izolasyon sağlanmış olur.

Cephe tasarım şablonu ne zaman kullanılır?

Kullanıcı sınıf ve sistemin parçalarını oluşturan altsistemler (subsystem) ya da komponentler arasındaki bağ, cephe tasarım şablonu ile esnek bir yapıda oluşturulur. Altsistemlerde oluşan değişikliklerden kullanıcı sınıflar etkilenmez. Komponent ve altsistem tasarımlarında cephe tasarım şablonu kullanılmalıdır.

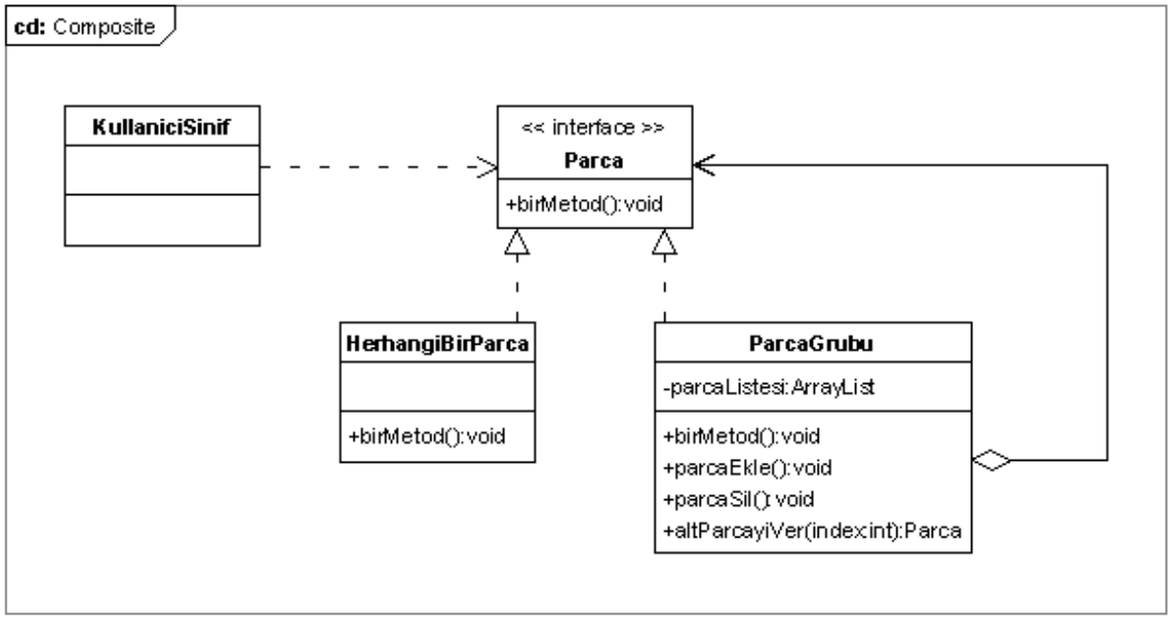
Refactoring yöntemiyle mevcut kod, altsistem ya da komponent olarak yeniden düzenlenmelidir. Bu sayede görevi ve sunduğu hizmet tanımlanmış ve bakımı kolay komponentler oluşur. Bu komponentlerin yapılandırılmalarında cephe tasarım şablonu kullanılmalıdır.

İlişkili tasarım şablonları

Facade yerine abstract factory kullanılarak sistem içinde kullanılan nesnelerin oluşumu ve kullanımı saklı tutulabilir. Facade ile yeni bir interface tanımlanırken, adaptör tasarım şablonu ile mevcut bir interface sınıf kullanılır. Facade sınıfları singleton olarak implemente edilebilirler.

Bileşik (Composite)

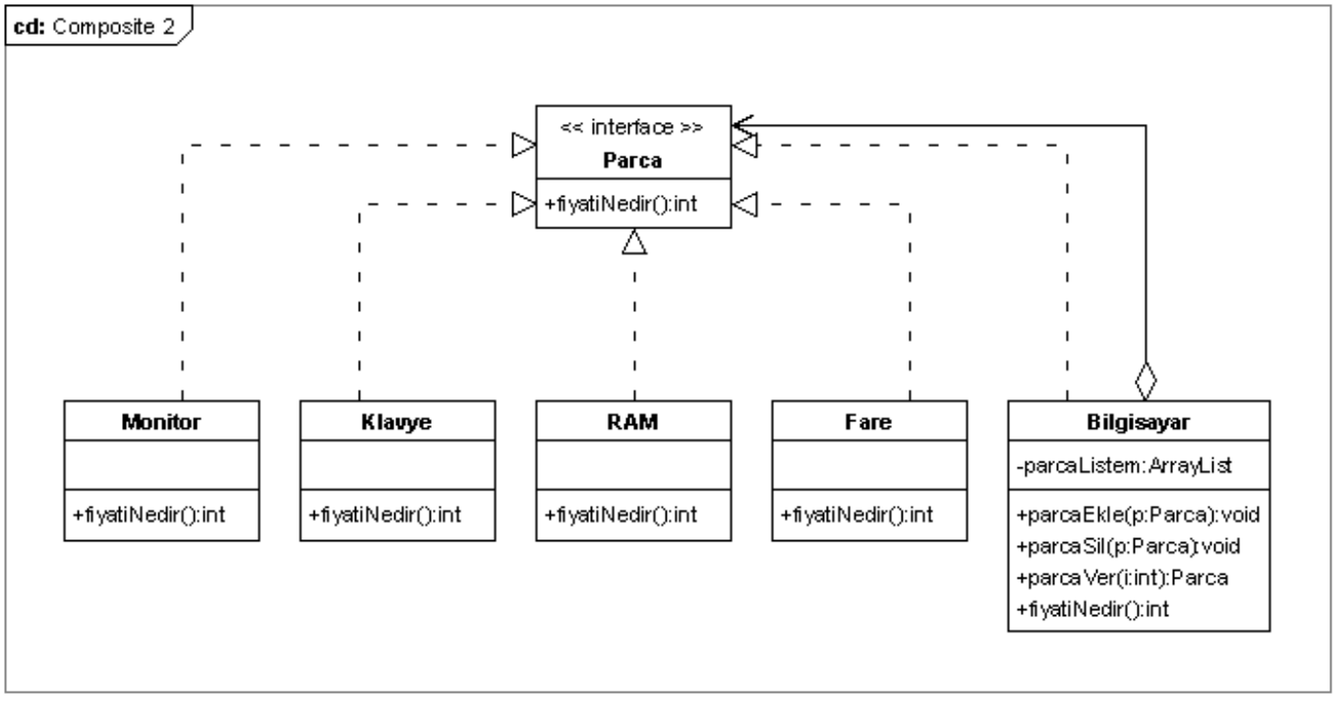
Bileşik tasarım şablonu bir sistemin bütünü ve parçaları arasındaki ilişkileri modellemek için kullanılmaktadır. Sistemin bütününü oluşturan parçalar, kendi içlerinde alt parçalardan oluşabilir. Bileşik tasarım şablonu kullanıcı sınıfın, sistem, sistemin parçaları ve alt parçalar arasında ayrım yapmadan nesneleri kullanmasına izin vermektedir. Bu şekilde sistem yazılımı ve kullanımı daha sadeleştirilmektedir.



Resim 10

Sistemin bütününü oluşturan nesneler Parca interface sınıfını implemente ederler. Kendi bünyesinde alt parçalardan oluşan parçalar (ParcaGrubu) Parca interface sınıfını implemente etmekle beraber, bünyelerinde barındırdıkları alt parçalar için bir liste (ArrayList) tanımlarlar. Bunun yanı sıra bu liste üzerinde ekleme, silme ve parça edinme işlemleri için metotlara sahiptirler. Kullanıcı sınıf bir parça grubu ve sistemin herhangi bir parçası arasında ayrım yapmaz, çünkü sistemin tüm parçaları Parca interface sınıfını implemente ederler.

Bilgisayar ve bilgisayar parçaları satımı yapan bir mağaza için bileşik tasarım şablonunu kullanarak, aşağıda yer alan modeli oluşturuyoruz:



Resim 11

Verilen bir siparişi sistemin bütünü olarak düşünürsek, sipariş içinde bulunan bir monitör ve bilgisayar siparişin parçalarıdır. Monitörün tek bir parça olduğunu farz ediyoruz. Bir bilgisayar RAM, anakart (mainboard), fare, klavye ve grafik kartı gibi değişik alt parçalardan oluşmaktadır. Bilgisayarın kendisi ve parçaları, Parca interface sınıfını implemente ederek birer parça haline gelmektedirler. Bilgayar ile sahip olduğu parçaların arasındaki tek fark, bilgisayar nesnesinin sahip olduğu parçaları tutabileceği liste ve metotlara sahip olmasıdır. Bunun haricinde kullanıcı sınıf açısından bakıldığında bir bilgisayar ve grafik kartı arasında bir farklılık yoktur, çünkü ikisi de Parca interface sınıfını implemente etmektedirler. Böyle bir farkın olmaması, kullanıcı sınıf için yazılan kodun daha sade bir hal almasını sağlamaktadır, çünkü kullanıcı sınıf if ve else yapıları ile

önce önündeki parçanın hangi sınıftan olduğunu tespit etmek zorunda kalmamaktadır. Parca interface sınıfı aşağıdaki yapıya sahiptir:

```
// Kod 25

package com.pratikprogramci.designpatterns.bolum6.composite;

public interface Parca {

    /**
     * Her altsınıf bu metodu implemente ederek, parçanın
     * fiyatını belirler.
     */
    public int fiyatiNedir();

}
```

Parca interface sınıfı bünyesinde fiyatiNedir() isminde bir metot tanımlıyarak, bu interface sınıfını implemente eden altsınıfların, parçanın birim fiyatını belirlemelerini sağlıyoruz.

Monitor, Klavye ve Ram gibi parçalar aşağıdaki yapıya sahiptirler:

```
//Kod 25

package com.pratikprogramci.designpatterns.bolum6.composite;

public class Monitor implements Parca {

    @Override
    public int fiyatiNedir() {
        return 250;
    }
}

package com.pratikprogramci.designpatterns.bolum6.composite;

public class Klavye implements Parca {

    @Override
    public int fiyatiNedir() {
        return 50;
    }
}

package com.pratikprogramci.designpatterns.bolum6.composite;
```

```
public class Ram1GB implements Parca {

    @Override
    public int fiyatiNedir() {
        return 100;
    }
}
```

Diğer sistem parçaları analog bir şekilde tanımlanır. Bileşik tasarım şablonu uygularak oluşturduğumuz bilgisayar nesnesi, diğer parçalara nazaran başka bir yapıya sahiptir. Bilgisayar sınıfı, sahip olduğu alt parçaları yönetebilmek için bir liste ve bu liste üzerinde işlem yapabilen metotlar tanımlar:

```
// Kod 26

package com.pratikprogramci.designpatterns.bolum6.composite;

import java.util.ArrayList;

/**
 * Bir bilgisayar ve parçalarını temsil eden sınıf. Bilgisayar, Parca interface
 * sınıfını implemente ettiği için kendiside bir parçadır.
 *
 */
public class Bilgisayar implements Parca {

    /*
     * Bilgisayar çeşitli parçalardan oluşur. Bu parçalar parcaListem içinde yer
     * alır. ArrayList içinde sadece Parca tipi nesneler yer alabilir.
     */
    private ArrayList<Parca> parcaListem = new ArrayList<Parca>();

    /**
     * Bilgisayarın parçalarını oluşturmak için kullanılan metot.
     */
    public void parcaEkle(final Parca parça) {
        getParcaListem().add(parça);
    }

    /**
     * Bilgisayar parçalarından herhangi birisini silmek için kullanılan metot.
     *
     */
    public void parcaSil(final Parca parça) {
```

```
        if (getParcaListem().contains(parça)) {
            getParcaListem().remove(parça);
        }
    }

    /**
     * Sıra numarası belli bir bilgisayar parçasını edinmek için kullanılan
     * metot.
     *
     */
    public Parca parcaVer(final int index) throws Exception {
        Parca parça = null;
        try {
            parça = getParcaListem().get(index);
        } catch (final Exception e) {
            throw new Exception(index + " nolu parça bulunamadı");
        }
        return parça;
    }

    public ArrayList<Parca> getParcaListem() {
        return parcaListem;
    }

    public void setParcaListem(final ArrayList<Parca> parcaListem) {
        this.parcaListem = parcaListem;
    }

    /**
     * Bilgisayar Parca interface sınıfını implemente ettiği için fiyatiNedir()
     * metodunu sunmak zorundadır. Bu metot içinde bilgisayarı oluşturan tüm
     * parçaların fiyatları toplanarak, bilgisayarın fiyatı edinilir.
     */
    @Override
    public int fiyatiNedir() {
        int fiyat = 0;

        for (int i = 0; i < getParcaListem().size(); i++) {
            fiyat += getParcaListem().get(i).fiyatiNedir();
        }
        return fiyat;
    }
}
```

Bilgisayar sınıfı Parca interface sınıfını implemente ettiği için öncelikle bir parçadır. Bünyesinde barındırdığı parçaların yer aldığı bir listeye sahip olduğu için bir bileşik (composite) nesne halini

alır. Bu liste içinde Parca interface sınıfını implemente eden diğer parçalar yer almaktadır.

Bilgisayar sınıfında da fiyatiNedir() metodunu implemente etmemiz gerekiyor. Bu metot içinde bilgisayar nesnesinin parcaListem (ArrayList) değişkeninde yer alan tüm parçaların fiyatiNedir() metodu kullanılarak, bilgisayarın toplam fiyatı tespit edilmiş olur. Kullanıcı sınıf açısından bakıldığında, Parca interface sınıfında yer alan fiyatiNedir() metodu hem normal bir parça nesnesinde hem de bilgisayar gibi bir bileşik nesne üzerinde kullanılabilmektedir. Kullanıcı sınıf normal bir parça nesne ile bileşik nesne arasında ayrım yapmak zorunda kalmaz.

Her örneğimizde olduğu gibi, tasarım şablonunun kullanıldığı Test sınıfı aşağıdaki yapıya sahiptir:

```
// Kod 27

package com.pratikprogramci.designpatterns.bolum6.composite;

public class Test {

    public static void main(final String[] args) {

        /*
         * Sipariş verilen bir monitör
         */
        final Parca monitor = new Monitor();

        /*
         * Sipariş verilen bir bilgisayar
         */
        final Bilgisayar bilgisayar = new Bilgisayar();

        /*
         * Sipariş verilen bilgisayar parçaları
         */
        final Parca grafikKarti = new GrafikKarti();
        final Parca klavye = new Klavye();
        final Parca fare = new Fare();
        final Parca modem = new Modem();
        final Parca ram1 = new Ram1GB();
        final Parca ram2 = new Ram1GB();
        final Parca ram3 = new Ram1GB();
        final Parca ram4 = new Ram1GB();
        final Parca mainboard = new MainBoard();

        /*
         * Parçalar bilgisayara ekleniyor.
         */
```

```
        bilgisayar.parcaEkle(grafikKarti);
        bilgisayar.parcaEkle(klavye);
        bilgisayar.parcaEkle(fare);
        bilgisayar.parcaEkle(modem);
        bilgisayar.parcaEkle(ram1);
        bilgisayar.parcaEkle(ram2);
        bilgisayar.parcaEkle(ram3);
        bilgisayar.parcaEkle(ram4);
        bilgisayar.parcaEkle(mainboard);

        System.out.println("Monitörğn fiyatı: " + monitor.fiyatiNedir());
        System.out.println("Bilgisayarın fiyatı: " + bilgisayar.fiyatiNedir());
        System.out.println("Toplam: "
                + (monitor.fiyatiNedir() + bilgisayar.fiyatiNedir()));
    }
}
```

Test.main() metodunda sipariş edilen monitor ve bilgisayar nesnelerini oluşturuyoruz. Bilgisayar nesnesi değişik alt parçalardan oluştugu için sipariş verilen alt parçaları tek tek oluşturduktan sonra bilgisayar.parcaEkle() metodu ile listeye ekliyoruz. Monitor nesnesinin fiyatını monitor.fiyatiNedir() metodu ile ekranda görüntülüyoruz. Aynı metodu bilgisayar nesnesi içinde kullanıyoruz. İki nesne arasında belirli bir fark görünmesede, bilgayarın fiyatını edinmek için bilgisayarın bünyesinde bulunan tüm parçaların fiyatlarının toplanması gerekiyor. Bu işlem bilgisayar.fiyatiNedir() metodu içinde yapılıyor. Bilgisayar değişik parçalardan oluştuğu için Monitor ve Bilgisayar sınıflarındaki fiyatiNedir() metodunun implementasyonları da farklıdır. Ama bu değişiklik kullanıcı sınıf (Test) için tamamen göz ardı edebileceği bir durum, çünkü monitor nesnesi de, bilgisayar nesnesi de birer Parca dır ve aynı metotlara sahip olmak zorundadırlar. Bu iki sınıf tarafından fiyatiNedir() metodunun değişik implemente edilmesine engel değildir.

Bileşik tasarım şablonu ne zaman kullanılır?

- Bir nesnenin bütünü ve parçaları arasındaki ilişkiyi modellemek için.
- Eğer kullanıcı sınıf için nesnenin parçaları ve kendisi üzerinde yapılan işlemlerin transparen olması isteniyorsa.

İlişkili tasarım şablonları

Bileşik tasarım şablonu dekoratör tasarım şablonu ile beraber kullanılabilir. İki tasarım şablonunun beraber kullanılabilmesi için her iki tasarım şablonu için kullanılan sınıfların aynı üst sınıftan olmaları gerekmektedir. Sadece bu durumda Decorator nesneleri parcaEkle(), parcaSil() ve parcaVer() metotlarına sahip olabilir. Döngücü (iterator) tasarım şablonu kullanılarak, composite nesnenin parçaları ve bunların alt parçaları üzerinde işlem yapılabilir. Flyweight tasarım şablonu

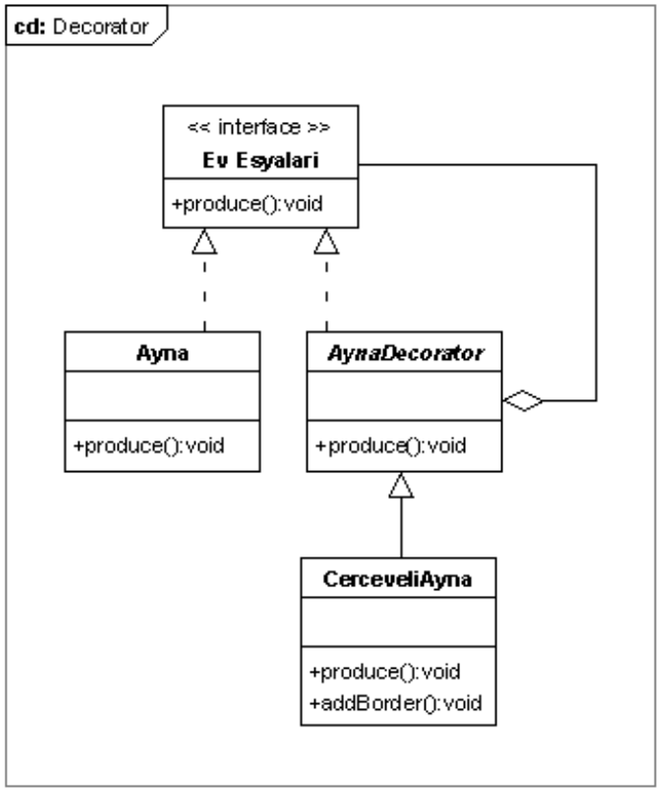
ile nesneler ve sahip oldukları alt parçalar ortak kullanılabilir.

Dekoratör (Decorator)

Mevcut bir sınıf hirarşisini ya da sınıfın yapısını değiştirmeden, oluşturulan nesnelere yeni özelliklerin eklenme işlemini gerçekleştirmek için dekoratör tasarım şablonu kullanılmaktadır.

Altsınıfların oluşturulması yöntemiyle sınıflara yeni özelliklerin eklenmesi, daha sonra sisteme eklenecek altsınıflar için değiştirilmesi zor kalıpların oluşmasını beraberinde getirmektedir. Bu durumda üstsınıflarda tanımlanmış olan bazı özellikler statik ve altsınıflar için değistirilemez ya da kullanımı engellenemez bir hal alabilir. Kullanıcı sınıflar için de bu durum sorun teşkil edebilmektedir, çünkü kendi istekleri doğrultusunda bir nesnenin ne zaman ve nasıl oluşturulması gerektiğini yönlendiremeyebilirler.

Nesnelere sahip oldukları sınıfların yapılarının değistirilmeden yeni özelliklerin eklenmesini sağlayan dekoratör tasarım şablonu ile istenilen özelliklerin ekleneceği nesne başka bir nesne içine gömülür. Yeni özellik eklenen nesneyi içine alan nesneye dekoratör ismi verilir. Dekoratör nesnesi ile yeni özellik eklenen nesne aynı üstsınıfa dahil olduklarından, birbirleriyle değiştirilebilir haldedirler. Bu özellikten dolayı kullanıcı sınıf, dekoratör sınıf ile dekoratör nesne bünyesinde bulunan diğer nesne arasında ayrım yapmaz. Nesneler arası ilişkiye aşagıda yer alan UML diyagramında görmekteyiz.



Resim 12

Şimdi dekoratör tasarım şablonunun nasıl uygulandığını bir örnek üzerinde birlikte inceleyelim. Ev eşyaları üzerine uzmanlaşmış bir firma, bir fabrikadan hammadde halinde ayna satın alıp, bunları çeşitli ürünler haline getirdikten sonra sonra pazarlamaktadır. Programcı olarak bize verilen görev, mevcut Ayna sınıfını kullanarak bir üretim modeli oluşturmaktır. Ayna sınıfı üzerinde bir değişiklik yapılması istenmemektedir. Daha ziyade, değişik aynalı ürünleri modellemek için yeni sınıfların oluşturulması istenmektedir. Pazarlama firması ilk etapta değişik çerçevelere sahip aynaları piyasaya sürmeyi planladığından, hazırlıyacağımız program bu ihtiyaca cevap vermelidir.

Pazarlama firmasının ihtiyaçlarını analiz ettikten sonra, dekoratör tasarım şablonu ile istenilen modelin oluşturulabileceği kanaatına varıyoruz. Bir ayna nesnesini kullanarak, çerçeveli bir ayna nesnesi oluşturabiliriz. Burada yapılması gereken tek işlem, mevcut bir ayna nesnesine çerçeve eklemektir yani bir ayna nesnesini değişik özellikler kullanarak (örneğin çerçeve sahibi olması) dekore etmektir. EvEsylari interface sınıfı ve Ayna sınıfı aşağıdaki yapıya sahiptirler ve bizim tarafımızdan değiştirilmeden program içinde kullanılmaktadırlar.

```
// Kod 28

package com.pratikprogramci.designpatterns.bolum6.decorator;
```

```
/**
 * Bir ev esyasını temsil eden interface sınıf.
 *
 */
public interface EvEsyalari {

    /**
     * Üretimi gerçekleştirmek için kullanılan metot.
     */
    public void produce();
}


// Kod 29

package com.pratikprogramci.designpatterns.bolum6.decorator;

public class Ayna implements EvEsyalari {

    @Override
    public void produce() {
        System.out.println("Ayna imal edildi.");
    }
}
```

Çerveveli aynalar üretebilmek için EvEsyalari interface sınıfını implemente eden AynaDecorator sınıfını aşağıdaki şekilde programlıyoruz:

```
// Kod 30

package com.pratikprogramci.designpatterns.bolum6.decorator;

/**
 * Değişik tipte aynalı ürünleri temsil eden üstsınıf.
 *
 */
public abstract class AynaDecorator implements EvEsyalari {
    /*
     * Bünyesinde mevcut bir ayna nesnesi bulundurur ve değişik metotlar
     * kullanarak bu ayna nesnesini dekore eder.
     */
    private EvEsyalari ayna = new Ayna();

    public EvEsyalari getAyna() {
        return ayna;
    }
```

```
    public void setAyna(final EvEsyalari ayna) {
        this.ayna = ayna;
    }
}
```

AynaDecorator sınıfı nesne katma (aggregation) yöntemiyle bünyesinde bir Ayna nesnesi tutmaktadır. Dekorasyonu, yani aynaya bir çerçeve ekleme işlemini gerçeklestirmebilmek için Ayna nesnesi ile böyle bir bağ oluşturulması gerekmektedir. Ayna nın üretilme işlemi Ayna.produce() metoduna delege edildikten sonra, dekorasyon için gerekli metot, yani aynaya çerçeve takma işlemi AynaDecorator ya da bir altsınıfı tarafından kullanılacaktır.

AynaDecorator sınıfını soyut olarak tanımlıyoruz. Amacımız değişik türdeki aynaları üretebilmek için esnek bir model oluşturabilmektir. Bugün çerçeveli aynalar üretimi için kullanılan programımız, yarın yeni bir altsınıf oluşturularak, başka türde bir aynanın üretimine izin verebilecek yapıda olmalıdır. Bu yüzden AynaDecorator isminde bir soyut sınıf tanımlıyarak, gelecekte üretimi yapılacak tüm aynalı ürünler için bir baz oluşturmuş oluyoruz. Çerçeveli aynaların üretimi için CerceveliAyna sınıfını oluşturuyoruz.

```
// Kod 31

package com.pratikprogramci.designpatterns.bolum6.decorator;

/**
 * Çerceveli ayna üretimi yapmak için kullanılan sınıf.
 *
 */
public class CerceveliAyna extends AynaDecorator {

    /**
     * Üretim için kullanılan sınıf. addBorder metodu ile aynaya cerceve ekler.
     */
    @Override
    public void produce() {
        getAyna().produce();
        addBorder();
    }

    /**
     * Cerceve ekleme işlemini gerçekleştirmek için kullanılan metot.
     *
     */
    public void addBorder() {
        System.out.println("Aynaya cerceve eklendi.");
```

```
    }
}
```

CerceveliAyna sınıfı, ayna üretim işlemini üstsınıfı AynaDecorator üzerinden gerçekleştirmektedir. AynaDecorator sınıfında ayna isminde bir sınıf değişkeni olduğu için CerceveliAyna sınıfı getAyna() metodu ile üretilen aynaya ulaşabilmektedir. Üretimi yapılmış bir aynaya çerçeve takmak için addBorder() metodu tanımlanmıştır.

CerveveliAyna.produce() metodu içinde önce ayna üretilmekte, daha sonra addBorder() metodu ile bu aynaya bir çerçeve takılmaktadır yani ayna dekore edilmiş olmaktadır.

Sonuç itibariyle aynanın ve aynalı ürünlerin üretimini birbirinden ayırmış olduk ve mevcut bir nesneye (Ayna), sahip olduğu sınıfın yapısını değiştirmeden yeni bir özellik (çerçeve) ekleyebildik. Aşağıda yer alan Test sınıfı ile çerçeveli ayna üretimine başlayabiliriz:

```
// Kod 32

package com.pratikprogramci.designpatterns.bolum6.decorator;

public class Test {

    public static void main(final String[] args) {
        final EvEsyalari ayna = new CerceveliAyna();
        ayna.produce();
    }
}
```

Ekran çıktısı aşagıdaki şekilde olacaktır:

```
Ayna imal edildi.
Aynaya cerceve eklendi.
```

Dekoratör tasarım şablonu ile dekore edilmiş nesneler zincirleme tarzı diğer dekorasyon işlemleri için de kullanılabilirler. Örneğin çercevesi takılmış bir ayna, AynaliDolapDecorator sınıfı bünyesinde aynalı bir dolabın oluşturulmasında tekrar kullanılabilir. Bu durumda fabrikadan çıkan ayna önce çerçeve takılarak çerçeveli bir ayna nesnesi ve daha sonra aynalı bir dolabın üretilmesi için kullanılabilir. Ayna nesnesi iki sefer dekore edilerek aynalı bir dolabın üretimi gercekleşmiş olacaktır.

Dekoratör tasarım şablonu ne zaman kullanılır?

- Mevcut nesnelere sınıf yapıları değiştirilmeden yeni özelliklerin eklemesi gerektiği durumlarda.

- Sınıflara daha sonra kullanımdan kaldırılmak üzere yeni metot ve fonksiyonların eklemesi gerektiği durumlarda.
- Sınıf yapısının ya da sınıf hirarşilerinin yeni özelliklerin eklenmesine izin vermedikleri durumlarda.

İlişkili tasarım şablonları

Dekoratör ve adaptör arasındaki tek farklılık, dekoratör tasarım şablonunun nesnenin sunduğu fonksiyonları değiştirmesidir. Dekoratör nesnenin dış dünyaya sunduğu metot yapılarını değiştirmez. Adaptör tasarım şablonu ile nesnenin dış dünyaya sunduğu metot yapıları tamamen değiştirilir.

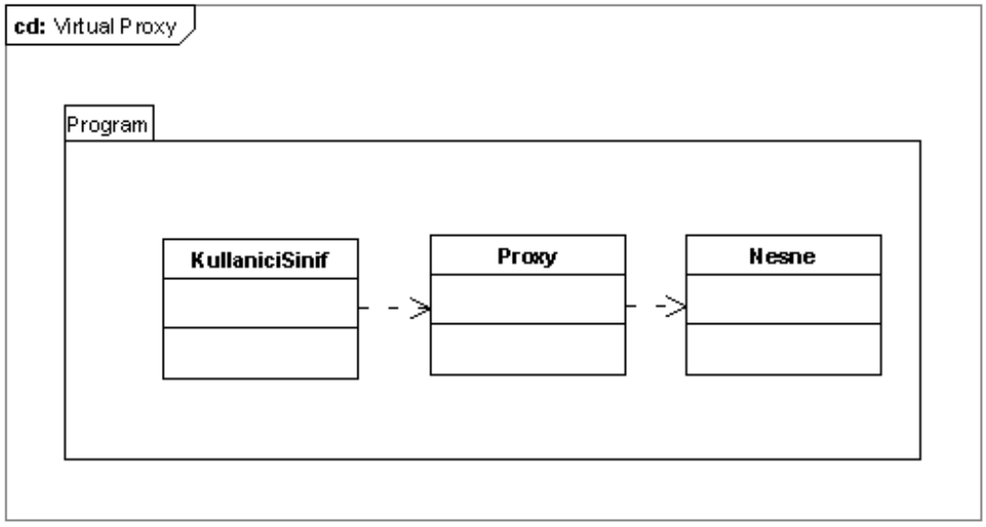
Dekoratör ile bir nesnenin dış görüntüsü değiştirilir. Strateji tasarım şablonu nesnenin iç dünyasını değiştirir. Kullanılan strateji tipine göre nesne başka türlü bir davranış sergileyecektir.

Vekil (Proxy)

Gerçek hayatta insanlar kendilerine iş takibi, alım-satım ve benzeri işler için vekil tayin ederler. Vekil, vekil olduğu kişiyi diğer şahıslara karşı temsil eder ve vekil ile onlar arasında aracı olur.

Yazılım sistemlerinde de vekil yapıları oluşturmak ve kullanmak mümkündür. Bunun en güzel örneğini vekil tasarım şablonu oluşturmaktadır.

Vekil nesnesini (proxy) KullaniciSinif ve Nesne arasında aracılık yapan bir yapı olarak düşünebiliriz. Bu ilişkiyi resim 13 de görmekteyiz.



Resim 13

Vekil tasarım şablonunu dört değişik kullanım şekli bulunmaktadır. Bunlar

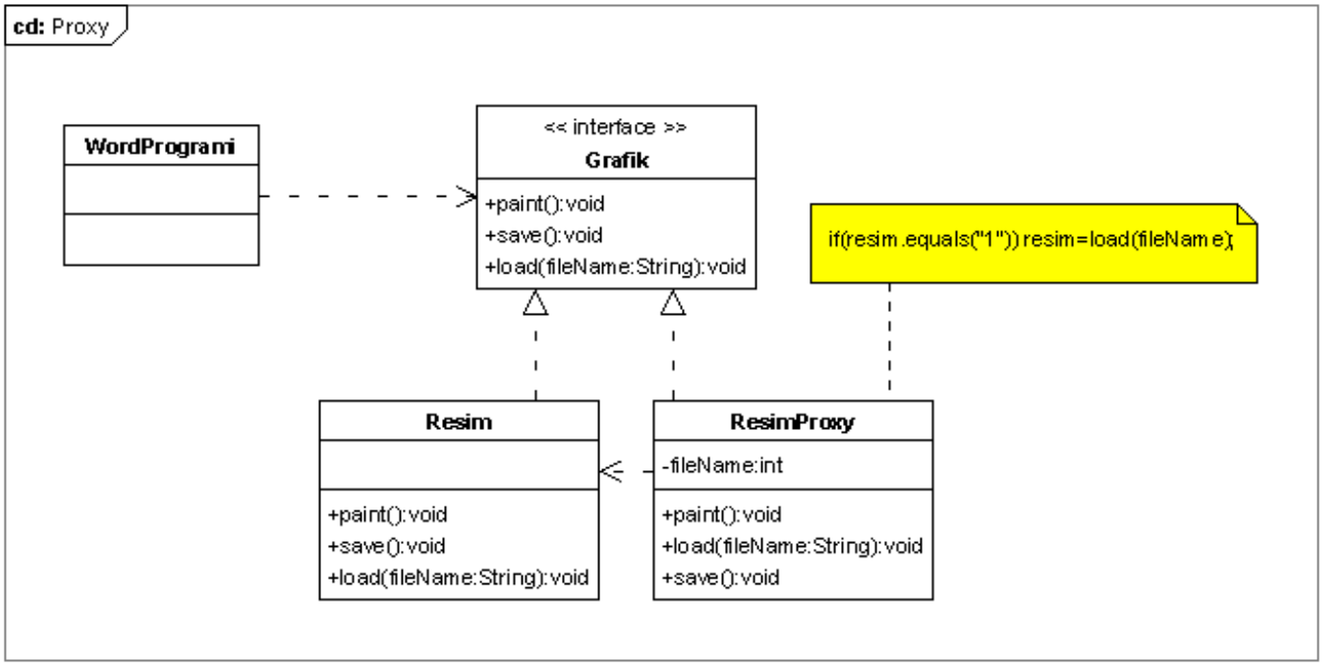
- Sanal vekil (virtual proxy)
- Koruyucu vekil (protection proxy)
- Dinamik vekil
- Uzak vekil (remote proxy)

Bu bölümde vekil tasarım şablonunun değişik türlerinin nasıl implemente edildigini örnekler üzerinde inceleyeceğiz. Örneklere geçmeden önce, vekilin nasıl çalıştığına tekrar bir göz atalım.

Oluşturulmaları zaman alıcı ve sistem kaynaklarını zorlayan, korunmaya ihtiyaç duyan ya da uzak sunucularda bulunan kaynaklara erişimi kolaylaştıran ve kullanıcı için erişimi transparen kılan nesnelere vekil (proxy) nesneleri adı verilmektedir. Bu nesneler vekil oldukları nesnelerin tüm metotlarına sahiptirler ve kullanıcı sınıf ile vekil olunan nesne arasında aracılık yaparlar. Vekil olan nesne, kullanıcı sınıfa, vekil olunan nesne gibi davranır ve kullanıcı sınıftan gelen tüm istekleri vekil olduğu nesneye iletir. Böyle bir yapının kullanılmasının sebebi, gerek olmadığı sürece vekil olunan nesnenin oluşturulmasını engellemektir ya da vekil olunan nesneyi gizlemektir. Böylece vekil olunan nesneye dışardan erişimlerde kontrol altına alınmış olur. Yazılan programın yapısına göre değişik tipte vekil nesneler kullanılabilir. Şimdi bu vekil türlerini yakından inceleyelim.

Sanal Vekil (Virtual Proxy)

İçinde yüzlerce sayfanın ve resmin bulunduğu bir Microsoft Office dökümanı, sanal vekil nesnelerin kullanımını açıklamak için iyi bir örnektir. Döküman açıldığında, döküman içinde bulunan tüm resimler bir seferde Office programı tarafından yüklenmez. Eğer böyle olsa idi, dökümanın ilk sayfasını görmek çok uzun zaman alırdı, çünkü Office programı uzun bir süre resimleri yüklemek ile meşgul olurdu. Office programının nasıl çalıştığını tam olarak bilmemekle beraber, vekil nesnelerin kullanıldığını düşünebiliriz. Döküman içinde yer alan tüm resimler için bir vekil nesne oluşturulur. Bir resim için bir vekil nesnesinin oluşturulması, temsil ettiği resmin yüklenmesi ve gösterilmesi anlamına gelmez. Daha ziyade vekil nesne, döküman içinde resim yerine oluşturulan bir yer tutucu öğe (placeholder) olarak düşünülebilir.

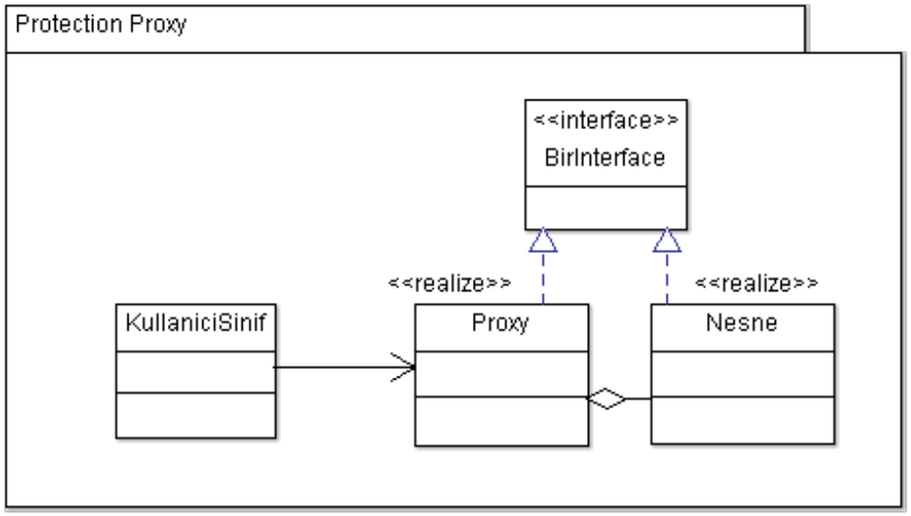


Resim 14

Vekil, yerini aldığı resmin boyutlarını tanıyor olabilir ve Office programına yüklenmesi gereken resmin eni ve boyu hakkında bilgi verebilir. Resmin yüklenme işlemi ise Office programı içinde resmin yer aldığı sayfanın gösterimi ile gerçekleşir. Resmin bulunduğu sayfa gösterilmediği sürece, vekil nesne resmin yerini alır ve resmi temsil eder. Office programı resmin bulunduğu sayfaya gelindiğinde vekil nesnesinin ciz() ya da goster() metodunu kullanarak, resmin sayfa içinde gösterilmesini sağlar. Aslında resmin gösterilme işlemi vekil nesne tarafından değil, vekil nesnesinin ciz() ya da goster() metoduna gelen çağrının asıl resim nesnesinin bu isimdeki metoduna delege edilmesiyle gerçekleşir. Bu sayede gerek duyulmadığı sürece, yani resmin yer aldığı sayfa gösterimde olmadığı sürece asıl resim nesnesi oluşturulmaz.

Koruyucu Vekil (Protection Proxy)

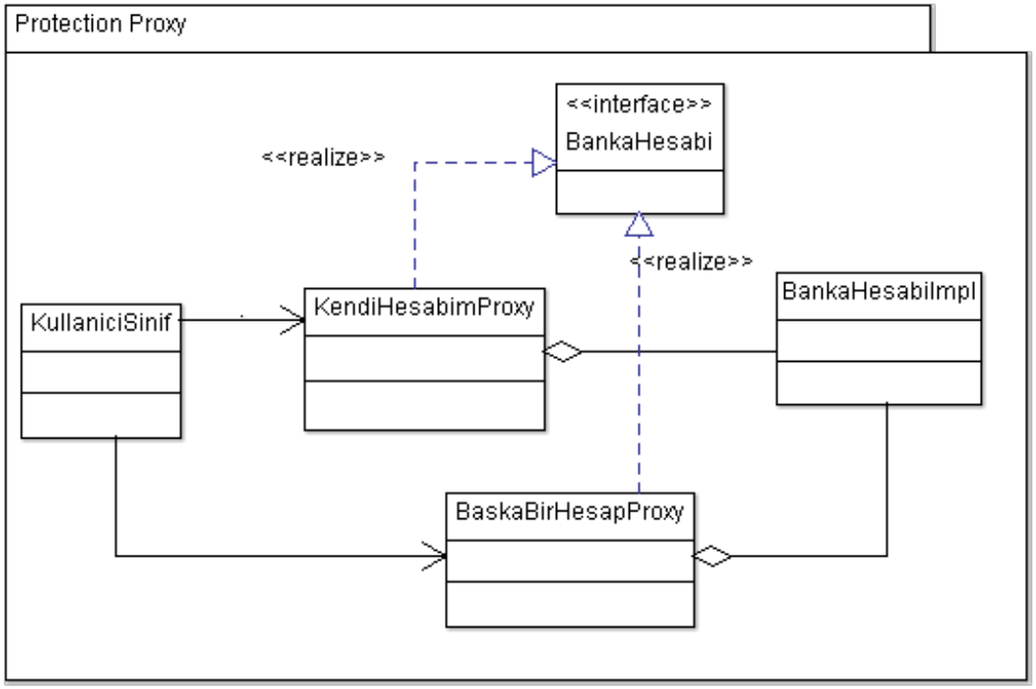
Bir nesnenin başka nesneler tarafından kontrolsüz bir şekilde kullanımını engellemek için koruyucu vekil tasarım şablonu kullanılmaktadır.



Resim 15

UML diyagramında yer alan Nesne Proxy tarafından korunan nesnedir. Nesne sınıfı Birinterface sınıfını implemente etmektedir. Bu durumda Nesne yi korumak için oluşturacağımız Proxy nesnesinin de bu interface sınıfını implemente etmesi gerekmektedir. Sadece bu durumda Proxy nesnesi Nesne nesnesi gibi davranabilir.

Koruyucu vekil tasarım şablonunun nasıl uygulandığını şimdi bir örnek üzerinde yakından inceliyelim:



Resim 16

Bir banka hesabını modellemek için BankaHesabi interface sınıfını tanımlıyoruz. Bu interface içinde paraYatir(), paraCek(), getIsim(), getSoyad() gibi metotlar bulunmaktadır. Bir müşteriye açılan banka hesabını temsil etmek için BankaHesabiImpl isminde, BankaHesabi interface sınıfını implemente eden bir sınıf oluşturuyoruz.

Kendi hesabıma para yatırabilir ve hesaptan para çekebilirim. Başka bir şahsın banka hesabına para yatırabilir ama hesaptan para çekemem, çünkü bu hesap bana ait değildir. Her banka müşterisi için bir BankaHesabiImpl nesnesi oluşturulacağından, nesnelerin kullanımının kontrol altına alınması gerekmektedir. Sistemin, benim başka bir müşterinin hesabından para çekmemi engellemesi gerekmektedir. Başka bir hesaptan para çekme işlemini engellemek için bir Proxy sınıfı hazırlamamız gerekiyor. Bunun yanı sıra kendi hesabım üzerinde işlemler yapabilmek için ikinci bir Proxy sınıfına ihtiyacım var. Bunlar:

- KendiHesabimProxy: Bu vekil kendi hesabım üzerinde paraCek(), paraYatir() ve get ile başlayan tüm metotları kullanmama izin vermektedir.
- BaskaBirHesapProxy: Bu vekil başka bir şahsın hesabından para çekmeme izin vermez. Başka bir hesaba para yatırabilirim.

BankaHesabi interface sınıfı aşağıdaki yapıdadır:

```
// Kod 33

package com.pratikprogramci.designpatterns.bolum6.proxy.aggregated;

/**
 * Bir banka hesabını modelleyen interface sınıf.
 *
 */
public interface BankaHesabi {

    public int paraCek(int miktar) throws Exception;

    public void paraYatir(int miktar);

    public String getIsim();

    public void setIsim(String isim);

    public String getSoyad();

    public void setSoyad(String soyad);

    public int getHesapDurumu();
```

```
    public void setHesapDurumu(int hesapDurumu);

    public boolean iptalEt();
}
```

BankaHesabiImpl sınıfı BankaHesabı interface sınıfını implemente ederek, müşteriler için banka hesabı oluşturulmasında kullanılır.

```
// Kod 34

package com.pratikprogramci.designpatterns.bolum6.proxy.aggregated;

/**
 * Bir şahsın banka hesabını temsil eden sınıf.
 *
 */
public class BankaHesabiImpl implements BankaHesabi {

    private String isim;
    private String soyad;
    private int hesapDurumu = 0;
    private boolean iptalEdildi = false;

    public BankaHesabiImpl(final String isim, final String soyad,
            final int miktar) {
        setIsim(isim);
        setSoyad(soyad);
        setHesapDurumu(miktar);
    }

    public BankaHesabiImpl() {

    }

    /**
     * Belirli bir miktar parayı hesaptan çeker.
     *
     */
    @Override
    public int paraCek(final int miktar) throws Exception {
        if (getHesapDurumu() - miktar > 0) {
            setHesapDurumu(getHesapDurumu() - miktar);
        } else {
            throw new Exception("Hesabinizda yeterinde " + "para bulunmuyor.");
        }
```

```
        return miktar;
    }

    /**
     * Hesaba bir miktar para yatırmak için kullanılan metot.
     *
     */
    @Override
    public void paraYatir(final int miktar) {
        setHesapDurumu(getHesapDurumu() + miktar);
    }

    /**
     * Bir banka hesabını iptal etmek için kullanılan metot. Sadece banka
     * çalışanları tarafından bu metot kullanılabilir. Hesap sahibi kendi
     * hesabını iptal edemez.
     */
    @Override
    public boolean iptalEt() {
        setIptalEdildi(true);
        return isIptalEdildi();
    }

    @Override
    public String getIsim() {
        return isim;
    }

    @Override
    public void setIsim(final String isim) {
        this.isim = isim;
    }

    @Override
    public String getSoyad() {
        return soyad;
    }

    @Override
    public void setSoyad(final String soyad) {
        this.soyad = soyad;
    }

    @Override
    public int getHesapDurumu() {
        return hesapDurumu;
    }
```

```
    @Override
    public void setHesapDurumu(final int hesapDurumu) {
        this.hesapDurumu = hesapDurumu;
    }

    public boolean isIptalEdildi() {
        return iptalEdildi;
    }

    public void setIptalEdildi(final boolean iptalEdildi) {
        this.iptalEdildi = iptalEdildi;
    }
}
```

Kendi banka hesabım üzerinde işlem yapabilmek için KendiHesabimProxy isminde bir vekil sınıfı oluşturuyorum. Bu vekil kendi banka hesabım üzerinde gerekli tüm işlemleri yapmama izin verecek şekilde programlanacak.

```
// Kod 35

package com.pratikprogramci.designpatterns.bolum6.proxy.aggregated;

/**
 * Kendi banka hesabim üzerinde gerekli işlemlerin yapılmasına izin
 * veren sınıf
 *
 */
public class KendiHesabimProxy implements BankaHesabi {

    private BankaHesabi hesap;

    public KendiHesabimProxy(final BankaHesabi bh) {

        this.hesap = bh;
    }

    public BankaHesabi getHesap() {
        return hesap;
    }

    public void setHesap(final BankaHesabi hesap) {
        // Bu metodu sadece banka çalışanları koşturabilir.
        throw new RuntimeException();
    }
```

```
    @Override
    public int paraCek(int miktar) throws Exception {
        return getHesap().paraCek(miktar);
    }

    @Override
    public void paraYatir(int miktar) {
        getHesap().paraYatir(miktar);

    }

    @Override
    public String getIsim() {
        return getHesap().getIsim();
    }

    @Override
    public void setIsim(String isim) {
        // Bu metodu sadece banka çalışanları koşturabilir.
        throw new RuntimeException("Bu motodu koşturmak için izniniz bulunmuyor.");
    }

    @Override
    public String getSoyad() {
        return getHesap().getSoyad();
    }

    @Override
    public void setSoyad(String soyad) {
        // Bu metodu sadece banka çalışanları koşturabilir.
        throw new RuntimeException("Bu motodu koşturmak için izniniz bulunmuyor.");
    }

    @Override
    public int getHesapDurumu() {
        return getHesap().getHesapDurumu();
    }

    @Override
    public void setHesapDurumu(int hesapDurumu) {
        // Bu metodu sadece banka çalışanları koşturabilir.
        throw new RuntimeException("Bu motodu koşturmak için izniniz bulunmuyor.");
    }

    @Override
    public boolean iptalEt() {
        // Bu metodu sadece banka çalışanları koşturabilir.
```

```
        throw new RuntimeException("Bu motodu koşturmak için izniniz bulunmuyor.");
    }
}
```

Daha sonra detaylı olarak göreceğimiz Test sınıfı bünyesinde aşağıdaki şekilde bir vekil nesnesi oluşturabiliriz.

```
/**
 * Kendi banka hesabim üzerinde işlem yapabilmek için oluşturulan dynamic
 * proxy.
 *
 */
public BankaHesabi getBaskaHesapProxy(final BankaHesabi hesap) {
    return new KendiHesabimProxy(hesap);
}
```

Vekil ile korumak istediğimiz nesne (BankaHesabiImpl) arasındaki bağı oluşturmak için KendiHesabimProxy konstrüktörüne korunmasını istediğimiz BankaHesabi nesnesini parametre olarak veriyorum. KendiHesabimProxy sınıfı kendisine gelen tüm istekleri koruduğu nesneye yönlendirebilir. Asıl koruma işlemi de bu yönlendirme sürecinde sağlanmaktadır. Vekil izin verilmeyen işlemleri korunan nesneye iletmez ve bu şekilde nesne korunmuş olur ve vekilin arkasında saklı kalır.

Şimdi vekil içinde bir banka hesap nesnesinin nasıl korunduğunu inceliyelim. Hazırladığımız vekil tipine göre bazı metotların kullanılmasını engelleyebilir, bazılarının kullanılmasina izin verebiliriz. İzin verdiğimiz işlemler vekil tarafından korunan nesneye su şekilde yonlendirilecektir:

```
@Override
public int paraCek(int miktar) throws Exception {
    return getHesap().paraCek(miktar);
}

@Override
public void paraYatir(int miktar) {
    getHesap().paraYatir(miktar);

}

@Override
public void setHesapDurumu(int hesapDurumu) {
    // Bu metodu sadece banka çalışanları koşturabilir.
    throw new RuntimeException("Bu motodu koşturmak
            için izniniz bulunmuyor.");
```

```
}
```

Kullanma izni olan metotları doğrudan kosturabiliyorum. Eğer bir metodu kullanma iznim yoksa, o metot bünyesinde bir hata nesnesi (RuntimeException) oluşuyor ve bu şekilde vekil yapılmasına izin verilmeyen işlemleri engellemiş oluyor. Eğer bir işlemi yapma hakkım varsa, vekil bu istegi doğrudan bünyesinde barındırdığı hesap nesnesine yönlendiyor.

Başka bir şahsın hesabından para çekilmesini engellemek için BaskaBirHesapInvocationHandler sınıfını oluşturuyoruz.

```
// Kod 36

package com.pratikprogramci.designpatterns.bolum6.proxy.aggregated;

/**
 * Başka bir hesap üzerinde para çekme işlemini engelleyen proxy
 * sınıfı
 *
 */
public class BaskaBirHesapProxy implements BankaHesabi {

    private BankaHesabi hesap;

    public BaskaBirHesapProxy(final BankaHesabi bh) {
        this.hesap = bh;
    }

    public BankaHesabi getHesap() {
        return hesap;
    }

    public void setHesap(final BankaHesabi hesap) {
        // Bu metodu sadece banka çalışanları koşturabilir.
        throw new RuntimeException("Bu motodu koşturmak için izniniz bulunmuyor.");
    }

    @Override
    public int paraCek(int miktar) throws Exception {
        throw new Exception("Başka bir hesaptan para cekemezsiniz!");
    }

    @Override
    public void paraYatir(int miktar) {
        getHesap().paraYatir(miktar);
```

```
    }

    @Override
    public String getIsim() {
        return getHesap().getIsim();
    }

    @Override
    public void setIsim(String isim) {
        // Bu metodu sadece banka çalışanları koşturabilir.
        throw new RuntimeException("Bu motodu koşturmak için izniniz bulunmuyor.");
    }

    @Override
    public String getSoyad() {
        return getHesap().getSoyad();
    }

    @Override
    public void setSoyad(String soyad) {
        // Bu metodu sadece banka çalışanları koşturabilir.
        throw new RuntimeException("Bu motodu koşturmak için izniniz bulunmuyor.");
    }

    @Override
    public int getHesapDurumu() {
        // Bu metodu sadece banka çalışanları koşturabilir.
        throw new RuntimeException("Bu motodu koşturmak için izniniz bulunmuyor.");
    }

    @Override
    public void setHesapDurumu(int hesapDurumu) {
        // Bu metodu sadece banka çalışanları koşturabilir.
        throw new RuntimeException("Bu motodu koşturmak için izniniz bulunmuyor.");
    }

    @Override
    public boolean iptalEt() {
        // Bu metodu sadece banka çalışanları koşturabilir.
        throw new RuntimeException("Bu motodu koşturmak için izniniz bulunmuyor.");
    }
}
```

Bu implementasyonda görüldüğü gibi paraYatir() metodunun kullanımına izin verilmistir ama para çekme işlemi yasaklanmıştır. Eğer başka bir hesaptan para çekmeye çalışırsak, bu durumda bir Exception ile uyarı alırız.

Her örneğimizde olduğu gibi bir Test sınıfi ile hazırladığımız programı test edelim:

```
// Kod 37

package com.pratikprogramci.designpatterns.bolum6.proxy.aggregated;

public class Test {

    public static void main(final String[] args) throws Exception {
        final Test test = new Test();
        test.start();
    }

    public void start() throws Exception {
        final BankaHesabi hesap =
            new BankaHesabiImpl("Ahmet", "Yildirim", 1000);
        final BankaHesabi kendiHesabimProxy = getKendiHesabimProxy(hesap);
        kendiHesabimProxy.paraYatir(2000);
        kendiHesabimProxy.paraCek(100);

        System.out.println("Hesabtaki para miktarı: "
                    + kendiHesabimProxy.getHesapDurumu());

        final BankaHesabi baskaHesapProxy = getBaskaHesapProxy(hesap);

        baskaHesapProxy.paraYatir(500);
        baskaHesapProxy.paraCek(200);
    }

    /**
     * Kendi banka hesabim üzerinde işlem yapabilmek için oluşturulan
     * proxy.
     *
     */
    protected BankaHesabi getKendiHesabimProxy(final BankaHesabi hesap) {

        return new KendiHesabimProxy(hesap);
    }

    /**
     * Başka bir şahsın banka hesabim üzerinde işlem yapabilmek için
     * oluşturulan proxy.
     */
    protected BankaHesabi getBaskaHesapProxy(final BankaHesabi hesap) {
        return new BaskaBirHesapProxy(hesap);
    }
```

```
}
```

Test sınıfı çalıştırıldığında, aşağıdaki ekran görüntüsü oluşacaktır:

```
Hesabtaki para miktarı: 2900
Exception in thread "main" java.lang.Exception:
            Başka bir hesaptan para cekemezsiniz!
    at com.pratikprogramci.designpatterns.bolum6.proxy
        .aggregated.BaskaBirHesapProxy.paraCek(BaskaBirHesapProxy.java:27)
    at com.pratikprogramci.designpatterns.bolum6.proxy
        .aggregated.Test.start(Test.java:21)
    at com.pratikprogramci.designpatterns.bolum6.proxy.aggregated
        .Test.main(Test.java:7)
```

start() metodu bünyesinde Ahmet Yıldırım isimli bir şahsın banka hesabını ediniyoruz. KendiHesabimProxy yardımı ile bir vekil oluşturduğumuz için bu hesap üzerinde birçok işlemi gerçekleştirebiliriz. Örnekte yer aldığı gibi önce hesaba 2000 YTL yatırıyor ve akabinde 100 YTI hesaptan çekiyoruz.

KendiHesabimProxy kullanıldığı için Ahmet Yıldırım kendi hesabına para yatırabiliyor ve çekebiliyor. Daha sonra BaskaBirHesapProxy yardımı ile başka bir şahsın banka hesabı üzerinde işlem yapıyoruz. paraYatir() metodunu kullanarak bu hesaba 500 YTL yatırıyoruz. paraCek() metodunu kullanmak istediğimizde yukarıda yer alan Exception oluşuyor, çünkü BaskaBirHesapProxy yabancı bir hesaptan para çekmemizi engelliyor.

Dinamik Vekil

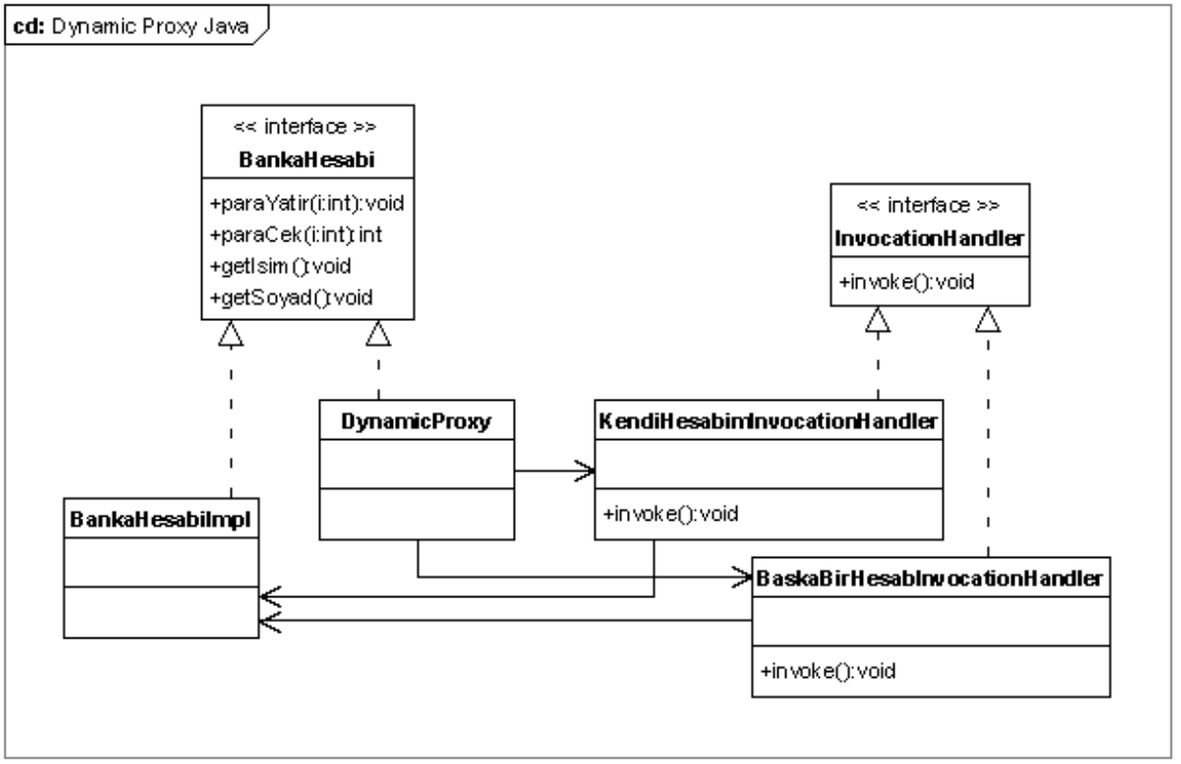
Koruyu vekil tasarım şablonunu kendi imkanlarımızla implemente ettik. Şimdi aynı işlemin Java dilindeki imkanlar aracılığı ile nasıl yapıldığına bir göz atalım.

Java dilinde koruyucu vekil tasarım şablonunu uygulamak için java.lang.reflect paketinde tanımlanmış olan sınıflar kullanılmaktadır. Bu sınıflar yardımı ile programın çalışma esnasında (at runtime) dinamik vekil nesneler oluşturulup, korunması gereken nesneler kontrol altına alınmaktadır. Kullanılan vekil nesnesi, programın çalışma esnasında (at runtime) oluşturulduğu için dinamik vekil ismini taşımaktadır.

Proxy nesnesi Java tarafından otomatik olarak oluşturulur. Bu sınıfı programlamamız gerekmiyor. Gerekli kodu Proxy sınıfına ekleyemeyeceğimiz için ikinci bir sınıfın oluşturulması gerekmektedir. Java da dinamik vekiller kullanabilmek için InvocationHandler sınıfını implemente eden bir sınıf oluşturmamız gerekmektedir. InvocationHandler sınıfının görevi, vekil nesnesinin isteklerini

karşılamaktır. Vekil nesnesinin bir metodu kullanıldığında, bu vekil nesnesi tarafından InvocationHandler nesnesine yönlendirilir. Akabinde InvocationHandler, korunan nesnenin aynı isimdeki metodunu koşturarak, korunan nesne üzerinde gerekli işlemi gerçekleştirir.

Koruyucu vekil tasarım şablonunu dinamik vekil nesneleri aracılığı nasıl uygulandığını şimdi bir örnek üzerinde yakından inceliyelim:



Resim 17

Bir banka hesabını modellemek için BankaHesabi interface sınıfını tanımlıyoruz. Bu interface içinde paraYatir(), paraCek(), getIsim(), getSoyad() gibi metotlar bulunmaktadır. Bir müşteriye açılan banka hesabını temsil etmek için BankaHesabiImpl isminde, BankaHesabi interface sınıfını implemente eden bir sınıf oluşturuyoruz.

Kendi hesabıma para yatırabilir ve hesaptan para çekebilirim. Başka bir şahsın banka hesabına para yatırabilir ama hesaptan para çekemem, çünkü bu hesap bana ait değildir. Her banka müşterisi için bir BankaHesabiImpl nesnesi oluşturulacağından, nesnelerin kullanımının kontrol altına alınması gerekmektedir. Sistemin, benim başka bir müşterinin hesabından para çekmemi engellemesi gerekmektedir. Başka bir hesaptan para çekme işlemini engellemek için bir InvocationHandler hazırlamamız gerekiyor. Bunun yanı sıra kendi hesabım üzerinde işlemler yapabilmek için ikinci bir InvocationHandler sınıfına ihtiyacım var. Bunlar:

- KendiHesabimInvocationHandler: Bu InvocationHandler sınıfı, kendi hesabım üzerinde paraCek(), paraYatir() ve get ile başlayan tüm metotları kullanmama izin vermektedir.
- BaskaBirHesabInvocationHandler: Bu InvocationHandler sınıfı, başka bir şahsın hesabından para çekmeme izin vermez. Başka bir hesaba para yatırabilirim.

BankaHesabi interface sınıfı aşağıdaki yapıdadır:

```
// Kod 38

package com.pratikprogramci.designpatterns.bolum6.proxy.dynamic;

/**
 * Bir banka hesabını modelleyen interface sınıf.
 *
 */
public interface BankaHesabi {
    public int paraCek(int miktar) throws Exception;

    public void paraYatir(int miktar);

    public String getIsim();

    public void setIsim(String isim);

    public String getSoyad();

    public void setSoyad(String soyad);

    public int getHesapDurumu();

    public void setHesapDurumu(int hesapDurumu);

    public boolean iptalEt();
}
```

BankaHesabiImpl sınıfı BankaHesabı interface sınıfını implemente ederek, müşteriler için banka hesabı oluşturulmasında kullanılır.

```
// Kod 39

package com.pratikprogramci.designpatterns.bolum6.proxy.dynamic;

/**
 * Bir şahsın banka hesabını temsil eden sınıf.
 *
```

```
 */
public class BankaHesabiImpl implements BankaHesabi {

    private String isim;
    private String soyad;
    private int hesapDurumu = 0;
    private boolean iptalEdildi = false;

    public BankaHesabiImpl(final String isim, final String soyad,
            final int miktar) {
        setIsim(isim);
        setSoyad(soyad);
        setHesapDurumu(miktar);
    }

    public BankaHesabiImpl() {

    }

    /**
     * Belirli bir miktar parayı hesaptan çeker.
     *
     */
    @Override
    public int paraCek(final int miktar) throws Exception {
        if (getHesapDurumu() - miktar > 0) {
            setHesapDurumu(getHesapDurumu() - miktar);
        } else {
            throw new Exception("Hesabinizda yeterinde " + "para bulunmuyor.");
        }
        return miktar;
    }

    /**
     * Hesaba bir miktar para yatırmak için kullanılan metot.
     *
     */
    @Override
    public void paraYatir(final int miktar) {
        setHesapDurumu(getHesapDurumu() + miktar);
    }

    /**
     * Bir banka hesabını iptal etmek için kullanılan metot. Sadece banka
     * çalışanları tarafından bu metot kullanılabilir. Hesap sahibi kendi
     * hesabını iptal edemez.
     */
```

```
    @Override
    public boolean iptalEt() {
        setIptalEdildi(true);
        return isIptalEdildi();
    }

    @Override
    public String getIsim() {
        return isim;
    }

    @Override
    public void setIsim(final String isim) {
        this.isim = isim;
    }

    @Override
    public String getSoyad() {
        return soyad;
    }

    @Override
    public void setSoyad(final String soyad) {
        this.soyad = soyad;
    }

    @Override
    public int getHesapDurumu() {
        return hesapDurumu;
    }

    @Override
    public void setHesapDurumu(final int hesapDurumu) {
        this.hesapDurumu = hesapDurumu;
    }

    public boolean isIptalEdildi() {
        return iptalEdildi;
    }

    public void setIptalEdildi(final boolean iptalEdildi) {
        this.iptalEdildi = iptalEdildi;
    }
}
```

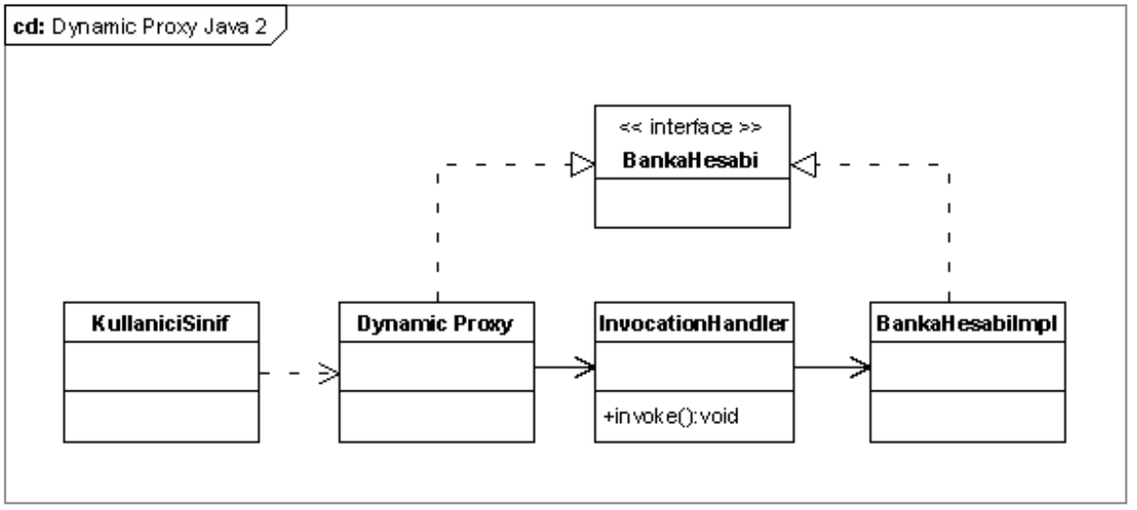
Kendi banka hesabım üzerinde işlem yapabilmek için KendiHesabimInvocationHandler isminde bir

InvocationHandler oluşturuyorum. Bu InvocationHandler sınıfı, kendi banka hesabım üzerinde gerekli tüm işlemleri yapmama izin verecek şekilde programlanacak. Sistem tarafından otomatik olarak oluşturulacak dinamik vekil nesne, bu sınıfın invoke() metodunu koşturarak, hesap nesnesi üzerinde işlem yapacaktır.

```
// Kod 40

package com.pratikprogramci.designpatterns.bolum6.proxy;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

/**
 * Kendi banka hesabim üzerinde gerekli işlemlerin yapılmasına izin veren
 * InvocationHandler sınıfı
 *
 */
public class KendiHesabimInvocationHandler implements InvocationHandler {

    private BankaHesabi hesap;

    public KendiHesabimInvocationHandler(final BankaHesabi bh) {
        setHesap(bh);
    }

    /**
     * Dynamic proxy invoke metodunu kullanarak, hesap nesnesi üzerinde
     * işlem yapabilir.
     */
    @Override
    public Object invoke(final Object proxy, final Method method,
            final Object[] args) throws Throwable {
        try {
            if (method.getName().startsWith("paraYatir")) {
                return method.invoke(getHesap(), args);
            } else if (method.getName().startsWith("paraCek")) {
                return method.invoke(getHesap(), args);
            } else if (method.getName().startsWith("get")) {
                return method.invoke(getHesap(), args);
            }
        } catch (final Exception e) {
            e.printStackTrace();
        }
        return null;
    }
```

```
    public BankaHesabi getHesap() {
        return hesap;
    }

    public void setHesap(final BankaHesabi hesap) {
        this.hesap = hesap;
    }
}
```

Nesnelerin ve sahip oldukları metotların kullanım sırasını tekrar bir UML diyagramında görelim:



Resim 18

Herhangi bir kullanıcı sınıf (KullaniciSinif) sistem tarafından oluşturulmuş dinamik vekil (Dynamic proxy) üzerinde herhangi bir BankaHesabi metodunu kullanmak istiyor. Dynamic proxy, BankaHesabi interface sınıfını implemente ettiği için KullaniciSinif bu interface sınıfında tanımlanan tüm metotları kullanabilir, örneğin paraYatir(). Dynamic proxy nesnesinin herhangi bir metodu kullanılmak istendiğinde, dinamik vekil bu isteği otomatik olarak sahip olduğu InvocationHandler sınıfına delege eder, yani InvocationHandler sınıfının invoke() metodu kullanır.

Daha sonra detaylı olarak göreceğimiz Test sınıfı bünyesinde, aşağıdaki şekilde bir dynamic proxy nesnesi oluşturabiliriz.

```
/**
 * Başka bir şahsın banka hesabim üzerinde işlem yapabilmek için oluşturulan
 * dynamic proxy.
 *
 */
protected BankaHesabi getBaskaHesapProxy(final BankaHesabi hesap) {
```

```
    return (BankaHesabi) Proxy.newProxyInstance(hesap.getClass()
            .getClassLoader(), hesap.getClass().getInterfaces(),
            new BaskaBirHesapInvocationHandler(hesap));
}
```

InvocationHandler ile korumak istediğimiz nesne (BankaHesabiImpl) arasındaki bağı oluşturmak için KendiHesabimInvocationHandler konstrüktörüne korunmasını istediğimiz BankaHesabi nesnesini parametre olarak veriyorum. KendiHesabimInvocationhandler sınıfı, dinamik vekilden gelen tüm istekleri, bünyesinde barındırdığı BankaHesabi nesnesine delege eder.

Şimdi InvocationHandler içinde bir banka hesap nesnesinin nasıl korunduğunu inceliyelim. Hazırladığımız InvocationHandler tipine göre, bazı metotların kullanılmasını engelliyebilir, bazılarının kullanılmasina izin verebiliriz. Bu işlemi InvocationHandler.invoke() metodu içinde gerçekleştiriyoruz:

```
/**
 * Dynamic proxy invoke metodunu kullanarak, hesap nesnesi üzerinde işlem
 * yapabilir.
 */
@Override
public Object invoke(final Object proxy, final Method method,
        final Object[] args) throws Throwable {
    try {
        if (method.getName().startsWith("paraYatir")) {
            return method.invoke(getHesap(), args);
        } else if (method.getName().startsWith("paraCek")) {
            return method.invoke(getHesap(), args);
        } else if (method.getName().startsWith("get")) {
            return method.invoke(getHesap(), args);
        }
    } catch (final Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

Yukarıda yer alan satırlar KendiHesabimInvocationHandler sınıfından alıntıdır. Dinamik vekil tarafından InvocationHandler.invoke() metodu koşturulduğunda, vekilin kendisi, vekil üzerinde kullanılan metot ve parametre listesi invoke() metoduna iletilir. Bu veriler sayesinde InvocationHandler, korunan nesnenin hangi metodunu hangi parametrelerle kullanılması gerektiğini tespit edebilir.

method.getName() ile proxy üzerinde kullanılan metodun ismini tespit edebiliriz. Şimdi

InvocationHandler sınıfının istenilen metodu korunan nesneye nasıl ilettiğini görelim:

```
if(method.getName().startsWith("paraYatir")){
      return method.invoke(getHesap(), args);
}
```

Eğer vekil üzerinde proxy.paraYatir() metodu kullanıldı ise, bunu if ile kontrol edip, tespit ediyoruz. Hesap nesnesi InvocationHandler sınıfında sınıf değişkeni olarak mevcut olduğundan (nesneyi InvocationHandler konstrüktörüne parametre olarak vermiştik) getHesap() ile bu nesneye ulaşabiliriz. Örneğin proxy.paraYatir(1000) şeklinde bir metot kullanımı olmuşsa, method değişkeninde "paraYatir", args içinde 1000 değeri olacaktır.

```
return method.invoke(getHesap(), args);
= (eşittir)
return getHesap().paraYatir(1000);
```

InvocationHandler hesap nesnesinin metodunu kullandıktan sonra, neticeyi proxy nesnesine geri verir ve görevini tamamlamış olur.

KendiHesabimInvocationHandler.invoke() metodu içinde paraYatir, paraCek ve get ile başlayan tüm metotların kullanımına izin verilmektedir. Başka bir şahsın hesabından para çekilmesini engellemek için BaskaBirHesapInvocationHandler sınıfını oluşturuyoruz.

```
// Kod 41

package com.pratikprogramci.designpatterns.bolum6.proxy.dynamic;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

/**
 * Başka bir hesap üzerinde para çekme işlemini engelleyen InvocationHandler
 * sınıfı
 *
 */
public class BaskaBirHesapInvocationHandler implements InvocationHandler {

    private BankaHesabi hesap;

    public BaskaBirHesapInvocationHandler(final BankaHesabi bh) {
        setHesap(bh);
    }
```

```
    /**
     * Dynamic proxy invoke metodunu kullanarak, hesap nesnesi üzerinde
     * işlem yapabilir.
     */
    @Override
    public Object invoke(final Object proxy, final Method method,
            final Object[] args) throws Throwable {
        try {
            if (method.getName().startsWith("paraYatir")) {
                return method.invoke(getHesap(), args);
            } else if (method.getName().startsWith("paraCek")) {
                throw new Exception("Başka bir hesaptan "
                        + "para cekemezsiniz!");
            } else if (method.getName().startsWith("get")) {
                return method.invoke(getHesap(), args);
            }
        } catch (final Exception e) {
            e.printStackTrace();
        }
        return null;
    }

    public BankaHesabi getHesap() {
        return hesap;
    }

    public void setHesap(final BankaHesabi hesap) {
        this.hesap = hesap;
    }
}
```

invoke() metodunda yer aldığı gibi paraYatir() metodunun kullanımına izin verilmiştir, ama para çekme işlemi yasaklanmıştır. Eğer başka bir hesaptan para çekmeye çalışırsak, bu durumda bir Exception ile uyarı alırız.

Her örneğimizde olduğu gibi bir Test sınıfi ile hazırladığımız programı test edelim:

```
// Kod 42

package com.pratikprogramci.designpatterns.bolum6.proxy.dynamic;

import java.lang.reflect.Proxy;

public class Test {

    public static void main(final String[] args) throws Exception {
```

```
        final Test test = new Test();
        test.start();
    }

    /**
     * Start
     *
     * @throws Exception
     */
    public void start() throws Exception {
        final BankaHesabi hesap = new BankaHesabiImpl("Ahmet", "Yildirim", 1000);
        final BankaHesabi kendiHesabimProxy = getKendiHesabimProxy(hesap);
        kendiHesabimProxy.paraYatir(2000);
        kendiHesabimProxy.paraCek(100);

        System.out.println("Hesabtaki para miktarı: "
                + kendiHesabimProxy.getHesapDurumu());

        final BankaHesabi baskaHesapProxy = getBaskaHesapProxy(hesap);

        baskaHesapProxy.paraYatir(500);
        baskaHesapProxy.paraCek(200);
    }

    /**
     * Kendi banka hesabim üzerinde işlem yapabilmek için oluşturulan
     * dynamic proxy.
     *
     */
    protected BankaHesabi getKendiHesabimProxy(final BankaHesabi hesap) {
        return (BankaHesabi) Proxy.newProxyInstance(hesap.getClass()
                .getClassLoader(), hesap.getClass().getInterfaces(),
                new KendiHesabimInvocationHandler(hesap));
    }

    /**
     * Başka bir şahsın banka hesabim üzerinde işlem yapabilmek için oluşturulan
     * dynamic proxy.
     *
     */
    protected BankaHesabi getBaskaHesapProxy(final BankaHesabi hesap) {
        return (BankaHesabi) Proxy.newProxyInstance(hesap.getClass()
                .getClassLoader(), hesap.getClass().getInterfaces(),
                new BaskaBirHesapInvocationHandler(hesap));
    }
}
```

Test sınıfı çalıştırıldığında, aşağıdaki ekran görüntüsü oluşacaktır:

```
Hesabtaki para miktarı: 2900
java.lang.Exception: Başka bir hesaptan para cekemezsiniz!
    at
    com.pratikprogramci.designpatterns.bolum6.proxy.
            BaskaBirHesapInvocationHandler
        .invoke(BaskaBirHesapInvocationHandler.java:30)
    at com.sun.proxy.$Proxy0.paraCek(Unknown Source)
    at com.pratikprogramci.designpatterns.bolum6.proxy.Test.start(Test.java:29)
    at com.pratikprogramci.designpatterns.bolum6.proxy.
            Test.main(Test.java:9)
Exception in thread "main" java.lang.NullPointerException
    at com.sun.proxy.$Proxy0.paraCek(Unknown Source)
    at com.pratikprogramci.designpatterns.bolum6.proxy.Test.start(Test.java:29)
    at com.pratikprogramci.designpatterns.bolum6.proxy.Test.main(Test.java:9)
```

start() metodu bünyesinde Ahmet Yıldırım isimli bir şahsın banka hesabını ediniyoruz. KendiHesabimInvocationHandler yardımı ile bir dynamic proxy oluşturduğumuz için bu hesap üzerinde birçok işlemi gerçekleştirebiliriz. Örnekte yer aldığı gibi önce hesaba 2000 YTL yatırıyor ve akabinde 100 YTL hesaptan çekiyoruz. KendiHesabimInvocationHandler kullanıldığı için Ahmet Yıldırım kendi hesabına para yatırabiliyor ve çekebiliyor. Daha sonra BaskaBirHesapInvocationHandler yardımı ile başka bir şahsın banka hesabı üzerinde işlem yapıyoruz. paraYatir() metodunu kullanarak bu hesaba 500 YTL yatırıyoruz. paraCek() metodunu kullanmak istediğimizde yukarıda yer alan Exception oluşuyor, çünkü BaskaBirHesapInvocationHandler yabancı bir hesaptan para çekmemizi invoke() metodunda engelliyor.

Test sınıfının ekran çıktısında dikkat çeken bir nokta daha bulunmaktadır:

```
at $Proxy0.paraCek(Unknown Source)
```

$Proxy0, Java Reflection API tarafından oluşturulan dynamic proxy sınıfıdır. Stacktrace içinde görüldüğü gibi, Test sınıfının 29. satırında sonra işlem $Proxy0 sınıfına devredilmektedir. Test sınıfının 26. satırında aşağıdaki komut yer almaktadır:

```
baskaHesapProxy.paraCek(200);
```

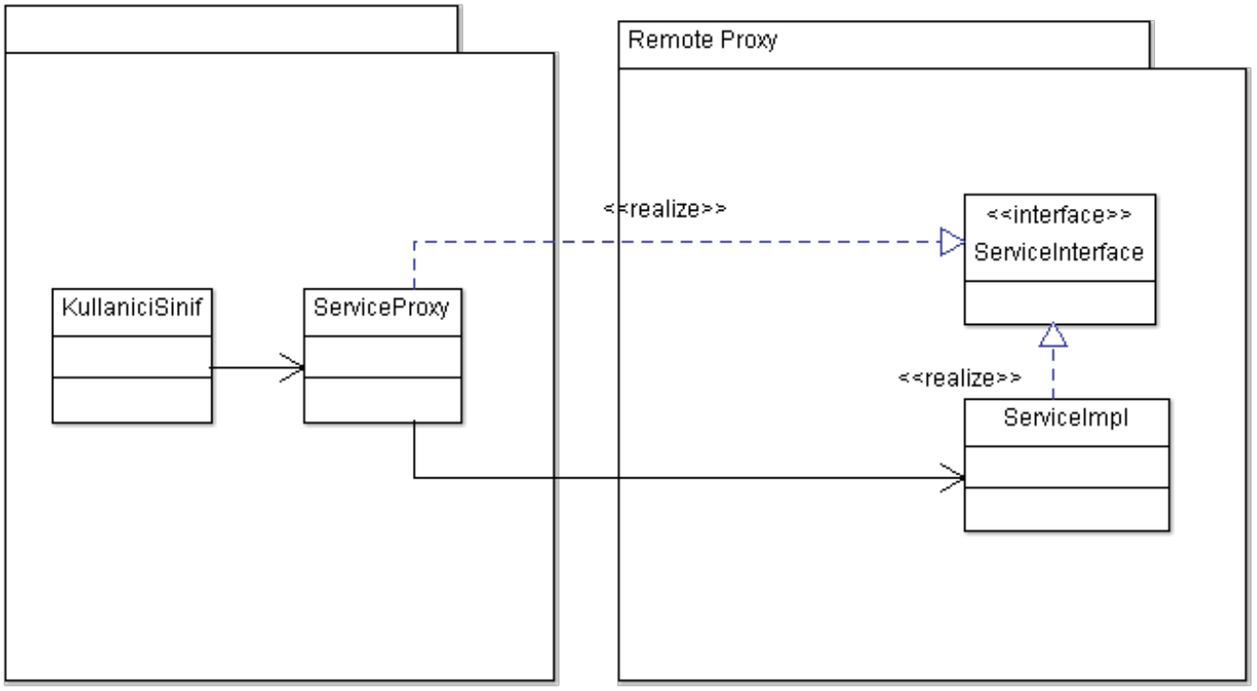
Stacktrace içinde görüldüğü gibi baskaHesapProxy.paraCek(200), $Proxy0.paraCek() halini almıştır. $Proxy0.paraCek() satırının bir üstünde BaskaBirHesapInvocationHandler sınıfının 30. satırında bir Exception oluşmuştur. Bu invoke() metodunda paraCek() metodunun kontrol edildiği

satırdır. BaskaBirHesapInvocationHandler yabancı bir hesaptan para çekilmesine izin vermedigi için bu sınıfın 30. satırında bir Exception oluşmaktadır. Stacktrace bünyesinde yer aldığı gibi kullandığımız program $Proxy0 nesnesinin metodunu kullanmakta, $Proxy0 bu işlemi kullanılan InvocationHandler sınıfına devretmekte ve InvocationHandler() sınıfıda korunan nesnenin gerekli metodunu işletmektedir.

Uzak Vekil

Vekil tasarım şablonunun diğer bir kullanım alanı, başka sunucular üzerinde bulunan servislere olan erişimin kontrol edilmesi ve kullanıcı için transparen hale getirilmesidir. Burada kullanılan vekil türlerine remote proxy ismi verilmektedir.

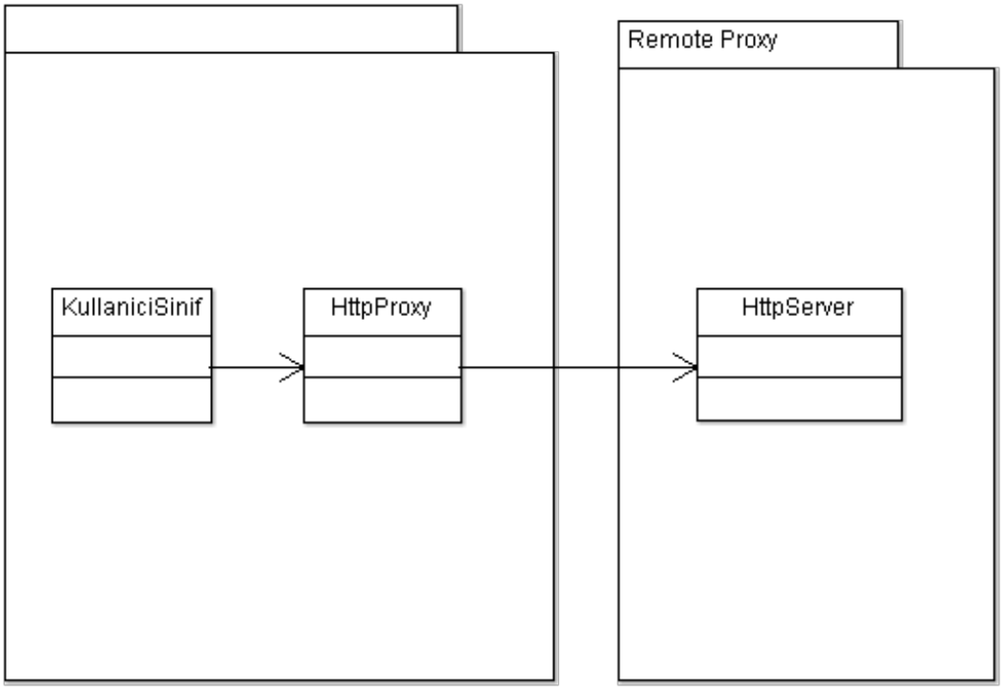
Resim 19 da remote proxy tasarım şablonunun kullanım şekli yer almaktadır. ServiceProxy isimli sınıf başka bir sunucuda yer alan ServiceImpl sınıfına olan erişimi sağlamaktadır. ServiceProxy ServiceInterface sınıfını implemente ettiğinden dolayı kullanıcısı için ServiceImpl sınıfından olan bir nesnenin yerine geçebilmekte ve kullanıcısına onun gibi davranabilmektedir. Sunucu ile olan tüm iletişim ServiceProxy bünyesinde implemente edilmektedir. Bu şekilde KullaniciSinif ServiceImpl sınıfının varlığından haberdar olmaz.



Resim 19

Remote proxy tasarım şablonunu protokol bazında da implemente etmek mümkündür. Resim 20 de bir HttpProxy implementasyonu yer almaktadır. Bu proxy HTTP protokolünü kullanarak, bir sunucudan gerekli bilgileri edinmektedir. Bu örneğin bir HTML sayfasındaki adres bilgileri olabilir.

KullaniciSinif bu remote proxy implementasyonu sayesinde HTTP protokolünden bihaber kendi işlemlerini gerçekleştirebilmektedir. Buna benzer şekilde Rest, Webservice, SSH vb. servisleri için benzer remote proxy implementasyonlari oluşturmak mümkündür.



Resim 20

Vekil tasarım şablonu ne zaman kullanılır?

- Oluşturulması zaman alıcı ve sistem kaynaklarını zorlayan nesnelere vekil olmak için sanal vekil tasarım şablonu kullanılır.
- Bir nesneyi ve metotlarını dışardan erişimlere karşı korumak için koruyucu tasarım şablonu kullanılır.
- Başka bir sunucuda bulunan servislere erişimi kullanıcı için transparen hale getirmek için remote proxy tasarım şablonu kullanılır.

İlişkili tasarım şablonları

Adaptör tasarım şablonu adapte ettiği nesnenin metot isimlerini değiştirir. Vekil tasarım şablonu ise kullandığı ve koruduğu nesne ile aynı metotlara sahiptir, çünkü nesne ile aynı interface sınıfını implemente eder. Bu koruyucu vekil nesneleri için geçerli değildir. Koruyucu vekil bazı metotların kullanılmasını engelleyebilir. Bu gibi durumlarda koruyucu vekil koruduğu nesnenin sahip olduğu metotların bir alt kümesine sahip olacaktır.

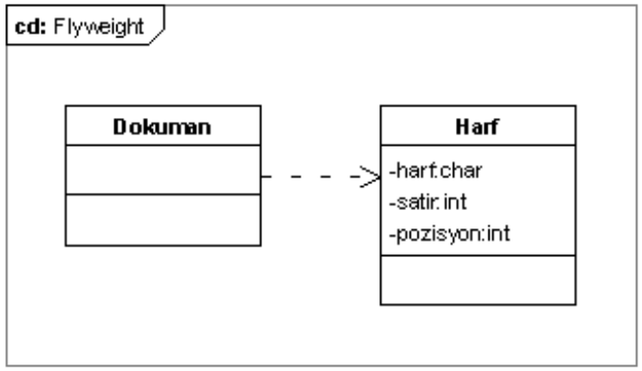
Dekoratör ve vekil nesneleri birbirlerine benzer implemente ediliyor olmalarına rağmen, iki

tasarım şablonu değişik amaçlar için kullanılırlar. Dekoratör tasarım şablonu bir nesneye yeni metotlar eklerken, vekil tasarım şablonu bir nesneyi korumak ve onun yerine geçmek için kullanılır.

Sinek Siklet (Flyweight)

Nesneye yönelik programlama dillerinde sınıflar ve bu sınıflardan oluşturulan nesneler kullanır. Bazen aynı sınıftan yüzlerce, belki binlerce nesne oluşturup, kullanıyor olabiliriz. Bu gibi durumlarda çok nesne oluşturulduğu için sistem performansı kötüye gidebilir. Sinek siklet tasarım şablonu kullanılarak, kullanılan nesne adedini aşağıya çekebiliriz.

Bu satırlar oluşurken, büyük bir ihtimalle kullandığım editör sinek siklet tasarım şablonunu kullanıyor olabilir. Yazdığım her cümle kelimelerden, her kelime birden fazla harften oluşmaktadır. Kullandığım editörün Java dilinde yazıldığını ve her harf için bir nesne kullandığını farzedersek, bir satırlık döküman için 80 ila 100 arası harf nesnesi oluşturması gerekmektedir. 100 satırlık bir döküman için bu 10.000 civarı harf nesnesinin oluşturulması anlamına gelmektedir.

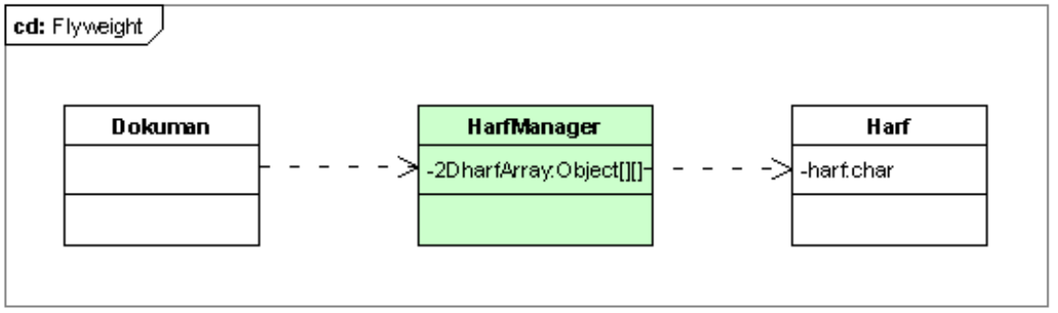


Resim 21

UML diyagramında da görüldüğü gibi her harf bulunduğu satırı ve pozisyonu bilmektedir.

Sinek siklet tasarım şablonu yardımı ile gerekli harf nesne adedini toplamda elliye düşürebiliriz. Her kelime ve satırda aynı harfler mutlaka birden fazla kullanılacaktır. Örneğin “kullanılacaktır” kelimesi içinde üç L harfi, üç A harfi, iki ı harfi vs. mecvuttur. L harfi için üç yeni L harfi nesnesi oluşturmak yerine, L harfini bir kere oluşturup, üç yerde kullanabiliriz. Böylece onlarca L harfi kullanmaktansa, L harfini bir kere oluşturup, L harfinin gerektiği her yerde kullanabiliriz. Aynı işlemi alfabedeki diğer harf, rakam ve işaretler için de yaptığımız zaman, sistem bünyesinde kullanılan harf nesnesi adedi elliyi geçmeyecektir. Her harf nesne olarak bir kere mevcut olacağından, harfin nerede kullanıldığı bilgisini başka bir mekanizma aracılığıyla sabitlememiz

gerekiyor. İmplementasyonumuzu aşağıdaki şekilde değiştiriyoruz.



Resim 22

```
// Kod 43

package com.pratikprogramci.designpatterns.bolum6.flyweight;

/**
 * Alfabenin bir harfini temsil eder.
 *
 */
public class Harf {
    private String harf;

    public Harf(final String h) {
        harf = h;
    }

    public String getHarf() {
        return harf;
    }

    public void setHarf(final String harf) {
        this.harf = harf;
    }
}
```

Tüm sistem bünyesinde, her harf sadece bir nesne olarak kullanılacağı için bu nesnelerin sahip olduğu satır ve o satırdaki pozisyon değişkenlerini yok etmemiz gerekiyor. Aynı harf birden fazla yerde kullanıldığı için satır ve pozisyon değişkenleri anlamlarını yitiriyorlar. Bir harfin nerede kullanıldığını bilmek için HarfManager isminde yeni bir sınıf programlıyoruz. HarfManager bünyesinde harfArrray2D isminde iki boyutlu bir Harf array tanımlıyoruz. Bu oluşan dökümanın satırlarını ihtiva eden bir yapı olacaktır. Birinci boyut satırları, ikinci boyut her satırın belirli bir pozisyonunu adreslemek için kullanılmaktadir. Örneğin 5. satırın 16. pozisyonunda L harfini

kullanmak istiyorsak:

```
harfArray2D[5][16] = Harf.getHarf("L");
```

şeklinde bir tanımlama yapabiliriz.

```
// Kod 44

package com.pratikprogramci.designpatterns.bolum6.flyweight;

import java.util.ArrayList;

/**
 * Flyweight tasarım şablonunu kullanarak, kullanılan harf nesne adedi
 * alfabedeki harf adedine indirilir.
 *
 */
public class HarfManager {
    /**
     * Singleton değişken
     */
    private static final HarfManager manager = new HarfManager();

    /**
     * Harf nesnelerinin yer aldığı liste
     */
    private ArrayList<Harf> harfList = new ArrayList<Harf>();

    /**
     * Doküman satırlarının ve satırlarda kullanılan harf pozisyonlarinin
     * tutulduğu iki boyutlu array.
     */
    private Harf[][] document = new Harf[50][50];

    /**
     * Singleton olduğu için konstrüktörü private olarak deklare ediyoruz.
     */
    private HarfManager() {

    }

    public static final HarfManager instance() {
        return manager;
    }

    /**
```

```
     * Bir harf nesnesi oluşturur. Mevcut harf nesneleri harfList degiskeninde
     * tutulur. Bu listede yer almayan harfler new ile oluşturularak listeye
     * dahil edilirler.
     *
     * @param h
     *            Harf
     * @return Harf harf nesnesi
     */
    public static Harf getHarf(final String h) {
        Harf harf = null;
        for (int i = 0; i < manager.harfList.size(); i++) {
            final Harf temp = manager.harfList.get(i);

            if (temp.getHarf().equals(h)) {
                harf = temp;
                break;
            }
        }

        // harf bulunamadı, listeye ekliyoruz.
        if (harf == null) {
            harf = new Harf(h);
            manager.harfList.add(harf);
        }
        return harf;
    }

    /**
     * Dökümana bir satır ekler.
     *
     * @param satır
     *            Doküman satırı
     * @param satirNo
     *            Dökümandaki satır no
     */
    public void addSatir(final String satır, final int satirNo) {
        /**
         * Bir döngü içinde satırın tüm harflerini ziyaret ederek, dökümanın
         * satırını harf nesneleri ile oluşturuyoruz.
         */
        for (int i = 0; i < satır.length(); i++) {
            final String harf = satır.substring(i, i + 1);
            document[satirNo][i] = getHarf(harf);
        }
    }

    /**
```

```
     * Doküman ekranda görüntülenir.
     */
    public static void getDokument() {
        for (int x = 0; x < 50; x++) {
            for (int y = 0; y < 50; y++) {
                if (manager.document[x][y] != null) {
                    System.out.print((manager.document[x][y]).getHarf());
                } else {
                    break;
                }
            }
            if (manager.document[x][0] != null) {
                System.out.println("");
            }
        }

        System.out.println("Oluşturulan harf nesnesi adedi: " + manager.harfList.size());
    }

    public ArrayList<Harf> getHarfList() {
        return harfList;
    }

    public void setHarfList(final ArrayList<Harf> harfList) {
        this.harfList = harfList;
    }

    public Harf[][] getDocument() {
        return document;
    }

    public void setDocument(final Harf[][] document) {
        this.document = document;
    }
}
```

HarfManager sınıfını bir singleton olarak tanımlıyoruz. Statik olarak tanımlanmış olan getHarf() metodu ile harf nesneleri oluşturulmaktadır. Harf nesneleri oluşturulduktan sonra harfList isimli listede tutulur. addSatir() metodu bünyesinde dökümana bir satır eklenir. satirNo döküman içinde hangi satırın kullanılacağına işaret eder.

```
// Kod 45

package com.pratikprogramci.designpatterns.bolum6.flyweight;
```

```
/**
 * Test programı
 */
public class Test {

    public static void main(final String[] args) {
        /**
         * Dökümanın ilk satırı
         */
        final String ilkSatir = "Bu ilk satirdir";

        /**
         * Dökümanın ikinci satırı
         */
        final String ikinciSatir = "Bu ikinci satirdir";

        /**
         * Dökümana ilk satırı ekliyoruz
         */
        HarfManager.instance().addSatir(ilkSatir, 1);

        /**
         * Dökümana ikinci satırı ekliyoruz.
         */
        HarfManager.instance().addSatir(ikinciSatir, 2);

        HarfManager.getDokument();
    }
}
```

Test programı ile ekran çıktısı şöyle olacaktır:

```
Bu ilk satirdir
Bu ikinci satirdir
Oluşturulan harf nesnesi adedi: 13
```

ilkSatir ve ikinciSatir değişkenleri ile dökümanın birinci ve ikinci satırları tanımlanır. HarfManager sınıfının sahip olduğu addSatir() metodu ile bu satırlar HarfManager sınıfı bünyesinde tanımlanmış olan dökümana (document değişkeni) eklenir. HarfManager.getDokument() metodu ile iki boyutlu array (harfArray2D değişkeni) içinde yer alan satırlar ekranda görüntülenir. İki satır için toplam 29 harf kullanıldı. harfList listesinde 13 harf bulunmaktadır. Örneğin i harfi 8 değişik yerde kullanılmıştır. Flyweight tasarım şablonu uygulandığı için 8 i harfi nesnesi oluşturmak yerine, sadece 1 kere oluşturarak, 8 değişik yerde kullanmış olduk.

Sinek siklet tasarım şablonu ne zaman kullanılır?

Uygulama için kullanılan nesne adedini önemli ölçüde düşürmek için sinek siklet tasarım şablonu kullanılır. Nesne adedinin düşmesi ile performans iyileşir ve daha az hafıza alanı kullanılır.

İlişkili tasarım şablonları

Durum (state) ve strateji (strategy) tasarım şablonlarında kullanılan nesnelerin sinek siklet olarak implemente edilmelerinde fayda vardır. Bu şekilde bu tür nesnelerin ortak kullanımı mümkün hale gelir.

7. Bölüm

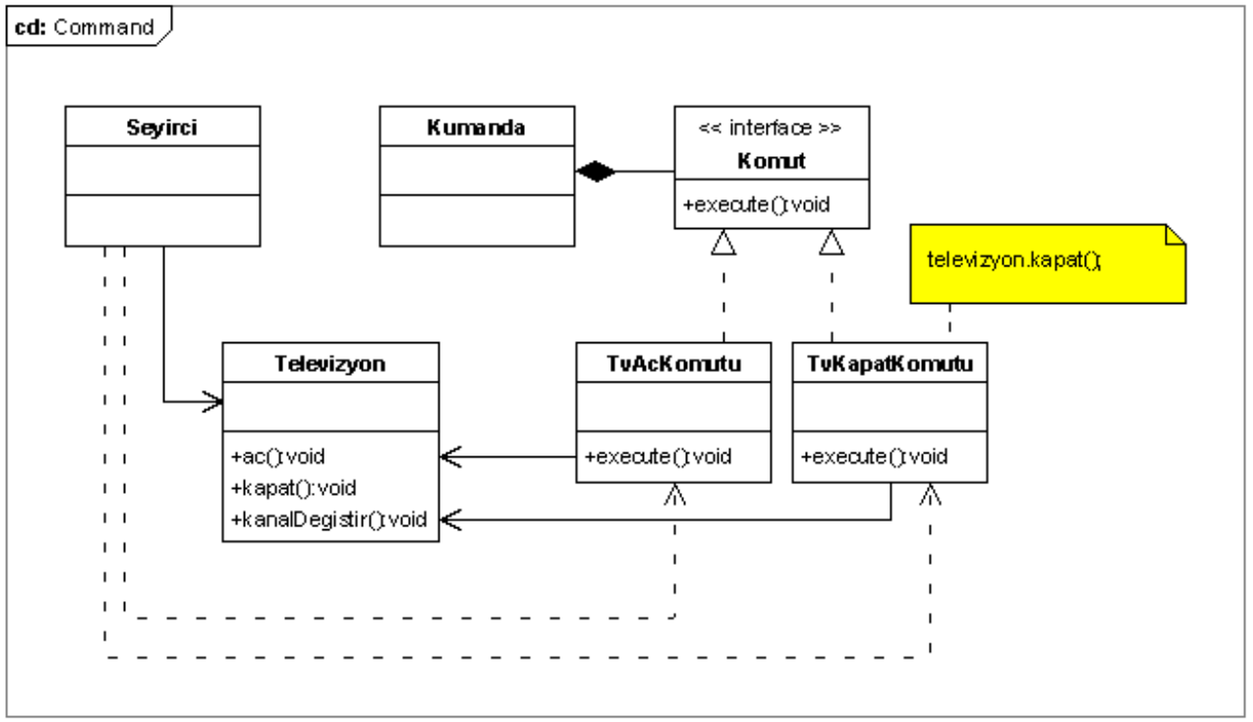
Davranışsal Tasarım Şablonları

Komut (Command Pattern)

Komut tasarım şablonunu açıklamak için televizyonu uzaktan yönetme aletini metafor olarak kullanmak istiyorum. Kanalları değiştirmek için yönetme aleti üzerinde belirli bir tuşa basarız. Tuşa basıldığı anda yönetme aleti televizyona bir komut göndererek, kanalın değişmesini sağlar. Aynı şekilde televizyonu açıp, kapatmak ve ses ve renk ayarlarını yapmak için değişik tuşlar kullanılır.

Kullanıcı olarak tuşa basıldığında televizyon bünyesinde ne gibi bir işlemin yapıldığı hakkında bilgi sahibi olmamız gerekmiyor. Bunu alıcının (televizyon) bilmesi yeterlidir. Bilmemiz gereken tek şey, hangi tuşun altında hangi komutun olduğudur.

Bir nesne üzerinde bir işleminin nasıl yapıldığını bilmediğimiz ya da kullanılmak istenen nesneyi tanımadığımız durumlarda, komut tasarım şablonu ile yapılmak istenen işlemi bir nesneye dönüştürerek, alıcı nesne tarafından işlemin yerine getirilmesi sağlıyabiliriz.



Resim 1

UML diyagramında görüldüğü gibi televizyonu açıp, kapatma işlemleri için TvAcKomutu ve TvKapatKomut isimlerinde iki sınıf tanımlıyoruz. Bu komutlar Komut interface sınıfını implemente ederek, birer Komut nesnesi haline geliyorlar. Komut interface sınıfında execute() isminde bir metot tanımlıyoruz.

```
// Kod 1

package com.pratikprogramci.designpatterns.bolum7.command;

/**
 * Komut interface sınıfı.
 *
 */
public interface Komut {

    /**
     * Alt sınıflar execute() metodunu implemente ederler.
     */
    void execute();
}
```

Gerçek bir komut sınıfı, Komut interface sınıfını implemente ettiği için execute() metoduna sahip olmak zorundadır. Bu metot bünyesinde yapılmak istenen işlem (örneğin televizyonu aç) alıcı nesneye (televizyon) delege edilir.

```
// Kod 2

package com.pratikprogramci.designpatterns.bolum7.command;

/**
 * Televizyonu açmak için kullanılan komut.
 *
 */
public class TvAcKomutu implements Komut {

    private Televizyon tv = null;

    public TvAcKomutu(final Televizyon tv) {
        this.tv = tv;
    }

    @Override
    public void execute() {
        tv.ac();
    }
}
```

TvAcKomutu sınıfı bünyesinde görüldüğü gibi Televizyon isminde bir sınıf değişkeni tanımlayarak, execute() metodu bünyesinde bu nesnenin ac() metodunu koşturuyoruz. Analog olarak

TvKapatKomutu sınıfı sahip olduğu execute() metodu bünyesinde tv.kapat() metodunu kullanır.

```
// Kod 3

package com.pratikprogramci.designpatterns.bolum7.command;

/**
 * Televizyonu kapatmak için kullanılan komut.
 *
 */
public class TvKapatKomutu implements Komut {

    private Televizyon tv = null;

    public TvKapatKomutu(final Televizyon tv) {
        this.tv = tv;
    }

    @Override
    public void execute() {
        tv.kapat();
    }
}
```

Televizyonu açıp kapatma işlemlerini yerine getirebilmek için bir televizyon yönetme (uzaktan kumanda) aletine ihtiyacımız var. İlk iki tuşu yukarıda yer alan komutlar ile programlayarak, televizyon nesnesinin metotlarını kullanacağız.

```
// Kod 4

package com.pratikprogramci.designpatterns.bolum7.command;

/**
 * Bir tv kumanda aleti
 *
 */
public class Kumanda {

    public Komut[] tus = new Komut[2];

    public Kumanda() {
        final Televizyon tv = new Televizyon();
        tus[0] = new TvAcKomutu(tv);
        tus[1] = new TvKapatKomutu(tv);
    }
```

```
    public void tusla(final int i) {
        if (i > tus.length || i < 0) {
            throw new RuntimeException("" + "Tus gecersiz!");
        }
        tus[i].execute();
    }
}
```

Kumanda sınıfında Komut interface sınıfından olan nesneleri barındırmak üzere bir Array tanımlıyoruz. Bu array kumanda aletinin sahip olduğu tuşları temsil etmektedir. Yukarıda yer alan örnekte sahip olduğumuz komundanın iki tuşu mevcuttur. Sınıf konströktörü bünyesinde bir televizyon nesnesi oluşturuyoruz. Bu nesneyi TvAcKomutu ve TvKapatKomutu sınıflarından birer nesne oluşturmak için konstruktör parametresi olarak kullanıyoruz. Örnekte görüldüğü gibi komut nesneleri, yapılmak istenen operasyonu bünyesinde barındıran nesneleri tanırlar (TvAcKomutu Televizyon nesnesini tanıyor) . Aksı taktirde TvKapatKomutu ya da TvAcKomutu ne yapmaları gerektiğini bilemezler ve bizim istediğimiz işlemi yerine getiremezler.

Kumanda aletinin sıfırıncı ve birinci tuşlarına basmak için aşağıda yer alan Test sınıfını kullanıyoruz.

```
// Kod 5

package com.pratikprogramci.designpatterns.bolum7.command;

/**
 * Test programı
 *
 */
public class Test {

    public static void main(final String[] args) {
        // Kumanda aletini oluşturur
        final Kumanda kumanda = new Kumanda();

        // sıfır nolu tusa basar
        kumanda.tusla(0); // televizyonu acar

        // bir nolu tusa basar
        kumanda.tusla(1); // televizyonu kapatir
    }
}
```

New operatörü ile yeni bir Kumanda nesnesi oluşturduktan sonra, tusla(0) ve tusla(1) metodları ile

sıfırıncı ve birinci tuşlara basıyoruz. Ekran çıktısı aşağıdaki şekilde olacaktır:

```
Televizyon açıldı.
Televizyon kapandi
```

Test sınıfı, Televizyon sınıfını ve sahip olduğu ac() ve kapat() metotlarını tanımadan, televizyonu uzaktan kontrol edebilmektedir. Bu komut tasarım şablonu sayesinde gerçekleşmiştir.

Komut tasarım şablonu ne zaman kullanılır?

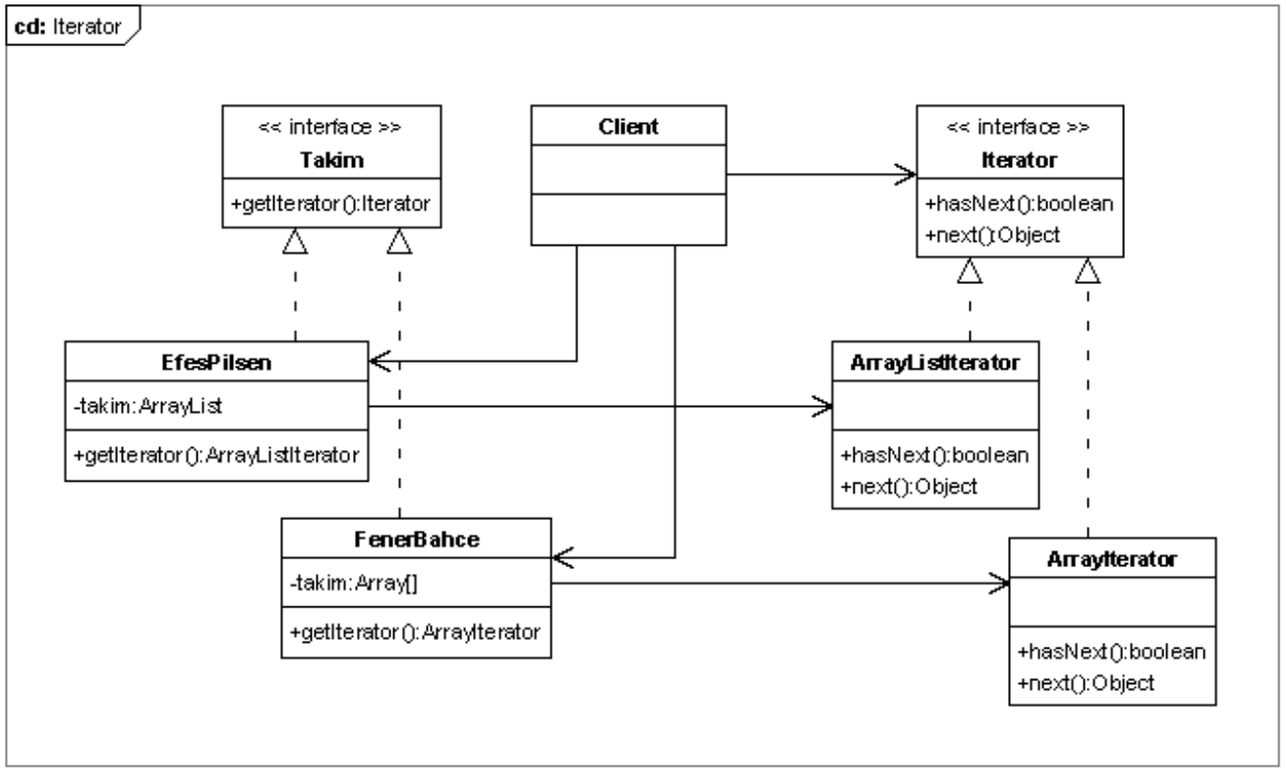
Yapmak istediğimiz işlemi ya da işlemin ait olduğu nesneyi tanımadığımız durumlarda komut tasarım şablonu kullanılır. Bir komut haline getirilen işlem, alıcı nesneye iletilerek, işlem gerçekleştirilir. Böylece komutu gönderen nesne ile işlemi gerçekleştirien nesne arasında esnek bir bağ oluşturulur. İki nesne birbirini tanımak zorunda değildir.

İlişkili tasarım şablonları

Komut tasarım şablonu memento nesneleri kullanarak, yapılan işlemlerin tekrar geri alınmasında sağlayabilir. Chain of responsibility tasarım şablonu command nesnelerini istek (request) nesnesi olarak kullanabilir.

Döngücü (Iterator Pattern)

Döngücü tasarım şablonu ile bir listede yer alan nesnelere sırayla, listenin yapısını ve çalışma tarzını bilmek zorunluluğumuz olmadan erişebilir ve bu nesneler üzerinde işlem yapabiliriz.



Resim 2

UML diyagramında Takim interface sınıfını implemente eden iki basketbol takımı yer almaktadır: Efes Pilsen ve Fenerbahse Basketbol takımlari. Ligde oynayan her takım için Takim interface sınıfını implemente eden bir sınıf oluşturulabilir.

Takım içinde yer alan oyuncuları tutmak için her sınıf kendi bünyesinde bir liste oluşturabilir. Takim interface sınıfı, bu listenin yapısı hakkında bir zorunluluk getirmediği için her takım kendi listesini istediği yapıda oluşturabilir. EfesPilsen sınıfını incelelediğimiz zaman, oyuncuların bir ArrayList içinde tutulduğunu görecegiz.

```
// Kod 6

package com.pratikprogramci.designpatterns.bolum7.iterator;

import java.util.ArrayList;

/**
 * Efes Pilsen Basketbol Takımı
 *
 */
public class EfesPilsen implements Takim {
```

```
    private ArrayList<Oyuncu> takım;

    public EfesPilsen() {
        // İsimler ve numaralar Efes Pilsen
        // takımı resmi sitesinden alinmistir
        // http://www.efesbasket.org
        takım = new ArrayList();
        takım.add(new Oyuncu("Drew", 4));
        takım.add(new Oyuncu("Ender", 6));
        takım.add(new Oyuncu("Cenk", 7));
        takım.add(new Oyuncu("Kerem", 12));
        takım.add(new Oyuncu("Loren", 15));
    }

    public ArrayList<Oyuncu> getTakim() {
        return takım;
    }

    public void setTakim(final ArrayList<Oyuncu> takım) {
        this.takım = takım;
    }

    @Override
    public Iterator getIterator() {
        return new ArrayListIterator(getTakim());
    }
}
```

FenerBahce sınıfı oyuncularını bir Array[] içinde tutmaktadır:

```
// Kod 7

package com.pratikprogramci.designpatterns.bolum7.iterator;

/**
 * Fenerbahce Ülker Basketbol Takımı
 *
 */
public class FenerBahce implements Takim {
    private Oyuncu[] takım = new Oyuncu[5];

    public FenerBahce() {
        // İsimler ve numaralar Fenerbahce Ülker
        // takımı resmi sitesinden alinmistir
        // http://www.fenerbahce.org/kurumsal/
        // kategori.asp?ContentCategoryID=38
```

```
        takım[0] = new Oyuncu("Ismail", 4);
        takım[1] = new Oyuncu("Willie", 5);
        takım[2] = new Oyuncu("Semih", 9);
        takım[3] = new Oyuncu("Ibrahim", 10);
        takım[4] = new Oyuncu("Serhat", 33);
    }

    public Oyuncu[] getTakim() {
        return takım;
    }

    public void setTakim(final Oyuncu[] takım) {
        this.takım = takım;
    }

    @Override
    public Iterator getIterator() {
        return new ArrayIterator(getTakim());
    }
}
```

Görüldüğü üzere iki takım oyuncuları iki değişik tip listede yer aldı. İki takımın sahip olduğu oyuncu isimlerini ekranda görüntülemek için aşağıdaki sınıf kullanılabilir:

```
// Kod 8

package com.pratikprogramci.designpatterns.bolum7.iterator;

/**
 * Test sınıfı
 *
 */
public class Test {

    public static void main(final String[] args) {
        final EfesPilsen efesPilsen = new EfesPilsen();
        Iterator it = efesPilsen.getIterator();

        while (it.hasNext()) {
            final Oyuncu oyuncu = (Oyuncu) it.next();
            System.out.println(oyuncu.getIsim());
        }

        final FenerBahce fenerBahce = new FenerBahce();
        it = fenerBahce.getIterator();
```

```
        while (it.hasNext()) {
            final Oyuncu oyuncu = (Oyuncu) it.next();
            System.out.println(oyuncu.getIsim());
        }
    }
}
```

İki takımın oyuncularını ekranda görünteleyebilmek için her takımın listesini edinip, o listenin gereksinimlerini göz önünde bulundurarak işlem yapmamız gerekiyor. Bu sebepten dolayı Test sınıfı hem komplike bir yapı alıyor hem de takımların oyuncularını nasıl ve hangi bir tip listede tuttuklarını bilmemiz gerekiyor. Bu bakımı çok zor bir Test sınıfının oluşması için yeterli iki sebeptir. Ligte onlarca takımın yer aldığını ve her takımın değişik bir oyuncu listesi olduğunu düşünürsek, Test sınıfının yapısının ne kadar daha da zorlaşacağını hemen görebiliriz.

Döngücü tasarım şablonunu kullanarak, Test sınıfının yapısını çok daha basit bir hale getirebiliriz. Amacımız takımlar tarafından kullanılan liste yapısını gizlemek ve kullanıcı sınıfların (örneğin Test) tanımlanmış interface metotları üzerinden değişik tipteki listeler üzerinde işlem yapmalarını sağlamaktır. Takımlar ne tip bir liste kullanırlarsa kullansınlar, kendilerine uygun bir iterator nesnesine sahip oldukları sürece, kullanıcı sınıflar tarafından oyuncu listeleri Iterator interface metodları kullanılarak işlenebilir hale gelecektir. Böylece kullanıcı sınıf takımların kullandıkları listelerden bağımsız bir hale gelebilir ve her türlü takım listesini istediğimiz şekilde işleyebiliriz.

Döngücü tasarım şablonunu uygulayabilmek için önce Takim isminde bir interface sınıf tanımlıyoruz:

```
// Kod 9

package com.pratikprogramci.designpatterns.bolum7.iterator;

/**
 * Takim interface sınıfı
 *
 */
public interface Takim {

    Iterator getIterator();
}
```

Takim interface sınıfı bünyesinde getIterator() isminde, kullanılan takıma göre uyumlu bir iterator nesnesi geri veren bir metot tanımlıyoruz.

EfesPilsen ve FenerBahce takımları, Takim interface sınıfını implemente ederek, dış dünyaya, sahip oldukları takım listesi üzerinde işlem yapmayı kolaylaştıracak bir iterator nesnesi sunarlar. Değişik tipte listeler üzerinde işlem yapabilecek şekilde Iterator sınıfları oluşturmadan önce, listeler üzerinde uygulanmak üzere kullanılacak Iterator interface ve metotları tanımlıyoruz:

```
// Kod 10

package com.pratikprogramci.designpatterns.bolum7.iterator;

/**
 * Iterator interface sınıfı
 *
 */
public interface Iterator {

    boolean hasNext();

    Object next();
}
```

hasNext() metodu ile bir takımın sahip olduğu listenin bir sonraki pozisyonunda oyuncu olup, olmadığı tespit edilir. Eğer liste sonuna gelinmişse, false değeri geri verilerek, listede oyuncu kalmadığı sinyali verilir. next() metodu ile listenin bir sonraki elementi edinilir.

Basketbol takımlarını Takim inteface sınıfını implemente edecek şekilde değiştiriyoruz:

```
// Kod 11

package com.pratikprogramci.designpatterns.bolum7.iterator;

import java.util.ArrayList;

/**
 * Efes Pilsen Basketbol Takımı
 *
 */
public class EfesPilsen implements Takim {

    private ArrayList<Oyuncu> takım;

    public EfesPilsen() {
        // İsimler ve numaralar Efes Pilsen
        // takımı resmi sitesinden alinmistir
        // http://www.efesbasket.org
```

```
        takım = new ArrayList();
        takım.add(new Oyuncu("Drew", 4));
        takım.add(new Oyuncu("Ender", 6));
        takım.add(new Oyuncu("Cenk", 7));
        takım.add(new Oyuncu("Kerem", 12));
        takım.add(new Oyuncu("Loren", 15));
    }

    public ArrayList<Oyuncu> getTakim() {
        return takım;
    }

    public void setTakim(final ArrayList<Oyuncu> takım) {
        this.takım = takım;
    }

    @Override
    public Iterator getIterator() {
        return new ArrayListIterator(getTakim());
    }
}
```

EfesPilsen sınıfı, Takim interface sınıfını implemente ettiği için getIterator() isminde yeni bir metoda sahip oluyor. Bu metot, EfesPilsen sınıfının sahip olduğu takım listesi üzerinde işlem yapmayı kolaylaştıracak olan ArrayListIterator sınıfından bir nesne oluşturarak, kullanıcı sınıfa verir. ArrayListIterator sınıfı aşağıdakı yapıya sahiptir:

```
// Kod 12

package com.pratikprogramci.designpatterns.bolum7.iterator;

/**
 * ArrayIterator sınıfı
 *
 */
public class ArrayIterator implements Iterator {

    private Oyuncu[] takım;
    private int pozisyon;

    public int getPozisyon() {
        return pozisyon;
    }

    public void setPozisyon(final int pozisyon) {
```

```
        this.pozisyon = pozisyon;
    }

    public ArrayIterator(final Oyuncu[] takım) {
        setTakim(takım);
    }

    @Override
    public boolean hasNext() {
        if (pozisyon >= getTakim().length ||
                getTakim()[pozisyon] == null) {
            return false;
        } else {
            return true;
        }
    }

    @Override
    public Object next() {
        final Oyuncu oyuncu = getTakim()[pozisyon];
        pozisyon++;
        return oyuncu;
    }

    public Oyuncu[] getTakim() {
        return takım;
    }

    public void setTakim(final Oyuncu[] takım) {
        this.takım = takım;
    }
}
```

Bu sınıf bünyesinde dikkatimizi çeken hasNext() ve next() metotlarının implementasyonudur. EfesPilsen sınıfı takım oyuncularını tutmak için bir ArrayList kullandığı için ArrayListIterator sınıfı, hasNext() ve next() metotlarını ArrayList tipi bir listede işlem yapabilecek şekilde implemente etmiştir.

EfesPilsen sınıfı üzerinde uyguladığımız değişiklikleri FenerBahce sınıfına da uyguluyoruz.

```
// Kod 13

package com.pratikprogramci.designpatterns.bolum7.iterator;

/**
```

```
 * Fenerbahce Ülker Basketbol Takımı
 *
 */
public class FenerBahce implements Takim {
    private Oyuncu[] takım = new Oyuncu[5];

    public FenerBahce() {
        // İsimler ve numaralar Fenerbahce Ülker
        // takımı resmi sitesinden alinmistir
        // http://www.fenerbahce.org/kurumsal/
        // kategori.asp?ContentCategoryID=38

        takım[0] = new Oyuncu("Ismail", 4);
        takım[1] = new Oyuncu("Willie", 5);
        takım[2] = new Oyuncu("Semih", 9);
        takım[3] = new Oyuncu("Ibrahim", 10);
        takım[4] = new Oyuncu("Serhat", 33);
    }

    public Oyuncu[] getTakim() {
        return takım;
    }

    public void setTakim(final Oyuncu[] takım) {
        this.takım = takım;
    }

    @Override
    public Iterator getIterator() {
        return new ArrayIterator(getTakim());
    }
}
```

FenerBahce takımı oyuncuların yer aldığı Array (Oyuncu[]) tipi bir liste kullanmaktadır. getIterator() metodunda Array üzerinde işlem yapmasını bilen ArrayIterator sınıfından bir nesne oluşturulur. ArrayIterator sınıfı aşağıdakı yapıya sahiptir:

```
// Kod 14

package com.pratikprogramci.designpatterns.bolum7.iterator;

/**
 * ArrayIterator sınıfı
 *
 */
public class ArrayIterator implements Iterator {
```

```
    private Oyuncu[] takım;
    private int pozisyon;

    public int getPozisyon() {
        return pozisyon;
    }

    public void setPozisyon(final int pozisyon) {
        this.pozisyon = pozisyon;
    }

    public ArrayIterator(final Oyuncu[] takım) {
        setTakim(takım);
    }

    @Override
    public boolean hasNext() {
        if (pozisyon >= getTakim().length ||
                getTakim()[pozisyon] == null) {
            return false;
        } else {
            return true;
        }
    }

    @Override
    public Object next() {
        final Oyuncu oyuncu = getTakim()[pozisyon];
        pozisyon++;
        return oyuncu;
    }

    public Oyuncu[] getTakim() {
        return takım;
    }

    public void setTakim(final Oyuncu[] takım) {
        this.takım = takım;
    }
}
```

ArrayIterator sınıfı bir Array üzerinde işlem yapmak üzere oluşturulduğu için hasNext() ve next() metotları bir Array listesi üzerinde işlem yapacak şekilde implemente edilmiştir.

Bu değişikliklerin ardından, takımların sahip oldukları liste tiplerini bilme zorunluluğunu kullanıcı

sınıftan (Test) alarak, Iterator interface sınıfını implemente eden sınıflara aktarmış olduk. Kullanıcı sınıflar tarafından kullanılmak üzere her liste tipi için geçerli metotları Iterator interface sınıfında tanımlıyarak, listeler üzerinde yapılan işlemler genel bir hale almış oluyor. Bu sayede kullanıcı sınıflar, takımlar tarafından kullanılan liste tipinden bağımsız bir hale geldi, yani kullanıcı sınıf bir takımın kullandığı liste tipini bilmek zorunda değil. Bu böyle olunca, yapılan işlemler ve kullanıcı sınıfın yapısı daha kolay bir hal alır. Test sınıfının yeni halı aşağıda yer almaktadır:

```
// Kod 15

package com.pratikprogramci.designpatterns.bolum7.iterator;

/**
 * Test sınıfı
 *
 */
public class Test {

    public static void main(final String[] args) {
        final EfesPilsen efesPilsen = new EfesPilsen();
        Iterator it = efesPilsen.getIterator();

        while (it.hasNext()) {
            final Oyuncu oyuncu = (Oyuncu) it.next();
            System.out.println(oyuncu.getIsim());
        }

        final FenerBahce fenerBahce = new FenerBahce();
        it = fenerBahce.getIterator();

        while (it.hasNext()) {
            final Oyuncu oyuncu = (Oyuncu) it.next();
            System.out.println(oyuncu.getIsim());
        }
    }
}
```

Test sınıfı her takımdan bir iterator nesnesi edinerek, hasNext() ve next() metotları ile o takımın sahip olduğu liste üzerinde işlem yapabilir. Bunun için listenin hangi tipte olması gerektiğini bilmek zorunda değildir. Böylece Test sınıfı bakımı daha kolay bir hale gelmektedir.

Döngücü tasarım şablonu ne zaman kullanılır?

Sınıflar bünyelerinde başka nesneleri barındırmak için değişik tipte listelere sahip olabilirler. Bu

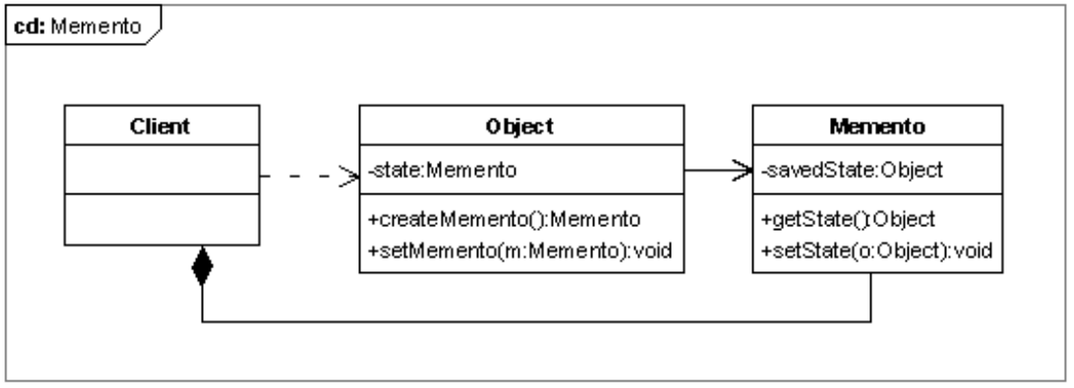
sınıfların nasıl implemente edildiği saklamak ve sahip oldukları listeler üzerinde işlem yapmayı kolaylaştırmak için döngücü tasarım şablonu kullanılır.

İlişkili tasarım şablonları

Fabrika tasarım şablonu kullanılarak uyumlu bir iterator altsınıfı oluşturulabilir. Hatırlayan (memento) tasarım şablonu çogu zaman döngücü tasarım şablonu ile beraber kullanılır. İterator bir memento nesnesini kullanarak, sahip olduğu değeri saklayabilir (hafızada tutabilir). Bunun için iterator memento nesnesini sınıf değişkeni olarak tanımlar. İteratörler kompozit (composite) nesnelerde olduğu gibi çogu zaman rekursiv yapılarda kullanılır.

Hatırlayan (Memento Pattern)

Bir nesneyi, daha önce sahip olduğu bir duruma tekrar dönüştürebilmek için memento tasarım şablonu kullanılır. Örneğin bir çizim programında üçgen nesneleri kullandığımızı düşünelim. Bu üçgenlerin boyutlarını değistirebiliriz, yani üçgenler küçültülebilirler ya da büyütülebilirler. Çizim programı "değişikliği geri al" (Undo) şeklinde bir menüye sahiptir. Bu menü üzerinden üçgenler üzerinde yaptığımız her türlü değişikliği geri alabilir ve üçgenleri eski hallerine dönüştürebiliriz. Yapılan değişiklikleri geri alabilmemiz için değişikliği yapmadan önce üçgen nesnelerinin son durumlarını saklamamız gerekmektedir. Memento tasarım şablonunu kullanarak nesnenin sahip olduğu son durumu kaydedip, gerekli durumlarda nesneyi eski haline dönüştürebiliriz.



Resim 3

UML diyagramında yer alan Client sınıfını çizim programı olarak düşünürsek, çizilen üçgenlerin en son durumları memento nesnesi olarak çizim programının bünyesinde barındırılır. Kullanıcı tarafından yapılan değişiklikler geri alındığı zaman, çizim programı sahip olduğu memento nesneleri yardımı ile üçgenleri eski haline getirebilir. Client sınıfı hiçbir zaman memento nesneleri üzerinde değişiklik yapmaz.

UML diyagramında yer alan Object sınıfını çizim programında kullandığımız bir üçgen element olarak düşünebiliriz. Object sınıfından olan bir nesne, sahip olduğu son durumu bir memento nesne oluşturarak, saklanabilir hale getirebilir. Aynı şekilde Object sınıfından bir nesne bir memento nesnesini kullanarak, daha önce sahip olduğu bir duruma geri dönebilir.

Memento sınıfından oluşturulan nesneler, Object sınıfından olan nesnelerin sahip oldukları değerleri bünyelerinde barındırdıkları için bu değerleri korumakla yükümlüdürler. Object sınıfı harici hiçbir sınıf bir Memento nesnesi bünyesinde bulunan değerleri görme ya da değiştirme hakkına sahip değildir.

Memento tasarım şablonunu nasıl uygulayabileceğimizi aşağıda yer alan örnekte inceliyelim.

```
// Kod 16

package com.pratikprogramci.designpatterns.bolum7.memento;

import java.util.ArrayList;

/**
 * Dokument sınıfı
 *
 */
public class Dokument {
    /**
     * Bir döküman içinde yer alan satırları satırlar
     * isimli ArrayList içinde tutuyoruz.
     */
    private ArrayList<String> satırlar =
            new ArrayList<String>();

    public void addSatir(final int index, final String satır) {
        getSatirlar().add(index, satır);
    }

    public void removeSatir(final int index) {
        getSatirlar().remove(index);
    }

    public Memento createMemento() {
        return new Memento(getSatirlar().toArray());
    }

    public void setMemento(final Memento memento) {
        getSatirlar().clear();
```

```
        final Object[] tempSatirlar = memento.getElements();
        for (int i = 0; i < tempSatirlar.length; i++) {
            final String satır = (String) tempSatirlar[i];
            getSatirlar().add(satır);
        }
    }

    @Override
    public String toString() {
        final StringBuilder temp = new StringBuilder();
        for (int i = 0; i < getSatirlar().size(); i++) {
            temp.append(getSatirlar().get(i)).append(" \n");
        }
        return temp.toString();
    }

    public ArrayList<String> getSatirlar() {
        return satırlar;
    }

    public void setSatirlar(final ArrayList<String> satırlar) {
        this.satırlar = satırlar;
    }

}
```

Dokument isminde bir sınıf oluşturuyoruz. Bu sınıf içinde bir dökümanın satırlarını tutabilmek için satırlar isminde ArrayList tipinde bir sınıf değişkeni tanımlıyoruz. addSatir() metodu ile dökümana bir satır ekliyebilir, removeSatir() ile dökümanın bir satırını silebiliriz.

createMemento() metodu içinde Dokument nesnesinin sahip olduğu son durum alınarak bir memento nesnesi oluşturulur. Bu andan itibaren memento nesnesi Dokument nesnesinin sahip olduğu satırları ihtiva eder.

setMemento() metodu ile mevcut bir memento nesnesi kullanılarak, Dokument nesnesinin sahip olduğu iç değerler değiştirilir. Bu işlemi yapmadan önce getSatirlar().clear() ile dökümanın sahip olduğu satırları siliyoruz. setMemento() metodunu kullanabilmemiz için daha önce createMemento() metodu ile bir memento nesnesi oluşturmuş olmamız gerekiyor.

```
// Kod 17

package com.pratikprogramci.designpatterns.bolum7.memento;

/**
```

```
 * Memento sınıfı. Dokument sınıfından bir nesnenin sahip
 * olduğu satırları bünyesinde barındırır.
 *
 */
public class Memento {

    private Object[] elements;

    public Memento(final Object[] elem) {
        elements = elem;
    }

    public Object[] getElements() {
        return elements;
    }

    public void setElements(final Object[] elements) {
        this.elements = elements;
    }
}
```

Memento sınıfı, Dokument nesnesinin sahip olduğu değerleri bünyesinde barındırmak zorundadır. Bu yüzden Object[] tipinde bir array tanımlıyoruz. Bu array daha sonra bir Dokument nesnesinin sahip olduğu satırları ihtiva edecektir.

```
// Kod 18

package com.pratikprogramci.designpatterns.bolum7.memento;

/**
 * Test sınıfı.
 *
 */
public class Test {
    public static void main(final String[] args) {
        final Dokument dokument = new Dokument();
        dokument.addSatir(0, "------------------");
        dokument.addSatir(1, "+                +");
        dokument.addSatir(2, "+      XXX       +");
        dokument.addSatir(3, "+                +");
        dokument.addSatir(4, "------------------");

        // Dökümanın son halini memento nesnesine
        // aktarıyoruz.
        final Memento memento = dokument.createMemento();
```

```
        System.out.println("Dökümanın ilk hali:");
        System.out.println(dokument.toString());

        // Doküman üzerinde değişiklik yapıyoruz.
        dokument.removeSatir(2);
        dokument.addSatir(2, "+      YYY       +");

        System.out.println("\n\nDokümanin yeni hali:");
        System.out.println(dokument.toString());

        // Dökümanı eski haline dönüstürüyoruz
        dokument.setMemento(memento);

        System.out.println("\n\nDokümanin en son hali:");
        System.out.println(dokument.toString());
    }
}
```

Kod 18 de yer alan Test sınıfı bünyesinde memento tasarım şablonu test etmek üzere bir Dokument nesnesi oluşturup, yeni satırlar ekliyoruz. Daha sonra createMemento() ile dökümanın en son halini bir memento nesnesine yerleştiriyoruz. Akabinde döküman üzerinde değişiklik yaparak, ekran çıktısı alıyoruz. En son işlem olarak değişikliği geri alarak, ilk oluşturulan döküman versiyonunu ekranda görüntülüyoruz.

Ekran çıktısı aşağıdaki şekilde olacaktır:

Dökümanın ilk hali:

```
------------------
+                +
+                +
+                +
+                +
+                +
+                +
+                +
------------------
```

Dökümanın yeni hali:

```
------------------
+                +
+      YYY       +
+                +
```

```
------------------
```

Dökümanın en son hali:

```
------------------
+                +
+      XXX       +
+                +
------------------
```

Java örneğinde gördüğümüz gibi memento tasarım şablonunu uygularak, bir nesneyi nasıl tekrar eski haline dönüştürebileceğimizi gördük.

Memento tasarım şablonu ne zaman kullanılır?

Bir nesnenin sahip olduğu son durumu saklamak için Memento tasarım şablonu kullanılır.

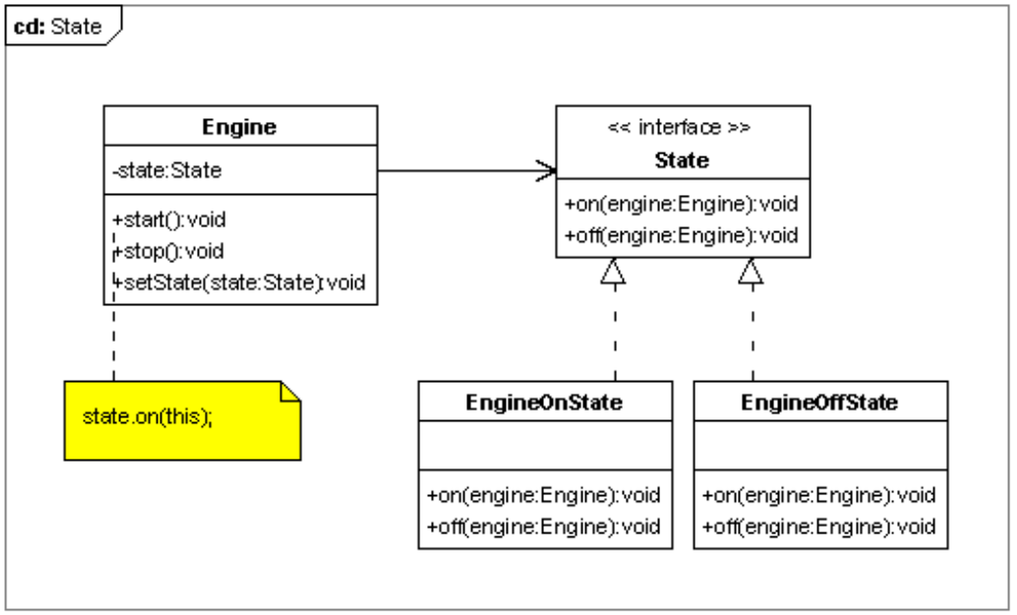
İlişkili tasarım şablonları

Komut tasarım şablonu memento nesneleri kullanarak, yapılan işlemlerin tekrar geri alınmasında kullanılabilir. İteratörlerin son durumunu saklamak için Memento nesneleri kullanılabilir.

Durum (State Pattern)

Bir nesne sahip olduğu duruma göre (değişkenlerin değerleri) davranışını (sahip olduğu metotlar) değiştirebilir mi? Normal şartlar altında bu sorunu cevabı hayır olurdu. Ama durum tasarım şablonu ile bu mümkün!

Durum tasarım şablonu kullanarak, bir nesnenin davranışı, sahip olduğu değerler değiştiği zaman değiştirilebilir. Bu durumda sanki nesne sahip olduğu sınıfı değiştirmiş gibi olacaktır.



Resim 4

UML diyagramında yer alan Engine sınıfı bir araba motorunu temsil ediyor. Bu sınıfın start() ve stop() metotları üzerinden moturu durdurabilir ve çalışıtırabiliriz. Engine sınıfında State tipinde bir sınıf değişkeni yer alıyor. Motorun açılıp, kapatılabilmesi için gerekli metotları State ismini taşıyan bir interface sınıfı bünyesine taşıyoruz. Motorun sahip olabileceği (kapalı, çalışır) her durum için State sınıfını implemente edecek bir altsınıf oluşturmamız gerekiyor. Engine sınıfının start() ve stop() metotları kullanıldığında, bunlar state nesnesine delege edilecektir. UML diyagramında durumu değiştiği zaman davranışı da değişen nesne Engine sınıfıdır. Hedefimiz motorun sahip olduğu duruma göre davranışını değiştirmektir.

```
// Kod 19

package com.pratikprogramci.designpatterns.bolum7.state;

/**
 * State sınıfı
 *
 */
public interface State {
    void on(Engine engine);

    void off(Engine engine);
}
```

State interface sınıfı motoru açıp, kapatmak için on() ve off() metotlarını tanımlar. Parametre

olarak Engine sınıfından bir nesne verildiğini görüyoruz. Bu davranışını değiştirmek istediğimiz nesnedir.

```
// Kod 20

package com.pratikprogramci.designpatterns.bolum7.state;

/**
 * Engine sınıfı.
 *
 */
public class Engine {
    private State state;

    public Engine() {
        // Motor ilk olarak
        // kapalı durumuna getirilir.
        setState(new EngineOffState());
        System.out.println("Motor kapalı.");
    }

    public void start() {
        getState().on(this);
    }

    public void stop() {
        getState().off(this);
    }

    public State getState() {
        return state;
    }

    public void setState(final State state) {
        this.state = state;
    }
}
```

Engine sınıfı bünyesinde State interface sınıfından bir değişken tanımlıyoruz. Bu değişkenin sahip olduğu değer değiştikçe, nesnenin davranışı değişecektir, çünkü start() ve stop() metotlarına bakıldığında, bu işlemler state nesnesine delege edilmekte yani başka bir nesne tarafından yerine getirilmektedir. Sınıf konstrüktöründe state değişkenin değeri new EngineOffState() olarak değiştiriliyor yani Engine nesnesi oluştuğunda motor kapalı durumdadır.

```
// Kod 21

package com.pratikprogramci.designpatterns.bolum7.state;

/**
 * EngineOffState
 *
 */
public class EngineOffState implements State {

    @Override
    public void off(final Engine engine) {
        // off konumunda olan bir
        // motorun tekrar off() metodu
        // kullanılarak kapatilmak
        // istenmesi anlamsız.
        System.out.println("Motor kapalı durumda!");
    }

    @Override
    public void on(final Engine engine) {
        engine.setState(new EngineOnState());
        System.out.println("Motor çalıştı...");
    }
}
```

EngineOffState State interface sınıfını implemente eder. Bu durumda on() ve off() metotlarını implemente etmek zorundadır. Off() metodu bünyesinde herhangi bir işlemin yapılmadığını görüyoruz. Bunun sebebi çalışmayan bir motor için off() metodunun tekrar kullanılarak motorun durdurulmasının anlamsız bir işlem olmasıdır, yani çalışmayan bir moturun durdurulması gereksizdir. On() metodu bünyesinde Engine nesnesinin state değişkeni değiştirilir. Motor çalışır duruma getirildiği için Engine nesnesinin state değişkenininde değişmesi gerekmektedir ki nesne start() ve stop() metotları kullanıldığında doğru davranış gösterebilsin, yani davranışını duruma göre adapte edebilsin.

```
// Kod 22

package com.pratikprogramci.designpatterns.bolum7.state;

/**
 * EngineOnState
 *
 */
public class EngineOnState implements State {
```

```
    @Override
    public void off(final Engine engine) {
        engine.setState(new EngineOffState());
        System.out.println("Motor durduruldu...");
    }

    @Override
    public void on(final Engine engine) {
        // on konumunda olan bir
        // motorun tekrar on() metodu
        // kullanılarak acilmak
        // istenmesi anlamsız.
        System.out.println("Motor çalışır durumda!");
    }
}
```

EngineOffState sınıfına paralel olarak EngineOnState sınıfını tanımlıyoruz. Bu sınıf State interface sınıfını implemente ettiği için kardeş sınıfı EngineOffState gibi on() ve off() metotlarına sahiptir. Off() metodunda motor kapatıldığı için Engine nesnesinin durumunun değiştirilmesi gerekmektedir. On() metodunda bir işlem yapılmadığını görüyoruz, çünkü çalışır durumda olan bir motorun tekrar on() ile herekete geçirilmesi gereksizdir.

```
// Kod 23

package com.pratikprogramci.designpatterns.bolum7.state;

/**
 * Test sınıfı
 *
 */
public class Test {

    public static void main(final String[] args) {
        final Engine engine = new Engine();
        engine.stop();
        engine.start();
        engine.start();
    }
}
```

Her program örneğimizde olduğu gibi kod 23 de yer alan Test sınıfı ile tasarım şablonu uygulamamızı test ediyoruz. Engine isminde bir nesne oluşturduktan sonra bu sınıfın stop() metodunu kullanıyoruz. Akabinde start() ile motoru çalıştırıyoruz. Aynı işlemi tekrarlıyoruz ve

ekran çıktısı aşağıdaki şekilde oluyor:

```
Motor kapalı.
Motor kapalı durumda!
Motor çalıştı...
Motor çalışır durumda!
```

Ekran çıktısından da anlaşıldığı gibi motor ilk etapta kapalıdır. Kapalı olan bir motoru tekrar stop() ile kapatmak istediğimiz zaman ikinci satırda yer alan mesajı alırız (Motor kapalı durumda!). Kullandığımız ilk start() metodu motoru çalıştırır. Çalışır durumda olan motor tekrar start() metodu ile çalıştırılamayacağı için en son satırdaki mesajı alırız (Motor çalışır durumda!).

Çok basit bir örnek olduğu için belki bu kadar zahmete girip, neden State tasarım şablonunu uyguladığımız sorusu aklınıza gelebilir. Örneğin çok basit olduğu doğrudur. Sonuçta motorun durumun kapalı/açık şeklinde bir değişkende tutabilir ve start(), stop() metotlarında bu değişkeni değerlendirerek motoru kapalıysa, çalıstırabilir, çalışıyor durumdaysa, durdurabiliriz. Kurumsal projelerde yapılar daha karmaşık olacağı için durum tasarım şablonuna gerek duyulabilir. Örneğimiz çok basit olsada bir nesnenin davranışını nasıl değiştirebileceğini görmüş olduk. Kolay bir şekilde yeni durum sınıfları tanımlıyabiliriz. Örneğin bir motorun tamirde olduğunu ve motorun açılıp, kapatılamıyacağını implemente etmiş EngineServiceState (motor serviste anlamında) bir sınıf oluşturabiliriz.

Durum tasarım şablonu ne zaman kullanılır?

Bir nesnenin davranışları (metot yapısı) sahip olduğu duruma bağlı olabilir. Nesnenin sahip olduğu durumun değişmesi halinde davranışının da değişmesi gerektigi durumlarda durum tasarım şablonu kullanılır.

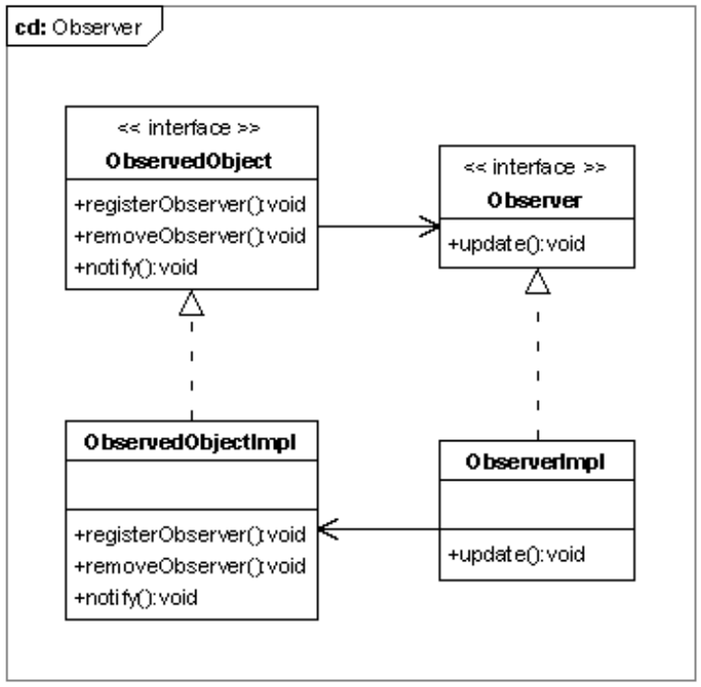
İlişkili tasarım şablonları

Durum nesneleri genelde singleton olarak implemente edilir. Durum nesneleri ortak kullanılıyorsa, flyweight tasarım şablonu uygulanmış olur.

Gözetleyici (Obverser Pattern)

Bir gazeteye günlük abone olmak istediğimizi düşünelim. Abone olduğumuz günden itibaren gazete her gün evimize posta aracılığıyla gönderilir. Burada gazete ile aboneleri arasında doğrudan bir ilişki oluşmaktadır. Hergün yeni bir baskı yapıldığı zaman aboneler otomatik olarak gazete tarafından bilgilendirilir. Bu bilgilendirme işlemi, gazetenin aboneye posta ile gönderilmesi

anlamına gelmektedir. Abone olduğumuz sürece her gün gazete bize gönderilir. İstediğimiz bir zaman gazeteye haber vererek, aboneliğimizi iptal ettirebiliriz. Abonelik sona erdikten sonra bize posta ile gazete gönderilmez. Gözetleyici tasarım şablonu bir abonelik sistemi gibi çalışmaktadır.



Resim 5

Sistem bünyesinde, bir nesnede meydana gelen değişikliklerden haberdar olmak isteyen diğer nesneler olabilir. Bu durumda haberdar olmak isteyen nesneler abone olarak, abone oldukları nesnede meydana gelen değişikliklerden haberdar edilirler. Abone olan nesne aboneliğini iptal ederek, abone olduğu nesne ile arasındaki ilişkiyi sonlandırabilir.

Gazete abonelik sistemini aşağıdaki şekilde gözetleyici tasarım şablonunu kullanarak implemente edebiliriz.

```
// Kod 24

package com.pratikprogramci.designpatterns.bolum7.observer;

/**
 * Gazete Inteface sınıfı
 *
 */
public interface Gazete {

    void aboneEkle(Abone abone);
```

```
    void aboneSil(Abone abone);

    void gazeteGonder();
}
```

Gazete interface sınıfında abonelik için gerekli aboneEkle() ve aboneSil() metotlarını tanımlıyoruz. Abonelere gazeteyi göndermek yani abone olan nesneleri bilgilendirmek için gazeteGonder() metodu bulunuyor. Bu interface sınıfını aşağıdaki şekilde implemente edebiliriz.

```
// Kod 25

package com.pratikprogramci.designpatterns.bolum7.observer;

import java.util.ArrayList;

/**
 * Hürriyet Gazetesi
 *
 */
public class Hurriyet implements Gazete {

    /**
     * Aboneleri tutmak için bir ArrayList tanımlıyoruz.
     */
    private ArrayList<Abone> aboneler = new ArrayList<Abone>();

    @Override
    public void aboneEkle(final Abone abone) {
        getAboneler().add(abone);
    }

    @Override
    public void aboneSil(final Abone abone) {
        getAboneler().remove(abone);
    }

    @Override
    public void gazeteGonder() {
        for (int i = 0; i < getAboneler().size(); i++) {
            getAboneler().get(i).update();
        }
    }

    public ArrayList<Abone> getAboneler() {
        return aboneler;
```

```
    }

    public void setAboneler(final ArrayList<Abone> aboneler) {
        this.aboneler = aboneler;
    }
}
```

Abone olan nesneleri tutmak için aboneler isminde bir ArrayList tanımlıyoruz. aboneEkle() ve aboneSil() bu liste üzerinde işlem yapıyorlar. gazeteGonder() metodu bünyesinde, aboneler listesinde yer alan tüm abone nesnelerin update() metodu kullanılarak bu nesneler haberdar ediliyor.

```
// Kod 26

package com.pratikprogramci.designpatterns.bolum7.observer;

/**
 * Abone interface
 *
 */
public interface Abone {

    void update();

    void aboneligiIptalEt();

    void aboneOl(Gazete gazete);
}
```

Abone olmak isteyen nesneler Abone interface sınıfını implemente etmek zorundadırlar. Abone olan nesneler aboneOl() metodunu kullanarak bir gazete nesnesine abone olabilirler ve aboneligiIptalEt() metodu ile aboneliği iptal edebilirler.

```
// Kod 27

package com.pratikprogramci.designpatterns.bolum7.observer;

/**
 * Müşteri sınıfı
 *
 */
public class Müşteri implements Abone {
    private Gazete gazete;
    private String isim;
```

```
    private String soyad;

    public Müşteri(final String isim, final String soyad) {
        setIsim(isim);
        setSoyad(soyad);
    }

    public String getIsim() {
        return isim;
    }

    public void setIsim(final String isim) {
        this.isim = isim;
    }

    public String getSoyad() {
        return soyad;
    }

    public void setSoyad(final String soyad) {
        this.soyad = soyad;
    }

    public Gazete getGazete() {
        return gazete;
    }

    public void setGazete(final Gazete gazete) {
        this.gazete = gazete;
    }

    @Override
    public void update() {
        System.out
                .println(getIsim() + " "
        + getSoyad() + " " + "gazeteyi aldi");
    }

    @Override
    public void aboneligiIptalEt() {
        getGazete().aboneSil(this);
        System.out.println(getIsim() + " " + getSoyad() + " "
                + "aboneligini sonlandirdi");
    }

    @Override
    public void aboneOl(final Gazete gazete) {
```

```
        setGazete(gazete);
        gazete.aboneEkle(this);
        System.out.println(getIsim() + " "
        + getSoyad() + " " + "abone oldu");
    }
}
```

Müşteri isminde Abone interface sınıfını implemente eden bir sınıf tanımlıyoruz. Abone interface sınıfında yer alan tüm metotlar bu sınıf tarafından implemente edilir. Abone olabilmek için aboneOl() metoduna bir gazete nesnesinin verilmesi gerekmektedir. Bu nesne kullanılarak abonelik işlemi (gazete.aboneEkle(this)) gerçekleştirilir. Abone olduğumuz gazete nesnesini kaybetmemek için gazete isimli bir sınıf değişkeninde tutuyoruz.

```
// Kod 28

package com.pratikprogramci.designpatterns.bolum7.observer;

/**
 * Test sınıfı.
 *
 */
public class Test {
    public static void main(final String[] args) {
        final Gazete hurriyet = new Hurriyet();

        final Abone musteri1 = new Müşteri("Tarik", "Akan");
        musteri1.aboneOl(hurriyet);

        final Abone musteri2 = new Müşteri("Filiz", "Akin");
        musteri2.aboneOl(hurriyet);

        hurriyet.gazeteGonder();

        musteri2.aboneligiIptalEt();

        hurriyet.gazeteGonder();
    }
}
```

Uygulamayı test etmek için Test.main() metodunu kullanıyoruz. İlk işlem olarak abone olunabilecek bir nesne (hurriyet) ve Müşteri sınıfını kullanarak abone olacak iki yeni müşteri nesnesi oluştuyoruz. Bu nesneler Abone.aboneOl() metodunu kullanarak, hurriyet nesnesine abone oluyolar. Bu andan itibaren bu iki nesne hurriyet nesnesinde meydana gelen değişikliklerden hurriyet.gazeteGonder() metodu aracılığıyla haberdar olacaklardır. Hurriyet nesnesi istediği zaman

gazeteGonder() metodunu kullanarak, abonelerine günlük gazeteyi gönderebilecektir. Aboneler istedikleri zaman sahip oldukları aboneliği iptal edebilirler. Bu işlem için aboneligiIptalEt() metodunu kullanmaları gerekiyor. Test.main() aşağıdaki ekran çıktısını verecektir:

```
Tarik Akan abone oldu
Filiz Akin abone oldu
Tarik Akan gazeteyi aldi
Filiz Akin gazeteyi aldi
Filiz Akin aboneligini sonlandirdi
Tarik Akan gazeteyi aldi
```

Gözetleyici tasarım şablonu ile sistemde bulunan nesneler arasında esnek bir bağ kurarak, bu nesnelerin birbirlerine çok fazla bağımlı olmadan nasıl beraber kullanılabileceklerini gördük. Birinci bölümde tasarım prensipleri ile tanışmıştık. Bunlardan birisi "nesneler arası esnek bağ oluşturmak" idi. Gözetleyici tasarım şablonu bunu gerçekleştirmek için biçilmiş kaftandır.

JDK bünyesinde de gözetleyici tasarım şablonu bulunmaktadır. Kendi Observer interface sınıflarını yazmadan, JDK API den bulunan java.util.Observer ve java.util.Observable sınıflarını kullanabiliriz.

Gözetleyici tasarım şablonu ne zaman kullanılır?

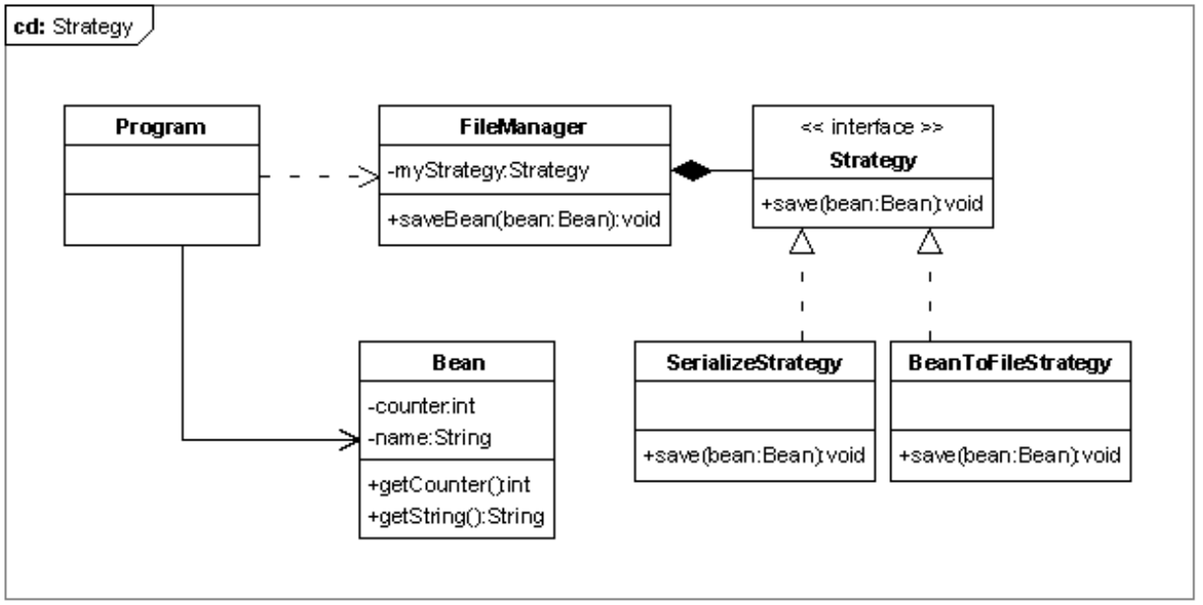
Bir nesne üzerinde yapılan değişiklik başka nesnelerin de değiştirilmesini zorunlu kılıyorsa ve kaç nesnenin değiştirilmesi gerektiğini bilmiyorsak, gözetleyici tasarım şablonun kullanarak nesneler üzerinde gerekli değişiklikleri uygulayabiliriz.

İlişkili tasarım şablonları

Adaptör tasarım şablonu kullanılarak, abone olmak isteyen nesneler belirli bir Abone interface sınıfını implemente etmek zorunda kalmadan abone olabilirler. Gözetleyici tasarım şablonu mediator tasarım şablonunu kullanarak, nesneler arası iletişimi koordine edebilir.

Strateji (Strategy Pattern)

Bir işlemi yerine getirmek için birden fazla yöntem (algoritma) mevcut olabilir. Yerine göre bir yöntem seçip, uygulamak için strateji tasarım şablonu kullanılır. Her yöntem (algoritma) bir sınıf içinde implemente edilir.



Resim 6

Seçtiğim örnekte Bean ismindeki bir sınıftan olan nesneleri iki değişik yöntemle bir dosyada tutmak istiyorum. Bean basit bir Java sınıfıdır ve counter ve name isimlerinde iki sınıf değişkenine sahiptir. get() metotları üzerinden sınıf değişkenlerine ulaşılır.

```
// Kod 29

package com.pratikprogramci.designpatterns.bolum7.strategy;

import java.io.Serializable;

/**
 * Basit bir java bean sınıfı
 *
 */
public class Bean implements Serializable {
    private int counter;
    private String name;

    public int getCounter() {
        return counter;
    }

    public void setCounter(final int counter) {
        this.counter = counter;
    }
```

```
    public String getName() {
        return name;
    }

    public void setName(final String name) {
        this.name = name;
    }
}
```

Strateji tasarım şablonunu uygulayabilmemiz için Strategy isminde bir interface sınıfı tanımlamamız gerekiyor. Bu interface sınıfı bünyesinde altsınıflar tarafından implemente edilmesi gereken metot ya da metotlar bulunur. Örneğimizde basit bir Java nesnesini bir dosyaya aktarmak üzere save(Bean bean) isminde bir metot tanımlıyoruz. Uygulamak istediğimiz yönteme göre Strategy interface sınıfını implemente eden bir algoritma seçerek, işlemi gerçekleştireceğiz. Strategy interface sınıfı aşağıdaki yapıya sahiptir:

```
// Kod 30

package com.pratikprogramci.designpatterns.bolum7.strategy;

/**
 * Strategy üst interface sınıfı. Oluşturacağımız
 * algoritmalar bu sınıfı
 * implemente ederler.
 *
 */
public interface Strategy {
    void save(Bean bean);
}
```

İmplemente etmek istediğimiz ilk yöntem (algoritma) BeanToFileStrategy ismini taşıyor. Bu sınıf bünyesinde bean nesnesinin sahip olduğu değerleri bir StringBuilder nesnesine dönüştürdükten sonra bean.txt isimli dosyaya yazıyoruz.

```
// Kod 31

package com.pratikprogramci.designpatterns.bolum7.strategy;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.Writer;

/**
```

```
 * Bean sınıfının sahip olduğu değişken d eğerlerini
 * bean.txt isimli bir dosyaya
 * yazar.
 *
 */
public class BeanToFileStrategy implements Strategy {

    @Override
    public void save(final Bean bean) {
        try {
            final StringBuilder temp = new StringBuilder();
            temp.append("counter: ").append(
                    bean.getCounter()).append("\n");

            temp.append("name: ").append(
                    bean.getName()).append("\n");

            File file = new File("c:/temp/bean.txt");
            if (file.exists()) {
                file.delete();
                file = new File("c:/temp/bean.txt");
            }

            Writer output = null;
            try {
                output = new BufferedWriter(
                        new FileWriter(file));
                output.write(temp.toString());
            } finally {
                if (output != null) {
                    output.close();
                }
            }
            System.out.println("bean.txt oluşturuldu.");
        } catch (final Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

BeanToFileStrategy sınıfı Strategy interface sınıfını implemente ettiği için save() metoduna sahiptir. Bu metot bünyesinde önce bir StringBuilder nesnesi oluşturuyoruz. Append() operasyonu ile bean nesnesinin sahip olduğu değişken isimleri ve değerleri StringBuilder nesnesine eklenir. Bu işlemin ardından File sınıfını kullanarak c:\temp altında bean.txt isminde bir dosya oluşturuyoruz. Eğer bu dosya varsa, silerek, yeniden oluşturulur. BufferedWriter sınıfını kullanarak StringBuilder

bünyesinde bulunan değerleri bu dosyaya aktarıyoruz. Bu işlemin ardından c:\temp dizininde bean.txt isminde bir dosya oluşturulacak ve içeriği aşağıdaki şekilde olacaktır:

```
counter: 10
name: name
```

BeanToFileStrategy sınıfı ile bir nesnenin sahip olduğu değişken değerlerini bir dosyaya aktardık. Bu uygulanabılır yöntemlerden bir tanesini teşkil etmektedir. Bean nesnesinin yapısını başka bir yöntem kullanarak da bir dosyaya aktarabiliriz. Yeni bir yöntem oluşturma adına SerializeStrategy isminde ikinci bir sınıf tanımlıyoruz. Bu sınıf StringBuilder yerine, bean nesnesini binary kod olarak doğrudan bean.ser dosyasına aktarır.

```
// Kod 32

package com.pratikprogramci.designpatterns.bolum7.strategy;

import java.io.FileOutputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;

/**
 * Bean sınıfının sahip olduğu değişken değerlerini
 * zerialize ederek bean.ser
 * isimli bir dosyaya yazar.
 *
 */
public class SerializeStrategy implements Strategy {

    @Override
    public void save(final Bean bean) {
        try {
            final ObjectOutput out = new ObjectOutputStream(
                    new FileOutputStream("c:/temp/bean.ser"));
            out.writeObject(bean);
            out.close();
        } catch (final Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

SerializeStrategy yöntemi kullanıldığında c:\temp dizininde bean.ser isminde bir dosya oluşturulur. Bu dosyayı bir text editörü ile incelediğimiz zaman aşağıdaki yapıda olduğunu görürüz:

```
¬í__sr_%org.javatasarim.pattern.strategy.Bean_QKy«”žÒ___I_ counterL__
namet__Ljava/lang/String;xp___t__name
```

Binary formatta olan bu dosya bean nesnesinin tüm yapısını ihtiva etmektedir. Bu dosyanın içeriğini kullanarak bean nesnesini tekrar oluşturabiliriz.

Bean nesnesini bir dosyaya aktarmak için FileManager isminde bir sınıf kullanıyoruz. Bu sınıf aşağıdaki yapıdadır:

```
// Kod 33

package com.pratikprogramci.designpatterns.bolum7.strategy;

import java.util.ResourceBundle;

/**
 * FileManager sınıfı
 */
public class FileManager {
    /**
     * Bünyesinde bir strategy nesnesi barındırır.
     */
    private Strategy strategy;

    /**
     * Singleton tasarım şablonunu kullanarak bu
     * sınıftan sadece bir nesnenin
     * oluşmasını sağlıyoruz.
     */
    public static final FileManager manager =
            new FileManager();

    /**
     * Sınıf konstrüktörü içinde strategy.properties
     * dosyasinda tanımlanmış olan
     * strategy sınıfını yükleyerek, bir strategy
     * nesnesi oluşturuyoruz.
     */
    private FileManager() {
        final String strategy =
                ResourceBundle.getBundle(
                        "com/pratikprogramci/"
                        + "designpatterns/bolum7/"
                        + "strategy/strategy")
                .getString("strategy");
```

```
        try {
            setStrategy(((Strategy) Class.forName(
                    strategy).newInstance()));
        } catch (final Exception e) {
            throw new RuntimeException(e);
        }
    }

    public static FileManager instance() {
        return manager;
    }

    public Strategy getStrategy() {
        return strategy;
    }

    public void setStrategy(final Strategy strategy) {
        this.strategy = strategy;
    }

    public void saveBean(final Bean bean) {
        getStrategy().save(bean);
    }
}
```

İlk etapta FileManager bünyesinde kullanmak istediğimiz strategy nesnesini tutmak için strategy isminde bir sınıf değişkeni tanımlıyoruz. FileManager sınıfını singleton tasarım şablonuna göre implemente edilmiştir çünkü sahip olduğu tek konstrüktör private olarak tanımlanmıştır. Bu durumda dışarıdan new operatörü ile bu sınıfın bir nesnesi oluşturulamaz. Static olarak tanımlanan manager ismindeki sınıf değişkeni ile sistemde mevcut olan tek FileManager nesnesine ulaşıyoruz.

Private olarak tanımladığımız sınıf konstrüktöründe yer alan ResourceBundle sınıfı dikkatinizi çekmis olabilir.

```
private FileManager() {
    final String strategy =
            ResourceBundle.getBundle(
                    "com/pratikprogramci/"
                    + "designpatterns/bolum7/"
                    + "strategy/strategy")
            .getString("strategy");

    try {
```

```
        setStrategy(((Strategy) Class.forName(
                strategy).newInstance()));
    } catch (final Exception e) {
        throw new RuntimeException(e);
    }
}
```

Sahip olduğumuz strategy sınıflarını daha esnek bir şekilde program kodunda kullanabilmek için strategy.properties isminde bir dosya tanımlayarak, kullanmak istediğimiz strategy sınıfının ismini bu dosyada belirtiyoruz. Strategy.properties sınıfı aşağıdaki yapıya sahiptir:

```
strategy=com.pratikprogramci.designpatterns.
            bolum7.strategy.BeanToFileStrategy
```

Bu dosya içinde strategy isminde bir anahtar ve anahtarın değeri olarak BeanToFileStrategy sınıfını tanımlıyoruz. FileManager konstrüktöründe ResourceBundle.getBundle() metodu ile strategy.properties içinde tanımlanmış olan strategy anahtarının değerini edinebiliriz. Akabinde Class.forname() operasyonu ile new yapmaya gerek kalmadan BeanToFileStrategy sınıfından bir nesne oluştuyoruz. Aynı şekilde sadece strategy.properties içinde başka bir strategy sınıfının ismini vererek, FileManager sınıfı içinde bu sınıftan bir nesne oluşturulmasını sağliyabiliriz. Bir property dosyasını kullanarak, FileManager sınıfını tekrar derlemek zorunda kalmadan istediğimiz şekilde ve zamanda kullanılan Strategy sınıfını değiştirebiliriz.

Aşağıda yer alan Test programı ile oluşturduğumuz bir bean nesnesini bir dosyaya aktarıyoruz.

```
// Kod 34

package com.pratikprogramci.designpatterns.bolum7.strategy;

/**
 * Test sınıfı
 */
public class Test {
    public static void main(final String[] args) {
        final Bean bean = new Bean();
        bean.setCounter(10);
        bean.setName("name");
        FileManager.instance().saveBean(bean);
    }
}
```

Test.main() metodunda görüldüğü gibi oluşturduğumuz bean nesnesini FileManager yardımı ile bir

dosyaya transfer etmiş oluyoruz. Test sınıfı transfer işlemi için kullandığımız yöntemi bile tanımamaktadır. Bu sayede Test sınıfını etkilemeden başka yöntemleri Strategy sınıfı olarak implemente ederek kullanabiliriz.

Strategy tasarım şablonu ne zaman kullanılır?

- Bir işlemi birden fazla yöntem (algoritma) ile implemente etmek için streteji tasarım şablonu kullanılır. Sistem gereksimleri doğrultusunda en uygun yöntem seçilerek, işlemin gerçekleştirilmesinde kullanılır.
- Kullanıcı sınıfların, kullanılan yöntemler hakkında bilgi sahibi olmamaları gerektiği durumlarda streteji tasarım şablonu kullanılır. Kullandığımız örnekte Test sınıfı hangi yöntemin kullanıldığını bilemez. Böylece kullanıcı ile sistem arasında gizli bir duvar örerek, sistemin nasıl çalıştığını kullanıcıdan saklama yeteneğine kavuşmuş oluyoruz.
- Java programlarında switch operatörünün kullanılmasi kötü bir tasarım yapıldığının işaretidir. Switch ve ona benzer yapılar yerine streteji tasarım şablonu kullanılarak daha iyi çözümler oluşturulabilir.

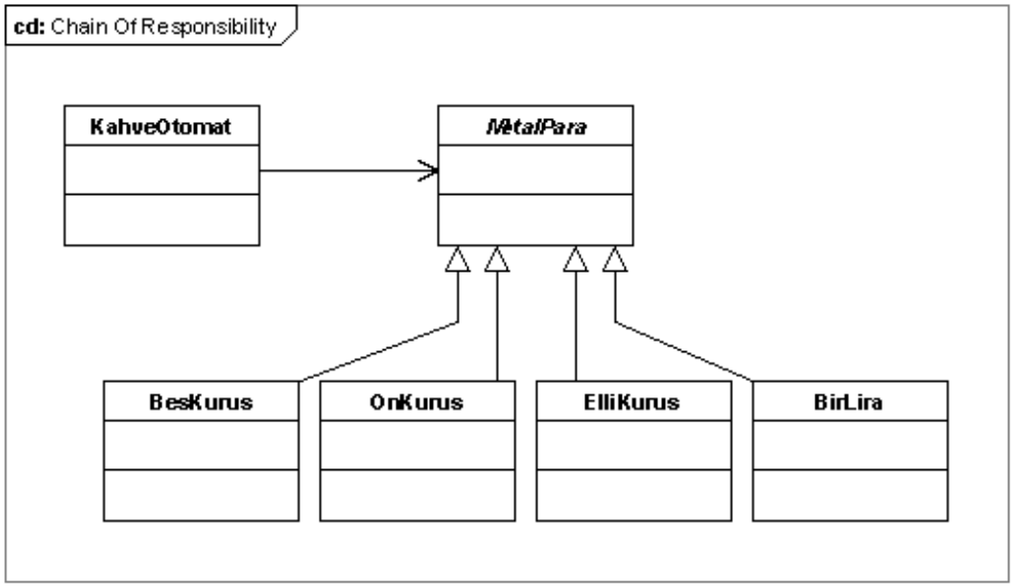
İlişkili tasarım şablonları

Strateji nesneleri flyweight tasarım şablonu kullanılarak implemente edilebilir. State nesnenin iç yapısını, strategy sunduğu davranış biçimlerini değiştirmek için kullanılırken, dekoratör nesnenin dış görüntüsünü değiştirmek için kullanılır.

Sorumluluk Zinciri (Chain Of Responsibility Pattern)

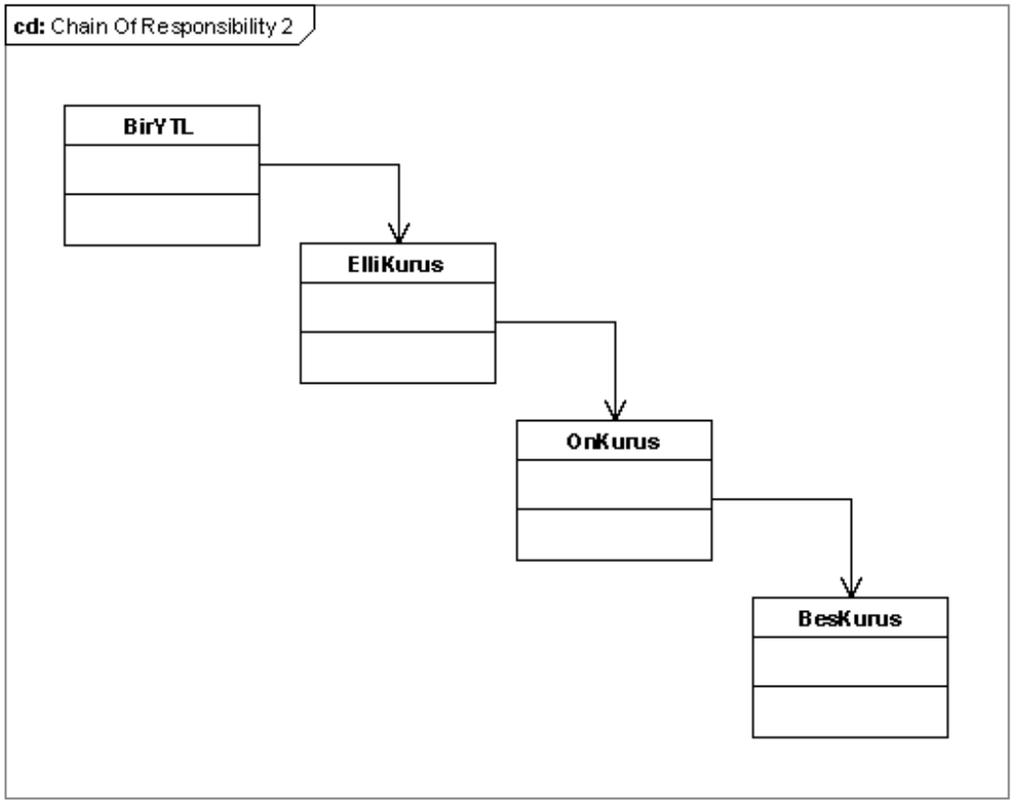
Chain of responsibility sorumluluk zinciri anlamına gelmektedir. Sisteme gönderilen bir istediğin (komut) hangi nesne tarafından cevaplanması gerektiğini bilmediğimiz durumlarda ya da isteği yapan nesne ve servis sağlayan nesne arasında sıkı bir bağ oluşmasını engellememiz gerektiğinde, Chain of Responsibility tasarım şablonu kullanılır. Bu tasarım şablonunda servis sağlayan ilgili tüm nesneler bir kolye üzerindeki boncuklar gibi birbirleriyle ilişkili hale getirilir. Bir nesne zincirdeki kendinden sonraki nesneyi tanır ve isteği kendi cevaplayamadığı durumda, kendinden sonraki nesneye iletir. Bu işlem, zincirde bulunan doğru servis sağlayıcı nesneyi bulana kadar devam eder.

Bu tasarım şablonu için ilginç bir örnek kullanmak istiyorum. İçine para atarak, kahve aldığımız bir kahve otomatı düşünelim. Bir kahvenin bedeli 1 TL olabilir. Kahveyi alabilmek için 1 TL değerindeki metal parayı otomata atmamız gerekiyor.



Resim 7

Otomatın içine atılan metal paraları tanıyabilmesi için metal paraları temsil eden nesnelerden bir zincir oluşturmamız gerekiyor. Her metal para bu zinciri, doğru para nesnesini bulana kadar baştan sona kadar dolaşacaktır. Örneğin metal para nesneler zincirini su şekilde oluşturabiliriz:



Resim 8

Otomata atılan metal para, zincirin ilk nesnesi olan BirLira tarafından karşılanır ve metal paranın bir Lira olup olmadıgı kontrol edilir. Eğer metal para bir Lira ise, otomat parayı kabul eder. Eğer atılan metal para bir Lira değilse, işlem zincirin ikinci nesnesi ElliKurus'a gönderilir. Bu işlem zincirin en son elemanı olan BesKurus nesnesine kadar devam eder. Atılan metal para eğer zincirde bulunan bir nesne ile eşdeğerde ise, o zaman metal para kabul edilir, aksi taktirde reddedilir.

```
// Kod 35

package com.pratikprogramci.designpatterns.bolum7.chainofres;

import java.util.ArrayList;

/**
 * Metal para üstsınıfı
 *
 */
public abstract class MetalPara {

    /**
     * Otomata atılan paraların tutulduğu liste
     */
    private ArrayList<MetalPara> metalParaListesi =
            new ArrayList<>();

    /**
     * Metal paranın sahip olduğu değer. 5, 10, 50,
     * 100 Kurus olabilir
     */
    private int value;

    /**
     * Zincirde yeralan bir sonraki nesne
     */
    private MetalPara next;

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }
```

```
    public MetalPara getNext() {
        return next;
    }

    public MetalPara setNext(MetalPara next) {
        this.next = next;
        return this;
    }

    public void check(MetalPara para) {
        System.out.println("Sıradaki nesne sadece "
    + this + " işleyebilir.");

        if (para.getValue() != this.value) {
            System.out.println("Uymadı, zincirdeki "
                    + "bir " + "sonraki nesneye iletiyoruz.");

            if (getNext() != null) {
                getNext().check(para);
            } else {
                System.out.println("Zincirin sonundayız. "
                        + "" + "Metal para " + para.toString() + " "
                        + "bu otomat için uygun değil.");
            }
        } else {
            metalParaListesi.add(para);
            System.out.println("Otomat tarafından "
            + this.toString() + " kabul edildi");
        }
    }
}
```

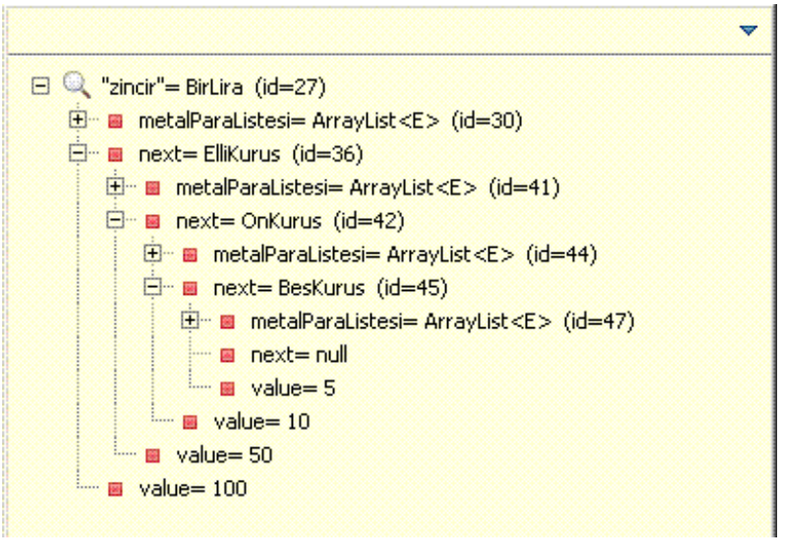
Metal paraları temsil eden MetalPara isminde soyut bir sınıf tanımlıyoruz. Metal paraları temsil eden nesnelerden oluşan bir zincir oluşturabilmek için, MetalPara sınıfı bünyesinde next isminde bir değişken yer almaktadır. Gerçek metal paralar (BesKurus, ElliKurus vs) MetalPara soyut sınıfını genişlettiği (extends) için, next değişkenini miras olarak alacaklardır. Daha sonra göreceğimiz gibi, next değişkeninde zincirin bir sonraki elemanı yer almaktadır.

check() ve setNext() metodlarının nasıl çalıştığını anlayabilmek için önce bir zincirin nasıl oluşturulduğuna göz atalım:

```
private static MetalPara zincir = null;

zincir = (new BirLira()).setNext((
```

```
        new ElliKurus()).setNext((
                new OnKurus()).setNext(
                        new BesKurus()))));
```

Zinciri oluşturan nesneler MetalPara tipinde olacaktır. Bu yüzden MetalPara tipine sahip zincir isminde bir değişken tanımlıyoruz. Zincirin ilk nesnesini new BirLira() şeklinde oluşturduktan sonra setNext() ile bir sonraki nesneyi oluşturuyoruz. Zincir için gerekli tüm nesneleri arka arkaya, setNext() metodu bünyesinde bu şekilde oluşturabiliriz. Programın bu satırlarını Eclipse altında debug ettiğimiz taktirde, şöyle bir nesne zinciri oluşacaktır:



Resim 9

Resimde de görüldüğü gibi zincir değişkeni aslında BirLira sınıfından oluşturulmuş bir nesnedir. Bu nesnenin next değişkeninde ElliKurus sınıfından bir nesne yer almaktadır. Aynı şekilde ElliKurus nesnesinin next değişkeni OnKurus ve OnKurus nesnesinin next değişkeni BesKurus sınıfından bir nesneyi ihtiva eder. Böylece

```
BirLira --> ElliKurus --> OnKurus --> BesKurus
```

şeklinde bir zincir oluşturmuş oluyoruz. Zincirdeki bir nesne, kendisinden sonraki nesneye next değişkeni üzerinden bağlıdır. Sadece zincirin en son elamanı olan BesKurus'un sahip olduğu next değişkeni null değerindedir. Null zincirin son bulduğunu ifade etmektedir.

MetalPara.check() metoduna tekrar geri dönelim. Bu metot bünyesinde otomata atılan para ile, zincirde sırası gelmiş olan nesne karşılaştırılır. Eğer sıradaki nesne, atılan metal para ile uyuşmuyorsa, getNext().check(para) ile kontrol zincirdeki bir sonraki nesneye verilir. Bu işlem zincirin sonuna gelene kadar tekrarlanır.

```
// Kod 36

package com.pratikprogramci.designpatterns.bolum7.chainofres;

/**
 * 100 Kurusluk (1 TL) metal para sınıfı
 *
 */
public class BirLira extends MetalPara {

    public String toString() {
        return "1 Lira";
    }

    public BirLira() {
        setValue(100);
    }
}
```

BirLira ve ElliKurus sınıflarını gördük. Diğer sınıflarıda analog olarak aynı şekilde oluşturuyoruz.

```
// Kod 37

package com.pratikprogramci.designpatterns.bolum7.chainofres;

public class KahveOtomati {

    private static MetalPara zincir = null;

    public static void main(String[] args) {
        zincir = (new BirLira()).setNext((
                new ElliKurus()).setNext((
                        new OnKurus()).setNext(
                                new BesKurus())));

        paraAt(new BirKurus());
        paraAt(new ElliKurus());
        paraAt(new OnKurus());
        paraAt(new OnKurus());
        paraAt(new OnKurus());
        paraAt(new OnKurus());
        paraAt(new BesKurus());
        paraAt(new BesKurus());
        paraAt(new BirKurus());

    }
```

```
    public static void paraAt(MetalPara para) {
        System.out.println(
                "+++ ------------------------ +++");
        System.out.println(
                "Otomata " + para.toString() + "  atıldı.");
        zincir.check(para);
        System.out.println(
                "+++ ------------------------ +++\n");
    }

}
```

Bir kahve otomatını simule etmek amacıyla KahveOtomati isminde bir test sınıfı oluşturuyoruz. Bu sınıfın bünyesinde metal paraları kontrol etmek için gerekli nesne zincirini oluştuyoruz. paraAt() metodu ile otomata metal para atmaya başlıyoruz. Şimdi ekran çıktısını inceliyelim:

```
1.  +++ ------------------------ +++
2. Otomata 1 kurus  atıldı.
3. Sıradaki nesne sadece 1 Lira işleyebilir.
4. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
5. Sıradaki nesne sadece 50 kurus işleyebilir.
6. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
7. Sıradaki nesne sadece 10 kurus işleyebilir.
8. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
9. Sıradaki nesne sadece 5 kurus işleyebilir.
10. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
11. Zincirin sonundayız. Metal para 1 kurus bu...
+++ ------------------------ +++

+++ ------------------------ +++
12. Otomata 50 kurus  atıldı.
13. Sıradaki nesne sadece 1 Lira işleyebilir.
14. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
15. Sıradaki nesne sadece 50 kurus işleyebilir.
16. Otomat tarafından 50 kurus kabul edildi
+++ ------------------------ +++

+++ ------------------------ +++
17. Otomata 10 kurus  atıldı.
18. Sıradaki nesne sadece 1 Lira işleyebilir.
19. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
20. Sıradaki nesne sadece 50 kurus işleyebilir.
21. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
22. Sıradaki nesne sadece 10 kurus işleyebilir.
23. Otomat tarafından 10 kurus kabul edildi
```

```
+++ ------------------------ +++

+++ ------------------------ +++
24. Otomata 10 kurus  atıldı.
25. Sıradaki nesne sadece 1 Lira işleyebilir.
26. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
27. Sıradaki nesne sadece 50 kurus işleyebilir.
28. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
29. Sıradaki nesne sadece 10 kurus işleyebilir.
30. Otomat tarafından 10 kurus kabul edildi
+++ ------------------------ +++

+++ ------------------------ +++
31. Otomata 10 kurus  atıldı.
32. Sıradaki nesne sadece 1 Lira işleyebilir.
33. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
34. Sıradaki nesne sadece 50 kurus işleyebilir.
35. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
36. Sıradaki nesne sadece 10 kurus işleyebilir.
37. Otomat tarafından 10 kurus kabul edildi
+++ ------------------------ +++

+++ ------------------------ +++
38. Otomata 10 kurus  atıldı.
39. Sıradaki nesne sadece 1 Lira işleyebilir.
40. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
41. Sıradaki nesne sadece 50 kurus işleyebilir.
42. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
43. Sıradaki nesne sadece 10 kurus işleyebilir.
44. Otomat tarafından 10 kurus kabul edildi
+++ ------------------------ +++

+++ ------------------------ +++
45. Otomata 5 kurus  atıldı.
46. Sıradaki nesne sadece 1 Lira işleyebilir.
47. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
48. Sıradaki nesne sadece 50 kurus işleyebilir.
49. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
50. Sıradaki nesne sadece 10 kurus işleyebilir.
51. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
52. Sıradaki nesne sadece 5 kurus işleyebilir.
53. Otomat tarafından 5 kurus kabul edildi
+++ ------------------------ +++

+++ ------------------------ +++
54. Otomata 5 kurus  atıldı.
55. Sıradaki nesne sadece 1 Lira işleyebilir.
```

```
56. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
57. Sıradaki nesne sadece 50 kurus işleyebilir.
58. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
59. Sıradaki nesne sadece 10 kurus işleyebilir.
60. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
61. Sıradaki nesne sadece 5 kurus işleyebilir.
62. Otomat tarafından 5 kurus kabul edildi
+++ ------------------------ +++

+++ ------------------------ +++
63. Otomata 1 kurus  atıldı.
64. Sıradaki nesne sadece 1 Lira işleyebilir.
65. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
66. Sıradaki nesne sadece 50 kurus işleyebilir.
67. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
68. Sıradaki nesne sadece 10 kurus işleyebilir.
69. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
70. Sıradaki nesne sadece 5 kurus işleyebilir.
71. Uymadı, zincirdeki bir sonraki nesneye iletiyoruz.
72. Zincirin sonundayız. Metal para 1 kurus bu otomat...
+++ ------------------------ +++
```

1. satırda otomata bir kuruşluk metal para atıldığını görmekteyiz. Bu metal para tüm zinciri baştan aşağı dolaştıktan sonra 11. satırda görüldüğü gibi otomat tarafından reddedilmektedir, çünkü otomat sadece 1 TL, 50 kuruş, 10 kuruş ve 5 kuruşluk metal paraları kabul etmektedir.

2. satırda 50 kuruşluk bir metal para atılmaktadır. 50 kuruşluk metal para zincirin ilk nesnesi BirLira'ya 13. satırda verilir. Bu nesne sorumlu olmadığından, kontrolü 14 satırda ElliKurus nesnesine devreder. 16. satırda görüldüğü gibi 50 kuruşluk metal para, zincirin bu elamanı tarafından kabul edilir. Tüm işlemlerin ardından atılan 1x50, 4x10 ve 2x5 kuruşluk metal paraların kabul edildiğini, lakin 2x1 kuruşun reddedildiğini görmekteyiz.

Chain of Responsibility tasarım şablonu ne zaman kullanılır?

Sisteme yöneltilen bir isteğin (request) birden fazla nesne tarafından işlenmesi gerektiği durumlarda Chain of Responsibility tasarım şablonu kullanılır.

Kullanıcı sınıf ile servis sağlayan nesne arasında sıkı bir bağ oluşmasını engellemek için Chain of Responsibility tasarım şablonu kullanılabilir. Chain of Responsibility ile kullanıcı ve servis sunucu nesneler birbirlerini tanımak zorunda değildirler.

İlişkili tasarım şablonları

Chain of responsibility genelde composite tasarım şablonu ile beraber kullanılır. Bir composite

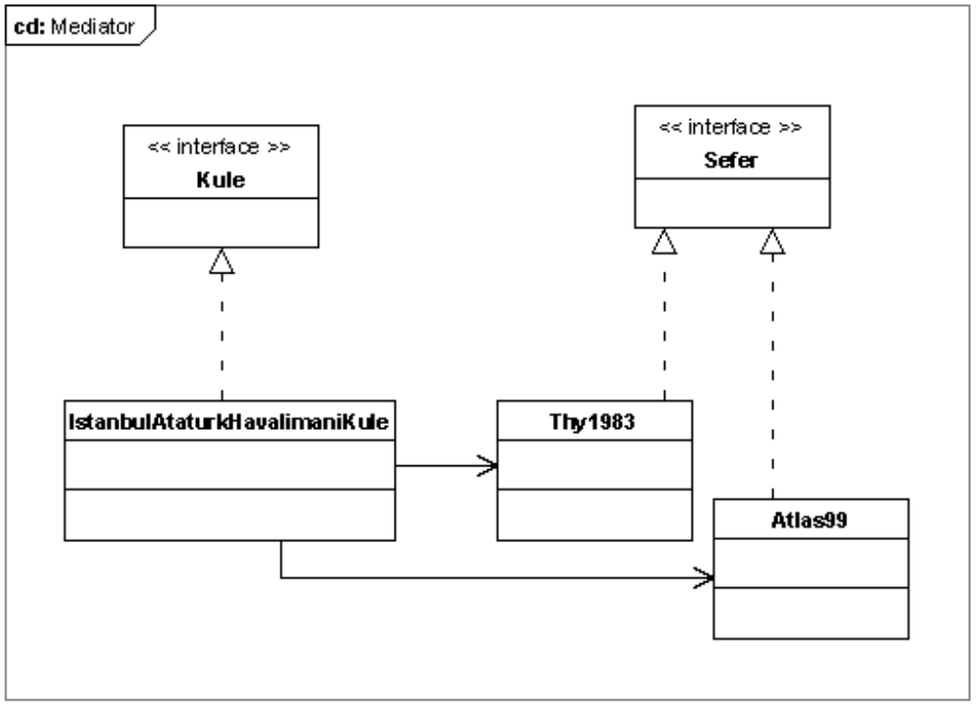
nesneyi oluşturan alt nesneler zincirin parçası olarak düşünülebilir. Chain of responsibility tasarım şablonu ile bu zincir üzerinde işlem yapılır. Command tasarım şablonu ile Chain of responsibility bünyesindeki istekler (request) implemente edilebilir.

Aracı (Mediator Pattern)

OO (object oriented = nesneye yönelik) tarzı programlanmış sistemlerin kökeninde sistemi oluşturan nesneler arasındaki iletişim ve interaksiyon yatmaktadır. Nesneler, görevlerini yerine getirmek için başka nesneler ile ortak çalışırlar. Bunu gerçekleştirebilmek için nesnelerin birbirlerine mesaj göndermeleri gerekir. Mesaj gönderme işlemi, nesnenin sahip olduğu bir metodunu kullanmak suretiyle gerçekleşir.

Birçok nesnenin olduğu bir sistemde, nesneler arası iletişi zaman içinde karmaşık bir hal alabilir. Bir nesnenin, beraber iş yapmak istediği her nesneyi tanıması ve haberleşmek için gerekli mantığı implemente etmesi gerekir. Bu zamanla bakımı ve geliştirilmesi zor bir kodun oluşmasına sebep verir. Ayrıca nesneler birbirlerini tanımak zorunda olduklarından, aralarında sıkı bir bağ oluşur. Birinci bölümde yer alan tasarım prensiblerini tekrar gözden geçirecek olursak, iyi bir tasarımın ana şartlarından birinin nesneler arası esnek bağlar oluşturmaktan geçtiğini görecegiz.

Mediator tasarım şablonunu nesnelerin yönetimi ve aralarındaki iletişimin merkezi bir noktadan koordinasyonu için kullanılır. Bu nesneler arasındaki bağı azaltır ve sadece bir sınıfı iletişimi koordine etmekle sorumlu kılar.



Resim 10

Bir havalimanında görev yapan kule mediator olarak düşünülebilir. Eğer havalımanlarında kuleler olmasaydı, havalimanında bulunan uçaklar birbirleriyle haberleşmek ve iniş kalkışları hepberaber koordine etmek zorunda kalırlardı. Bunun ne gibi sonuçlar doğurabileceğini anlamak zor değil.

Kulenin görevi, uçakların iniş ve kalkışlarını koordine etmektir. Bunun için sefer yapan her uçak iniş ya da kalkış için kuleyle haberleşmek ve gerekli direktifleri almak zorundadır. Bunun haricide kule uçağın iniş ya da kalkışına karışmaz. Her uçak iniş ve kalkıştan kendisi sorumludur.

Kulelerin sefer koordinasyonunu aşağıdaki şekilde Mediator tasarım şablonu ile implemente edebiliriz.

```
// Kod 38

package com.pratikprogramci.designpatterns.bolum7.mediator;

/**
 * Kule interface sınıfı
 *
 */
public interface Kule {
    /**
     * Sefer yapan ucaklar bu metot
     * ile iniş izni alirlar
```

```
     */
    void inisIzniVer(Sefer sefer);

    /**
     * Sefer yapan ucaklar bu metot ile
     * kalkış izni alirlar
     */
    void kalkisIzniVer(Sefer sefer);

    /**
     * Sefer yapan bir ucak kuledeki sefer
     * listesine bu metot ile eklenir. Kule
     * listesinde olan uçaklara servis sunar.
     */
    void seferEkle(Sefer sefer);
}
```

Öncelikle Kule isminde, seferler arası koordinasyonu sağlamak için kullanılacak bir interface sınıf oluşturuyoruz. Kule mediator olarak düşünülebilir. Her uçak bu interface sınıfında yer alan metotlar aracılığıyla kule ile baglantı kurup, iniş ve kalkış izni alacaktır.

```
// Kod 39

package com.pratikprogramci.designpatterns.bolum7.mediator;

import java.util.ArrayList;

/**
 * İstanbul Atatürk havalimanında bulunan kuleyi
 * temsil eden sınıf.
 *
 */
public class IstanbulAtaturkHavalimaniKule implements Kule {

    /**
     * İniş ve kalkış yapmak isteyen uçakların yer aldığı
     * liste
     */
    private ArrayList<Sefer> seferList = new ArrayList<Sefer>();

    public ArrayList<Sefer> getSeferList() {
        return seferList;
    }

    public void setSefer(ArrayList<Sefer> sefer) {
        this.seferList = sefer;
```

```
    }

    /**
     * İniş ya da kalkış yapmak isteyen ucaklar kendilerine bu
     * listeye eklemek
     * zorundadırlar.
     */
    public void seferEkle(Sefer sefer) {
        if (!getSeferList().contains(sefer)) {
            getSeferList().add(sefer);
            System.out.println("Kule: " + sefer.toString()
            + " " + "- sefer listesine eklendi");
        }
    }

    /**
     * Sefer listesinden bir uçağı silmek için kullanılır.
     *
     * @param sefer
     */
    public void seferSil(Sefer sefer) {
        if (getSeferList().contains(sefer)) {
            getSeferList().remove(sefer);
            System.out.println("Kule: " + sefer.toString()
            + " " + "- sefer listesinen silindi");
        }
    }

    /**
     * İniş izni almak isteyen ucaklar bu metot ile kuleden
     * iniş izni talep
     * ederler.
     */
    public void inisIzniVer(Sefer sefer) {
        boolean inisYapiliyor = false;
        Sefer inisYapanSefer = null;

        if (getSeferList().contains(sefer)) {
            /**
             * Tüm sefer listesini kontrol ediyoruz
             */
            for (int i = 0; i < getSeferList().size(); i++) {
                Sefer tempSefer = getSeferList().get(i);

                /**
                 * Eğer listede herhangi bir ucak iniş yapıyorsa,
                 * o zaman sefer
```

```
                 * değişkenindeki uçağa iniş izni verilmez!
                 */
                if (tempSefer.isInisYapiyor()) {
                    inisYapiliyor = true;
                    inisYapanSefer = tempSefer;
                    break;
                }
            }

            if (!inisYapiliyor) {
                /**
                 * Eğer iniş yapan ucak yoksa, iniş izni verilir.
                 * Aynı anda
                 * sadece bir ucak iniş yapabilir.
                 */
                sefer.setInisYapiyor(true);
                System.out.println("Kule: " +
                sefer.toString() + "iniş izni verildi.");
            } else {
                if (!sefer.equals(inisYapanSefer)) {
                    System.out.println("Kule: " +
                inisYapanSefer + " iniş yapıyor. " + "Bu yüzden "
                            + sefer
                            + " seferine " +
                            " iniş izni verilemiyor!");
                }
            }
        } else {
            System.out.println(
                    sefer.toString() + " sefer  "
                            + "listesinde yeralmiyor!");
        }

    }

    /**
     * Kalkis izni almak isteyen ucaklar bu metot
     * ile kuleden kalkış izni talep
     * ederler.
     */
    public void kalkisIzniVer(Sefer sefer) {
        boolean kalkisYapiliyor = false;
        Sefer kalkisYapanSefer = null;

        if (getSeferList().contains(sefer)) {
            /**
             * Tüm sefer listesini kontrol ediyoruz
```

```
             */
            for (int i = 0; i < getSeferList().size(); i++) {
                Sefer tempSefer = getSeferList().get(i);

                /**
                 * Eğer listede herhangi bir ucak kalkış yapıyorsa,
                 * o zaman
                 * sefer değişkenindeki uçağa kalkış izni verilmez!
                 */
                if (tempSefer.isKalkisYapiyor()) {
                    kalkisYapiliyor = true;
                    kalkisYapanSefer = tempSefer;
                    break;
                }
            }

            if (!kalkisYapiliyor) {
                /**
                 * Eğer kakis yapan ucak yoksa, kalkış
                 * izni verilir. Aynı anda
                 * sadece bir ucak kalkış yapabilir.
                 */
                sefer.setKalkisYapiyor(true);
                System.out.println("Kule: " +
                sefer.toString() + "kalkış izni verildi.");
            } else {
                if (!sefer.equals(kalkisYapanSefer)) {
                    System.out.println("Kule: " +
                kalkisYapanSefer + " kalkış yapıyor. "
                            + " Bu yüzden " + sefer + " "
                            + "seferine kalkış izni verilemiyor!");
                }
            }
        } else {
            System.out.println(sefer.toString() + " "
                    + "sefer " + "listesinde yeralmiyor!");
        }
    }
}
```

Kule interface sınıfını implemente eden IstanbulAtaturkHavalimaniKule sınıfı, uçakların iniş ve kalkışlarını koordine etmek için gerekli metotlara sahiptir. Kule kalkış ve inişleri koordine etmek zorunda olduğu için, tüm seferleri seferList isminde bir değişkende tutmaktadır. İniş ve kalkış yapan uçaklar bu listede yer almak zorundadırlar. Aynı anda sadece bir uçak inebilir ya da kalkabilir. Bir uçağın kakış anında, başka bir uçağın kalkış talebinde bulunması sonuç getirmez,

çünkü kule hangi uçağın kalkış yaptığını bildiğinden, kalkış işlemi tamamlanana kadar diğer uçaklara kalkış izni vermez. Kule sınıfı bu şekilde kalkış ve inişleri koordine eder.

```
// Kod 40

package com.pratikprogramci.designpatterns.bolum7.mediator;

/**
 * Sefer interface. Herhanbi bir havayolu bu interface
 * sınıfını implemente
 * ederek bir sefer oluşturabilir.
 *
 * @author Ozcan Acar
 *
 */
public interface Sefer {

    /**
     * Kuleden iniş izni almak için kullanılır.
     */
    void inisIzniAl();

    /**
     * Kuleden kalkış izni almak için kullanılır.
     */
    void kalkisIzniAl();

    /**
     * true ise ucak kalkış aşamasındadır.
     * Sadece bir sefer aynı anda kalkış
     * yapabilir. Kule bunu kontrol eder.
     *
     * @return
     */
    boolean isKalkisYapiyor();

    /**
     * true ise ucak iniş aşamasındadır Sadece bir
     * sefer aynı anda iniş
     * yapabilir. Kule bunu kontrol eder.
     *
     * @return
     */
    boolean isInisYapiyor();

    /**
```

```
     * Kule tarafından iniş yapan bir ucak için
     * true olarak set edilir.
     *
     * @param value
     */
    void setInisYapiyor(boolean value);

    /**
     * Kule tarafından kalkış yapan bir ucak için true
     * olarak set edilir.
     *
     * @param value
     */
    void setKalkisYapiyor(boolean value);

    /**
     * Kalkis yapan bir ucak, kalkış işlemini
     * bitirdiğini göstermek için true
     * olarak set eder.
     *
     * @param value
     */
    void setKalkisTamamlandi(boolean value);

    /**
     * İniş yapan bir ucak, iniş işlemini bitirdiğini
     * göstermek için true olarak
     * set eder.
     *
     * @param value
     */
    void setInisTamamlandi(boolean value);

}
```

Bir uçak şirketinin planlı bir sefer oluşturabilmesi için Sefer interface sınıfını implemente etmesi gerekmektedir. Bu interface bünyesinde kule ile olan koordinasyonu gerçekleştirmek için gerekli metotlar yer almaktadır. Örneğin uçak inisIzniAl() metodu ile kuleye iniş, kalkisIzniAl() metodu ile kalkış talebinde bulunur. Bu metotlar uçak ile kule arasında haberleşmek için kullanılır. Lakin işlem yapma kontrolü kulededir ve devamlı hangi uçağın iniş ya da kalkışta olduğunu takip edebildigi için, istediği uçağa iniş ya da kalkış izni verir. Sorumluluk tamamen kule sınıfındadır.

```
// Kod 41

package com.pratikprogramci.designpatterns.bolum7.mediator;
```

```
/**
 * Sefer interface sınıfını implemente eden altsınıflarda
 * ortak değişken ve
 * metot kullanımını sağlamak için bu abstract ara
 * sınıf oluşturulmuştur.
 * Altsınıflar bu sınıfın degiskenlerini kullanabilir.
 *
 */
public abstract class BaseSefer implements Sefer {

    private boolean inisTamamlandi;
    private boolean kalkisTamamlandi;
    private boolean inisYapiyor;
    private boolean kalkisYapiyor;

    private Kule kule;

    public Kule getKule() {
        return kule;
    }

    public void setKule(Kule kule) {
        this.kule = kule;
    }

    public boolean isInisTamamlandi() {
        return inisTamamlandi;
    }

    public void setInisTamamlandi(boolean
            inisTamamlandi) {
        this.inisTamamlandi = inisTamamlandi;
        this.inisYapiyor = false;
        System.out.println(this.toString() +
                " inişi tamamladı.");
    }

    public boolean isKalkisTamamlandi() {
        return kalkisTamamlandi;
    }

    public void setKalkisTamamlandi(
            boolean kalkisTamamlandi) {
        this.kalkisTamamlandi = kalkisTamamlandi;
        this.kalkisYapiyor = false;
        System.out.println(this.toString()
```

```
                + " kalkışı tamamladı.");
    }

    public boolean isInisYapiyor() {
        return inisYapiyor;
    }

    public void setInisYapiyor(boolean inisYapiyor) {
        this.inisYapiyor = inisYapiyor;
    }

    public boolean isKalkisYapiyor() {
        return kalkisYapiyor;
    }

    public void setKalkisYapiyor(boolean kalkisYapiyor) {
        this.kalkisYapiyor = kalkisYapiyor;
    }
}
```

Her uçağın kalkış ve iniş işlemleri hakkındaki bilgileri tutabileceği değişkenlere ihtiyacı vardır. Örneğin iniş işlemine geçmiş olan bir uçağın inisYapiyor değişkeninin değeri true dur. Kule bu şekilde her uçağın sahip olduğu son durumu bu değişkenler üzerinden takip eder ve gerekli durumlarda bu değişkenlerin değerlerini değiştirir. Ayrıca her uçak bağlı olduğu kuleyi tanımak zorundadır. Bu şekilde doğru kuleden iniş ya da kalkış izni alabilir. Daha sonra göreceğimiz gibi bir sefer oluşturulurken, konstrüktör parametresi olarak bağlı olunan kule nesnesi kullanılır. Oluşturacağımız her sefer (uçak) için bu değişkenlere ihtiyacımız olduğundan ve her sınıfta tekrardan bu değişkenleri tanımlamak istemediğimiz için BaseSefer isminde, Sefer interface sınıfını implemente eden bir soyut ara sınıf oluştuyoruz. Sefer yapan uçaklar Sefer interface sınıfı implemente etmek zorunda kalmadan, BaseSefer soyut sınıfını genişletir ve böylece gerekli metot ve değişkenlere sahip olurlar.

```
// Kod 42

package com.pratikprogramci.designpatterns.bolum7.mediator;

/**
 * 1983 sefer nolu THY uçağı
 *
 */
public class Thy1983 extends BaseSefer {

    public Thy1983(Kule kule) {
        setKule(kule);
```

```
        kule.seferEkle(this);
    }

    public String toString() {
        return "Thy 1983";
    }

    public void inisIzniAl() {
        getKule().inisIzniVer(this);
    }

    public void kalkisIzniAl() {
        getKule().kalkisIzniVer(this);
    }
}


package com.pratikprogramci.designpatterns.bolum7.mediator;

/**
 * 99 nolu Atlas seferi
 *
 */
public class Atlas99 extends BaseSefer {

    public Atlas99(Kule kule) {
        setKule(kule);
        kule.seferEkle(this);
    }

    public String toString() {
        return "Atlas 99";
    }

    public void inisIzniAl() {
        getKule().inisIzniVer(this);

    }

    public void kalkisIzniAl() {
        getKule().kalkisIzniVer(this);
    }
}
```

Atlas99 ve Thy1983 sınıfları sefer yapan iki uçağı temsil etmektedirler. İmplementasyonumuzu Test.main() ile test edebiliriz.

```
// Kod 43

package com.pratikprogramci.designpatterns.bolum7.mediator;

/**
 * Test sınıfı
 *
 */
public class Test {

    public static void main(String[] args) {
        Kule kule =
                new IstanbulAtaturkHavalimaniKule();
        Sefer thy1983 = new Thy1983(kule);
        Sefer atlas99 = new Atlas99(kule);

        thy1983.inisIzniAl();
        atlas99.inisIzniAl();
        thy1983.setInisTamamlandi(true);
        atlas99.inisIzniAl();
        thy1983.inisIzniAl();
        atlas99.setInisTamamlandi(true);

        atlas99.kalkisIzniAl();
        thy1983.kalkisIzniAl();
        thy1983.kalkisIzniAl();
        atlas99.setKalkisTamamlandi(true);
        thy1983.kalkisIzniAl();
    }
}
```

Ekran çıktısı aşağıdaki şekilde olacaktır:

```
Kule: Thy 1983 - sefer listesine eklendi
Kule: Atlas 99 - sefer listesine eklendi
Kule: Thy 1983inis izni verildi.
Kule: Thy 1983 iniş yapıyor. Bu yüzden Atlas 99 seferine  iniş izni verilemiyor!
Thy 1983 inişi tamamladı.
Kule: Atlas 99inis izni verildi.
Kule: Atlas 99 iniş yapıyor. Bu yüzden Thy 1983 seferine  iniş izni verilemiyor!
Atlas 99 inişi tamamladı.
Kule: Atlas 99kalkis izni verildi.
Kule: Atlas 99 kalkış yapıyor.  Bu yüzden Thy 1983 seferine kalkış izni verilemiyor!
Kule: Atlas 99 kalkış yapıyor.  Bu yüzden Thy 1983 seferine kalkış izni verilemiyor!
Atlas 99 kalkışı tamamladı.
```

```
Kule: Thy 1983kalkis izni verildi.
```

Görüldüğü gibi iki uçak birbirleriyle bağlantı kurmak zorunda kalmadan kule üzerinden iniş ve kalkış işlemleri için gerekli direktifleri almaktadırlar. Kule hangi seferin hangi durumda olduğunu takip etmekte ve buna göre iniş ve kalkış izinlerini vermektedir. Aynı anda iki uçak iniş ya da kalkış yapamaz, çünkü kule buna izin vermemektedir.

Kulenin olmadığı bir sistemi implemente ettigimiz zaman, ,iletişim havalimanında iniş ve kalkış için bekleyen tüm uçaklar arasında olmak zorundadır. Yüzlerce uçağın bulunduğu bir limanda, her uçağın geri kalan tüm uçaklarda iniş ve kalkış işlemlerini koordine etmesi imkansızdır. Mediator bu sorunu çözerek, uçaklar arası iletişimi koordine etmektedir. Böylece tüm uçaklar birbirlerini tanımak ve haberleşmek zorunda kalmadan iniş ve kalkış işlemlerini gerçekleştirebilirler.

Havalimanına gelen her yeni uçak, sadece kuleye geldiğini bildirerek, sisteme dahil olacaktır. Aynı şekilde kalkış işlemini tamamlamış olan bir uçak, kuleye ayrıldığını bildirir. Bunun haricinde diğer uçaklarla haberleşmek zorunluluğu yoktur. İniş ve kalkış için gerekli koordinasyon sadece Kule sınıfı tarafından implemente edildiği için, burada yer alan kod uçakları etkilemeden değiştirilebilir.

İmplementasyon esnasında dikkat edilmesi gereken bir husus vardır! Eğer özen gösterilmesse, mediator (Kule) sınıfı, tüm koordinasyonu gerçekleştirdiği için komplike bir hal alabilir.

Mediator tasarım şablonu ne zaman kullanılır?

Birden fazla nesnenin iletişim gereksinimini merkezi bir alanda koordine etmek için Mediator tasarım şablonu kullanılır.

İlişkili tasarım şablonları

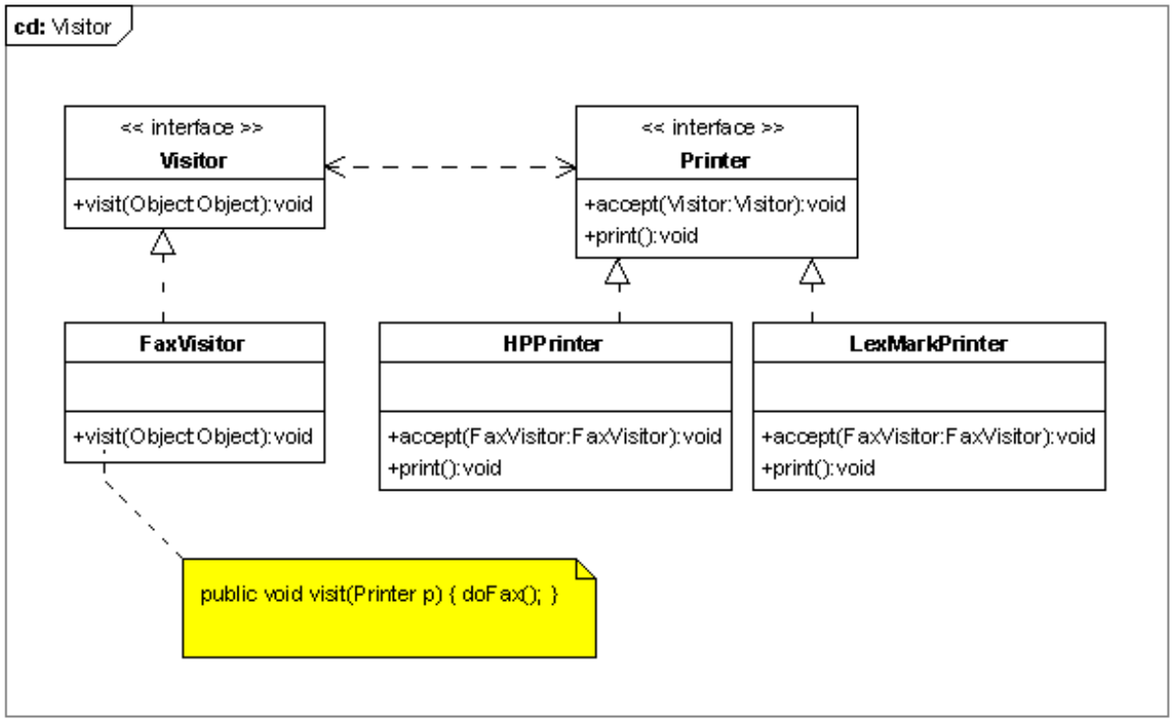
Facade tasarım şablonu ile alt sistemlere erişim koordine edilir. Burada erişim tek yönlüdür. Kullanıcı sınıf facade aracılığıyla alt sistemlere erişir. Bunun aksine Mediator tasarım şablonunda nesneler arası iletişim her yönde olabilir.

Ziyaretçi (Visitor Pattern)

Bir sınıf hirarşisinde yer alan sınıflara yeni metot eklemenmesi gerektiğinde ya soyut olan üstsınıflara yeni metot eklenir ve böylece altsınıflar bu metotları miras olarak alırlar, ya da altsınıflara metotlar eklenerek, gerekli işlem implemente edilir. Her iki yönteminde beraberinde getirdiği dezavantajlar bulunmaktadır. Soyut üstsınıflara metot eklendiğinde, bazı altsınıflar istemedikleri halde belirli davranış biçimlerine sahip olabilirler. Bu durumu değiştirmek için miras aldıkları metodu tekrar implemente etmek zorunda kalırlar. İkinci yöntem ile altsınıflara istenilen

metotlar eklenir, ama bu da zaman içinde bakımı zor sınıf hirarşisinin oluşmasına neden olabilir. Öyle ya da böyle sınıflar üzerinde değişiklik yapılması gerekmektedir. Ama sınıf hirarşisini ve sınıfları değiştirmek zorunda kalmadan yeni metotlar eklenebilir mi?

Visitor tasarım şablonu, bir sınıf hirarşisinde yer alan sınıflar üzerinde değişiklik yapmadan, bu sınıflara yeni metotların eklenmesini kolaylaştırmaktadır. İstenilen metot bir Visitor sınıfında implemente edilir.



Resim 11

Kitabın birinci bölümünde kullandığımız Printer örneğine geri dönelim ve Visitor tasarım şablonu ile herhangi bir yazıcıya fax gönderme özelliğini nasıl ekliyebileceğimizi görelim.

```
// Kod 44

package com.pratikprogramci.designpatterns.bolum7.visitor;

/**
 * Visitor interface sınıfı
 *
 */
public interface Visitor {
    void visit(HPPrinter p);
```

```
    void visit(LexMarkPrinter p);
}
```

Visitor tasarım şablonunu uygulayabilmek için Visitor isminde bir interface sınıf oluşturmamız gerekiyor. Visitor interface sınıfında, faks gönderme özelliğine kavuşması gereken tüm altsınıflar için bir visit() metodu oluşturuyoruz. İki çesit yazıcıya sahip olduğumuz için (HPPrinter ve LexMarkPrinter) iki visit() metodu oluştuyoruz. Faks gönderme işlemini daha sonra visit() metodu bünyesinde implemente edeceğiz.

```
// Kod 45

package com.pratikprogramci.designpatterns.bolum7.visitor;

/**
 * Fax gönderme özelligine sahip visitor sınıfı
 *
 */
public class FaxVisitor implements Visitor {

    public void visit(HPPrinter p) {
        System.out.println(p.toString() +
                " is faxing...");
    }

    public void visit(LexMarkPrinter p) {
        System.out.println(p.toString()
                + " can not fax! " + "(not implemented)");
    }
}
```

Faks göndermek için gerekli program mantığını FaxVisitor isminde, Visitor interface sınıfını implemente eden bir sınıf bünyesinde oluşturuyoruz. Her altsınıf için burada bir visit() metodu implemente edilir.

```
// Kod 46

package com.pratikprogramci.designpatterns.bolum7.visitor;

/**
 * Printer interface sınıfı
 *
 */
public interface Printer {
```

```
    void print();

    void accept(Visitor v);
}
```

Faks gönderme metodunu eklemek istediğimiz sınıflar HPPrinter ve LexMarkPrinter, Printer interface sınıfını implemente ederler. Bu interface bünyesinde yazıcı için gerekli metotların yanı sıra, accept() isminde bir metot yer almaktadır. Visitor ile yeni metot eklenmek istenen altsınıf arasındaki bağ accept() metodu üzerinden kurulur. Eklenmek istenen yeni metot Visitor bünyesinde implemente edildiği için, altsınıf bu metoda ulaşabilmek için bir Visitor referansını alır ve onun visit() metodunu kullanılır.

```
// Kod 47

package com.pratikprogramci.designpatterns.bolum7.visitor;

/**
 * HP Printer
 *
 */
public class HPPrinter implements Printer {
    /**
     * Kendisine gelen visitor referansı
     * üzerinden Visitor nesnesinin visit()
     * metodunu kullanır. Parametre olarak
     * kendisini verdiği için (visit(this))
     * Visitor nesnesi hangi visit() metodunu
     * calistirmasi gerektiğini bilir.
     */
    public void accept(Visitor v) {
        v.visit(this);
    }

    public void print() {
        System.out.println(this.toString()
                + " is printing...");
    }

    public String toString() {
        return "HP Printer";
    }
}
```

HPPrinter.accept() metodunda görüldüğü gibi, referans olarak gelen visitor nesnesinin visit(this)

metodu kullanılarak, kontrol tekrar Visitor sınıfına devredilir. Bu işleme double dispatch ismi verilmektedir.

```
// Kod 48

package com.pratikprogramci.designpatterns.bolum7.visitor;

/**
 * Test sınıfı
 *
 */
public class Test {

    public static void main(String[] args) {
        Printer hp = new HPPrinter();
        Printer lexmark = new LexMarkPrinter();

        hp.print();
        lexmark.print();

        Visitor visitor = new FaxVisitor();

        hp.accept(visitor);
        lexmark.accept(visitor);
    }
}
```

Double dispatch işleminin nasıl gerçekleştiğini Test.main() bünyesinde görmekteyiz. Printer veri tipinde olan hp, lexmark ve faks göndermek için kullanacağımız visitor nesnesini oluşturduktan sonra, printer nesnelerinin accept() metotlarına visitor nesnesini parametre olarak veriyoruz. Bu first (ilk) dispatch (yönlendirme) işlemidir. Kontrol burada printer sınıflarına geçmektedir. Printer.accept() metoduna tekrar göz attığımız zaman v.visit(this) ile kontrolün tekrar visitor nesnesine devredildiğini görmekteyiz, bu da second (ikinci) dispatch işlemidir. Yönlendirme iki sefer gerçekleştiği için bu işleme double (çift) dispatch ismi verilmektedir. Sonuç itibariyle printer nesnesi visit() metodunu kullanarak, faks gönderme işlemi gerçekleştirilmektedir. Bu şekilde değişik türde visitor sınıfları implemente edilerek, Printer veri tipindeki sınıflara sahip oldukları kodu değiştirmek zorunda kalmadan, yeni davranış biçimleri (metotlar) eklemek mümkün hale gelmektedir.

Ekran çıktısı aşağıdaki şekilde olacaktır:

```
HP Printer is printing...
LexMark Printer is printing...
```

```
HP Printer is faxing...
LexMark Printer can not fax! (not implemented)
```

Bu örnekte görüldüğü gibi Printer altsınıflarına (HPPrinter, LexMarkPrinter) sendFax() gibi bir metot eklemeden ya da sınıfların yapısını değiştirmeden Visitor tasarım şablonunu kullanarak altsınıflara faks gönderebilme yeteneği kazandırdık. Burada dikkatimizi çeken ekran çıktısının en son satırıdır. LexMark tipi bir yazıcı faks gönderme özelliğine sahip olmadığı için FaxVisitor.visit(LexMarkPrinter) metodunda bu tip yazıcı için faks gönderme mantığı bulunmamaktadır. Bu şekilde faks gönderme özelliğine sahip olmayan yazıcılar için gerekli visit() metodu boş bırakılır.

Visitor tasarım şablonu ne zaman kullanılır?

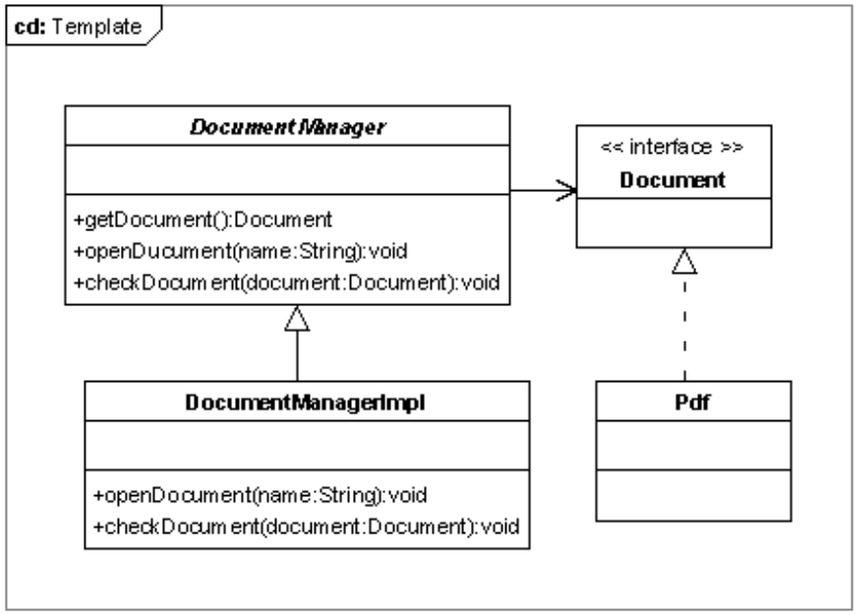
Sınıf hirarşisinin değişmediği yapılarda yeni metotların eklenmesi için Visitor tasarım şablonu kullanılır. Sınıf yapılarının değişmesi Visitor sınıflarının da değiştirilmelerini gerektirir ve bu zaman alıcı bir işlem haline gelebilir. Bu gibi durumlarda Visitor tasarım şablonu kullanılmamalıdır.

İlişkili tasarım şablonları

Composit tasarım şablonu ile oluşturulmuş sınıf hirarşilerinde visitor tasarım şablonu kullanılarak yeni metotlar uygulanabilir. Yorumlayıcı tasarım şablonundaki soyut söz dizimi ağacı (abstract syntax tree) composite olarak implemente edilebilir.

Şablon Metodu (Template Method Pattern)

Template method ile bir algoritma için gerekli işlemler soyut olarak tanımlanır. Alsınıflar algoritma için gerekli bir ya da birden fazla işlemi kendi bünyelerinde implemente ederek, kullanılan algoritmanın kendi istekleri doğrultusunda çalışmasını sağlayabilirler.



Resim 12

Bir dökümanı edinmek için DocumentManager isminde bir sınıf kullanıyoruz. Bu sınıf bünyesinde bulunan getDocument() metodu ile istediğimiz tipte bir dökümana ulaşabiliyoruz. Böylece bir dökümanı edinme algoritmasını getDocument() metodu bünyesinde tanımlanmış olmaktadır.

```
// Kod 49

package com.pratikprogramci.designpatterns.bolum7.template;

/**
 * DocumentManager sınıfı
 *
 */
public abstract class DocumentManager {
    /**
     * Bir dökümanı edinme algoritmasini getDocument()
     * metodunda tanımlıyoruz.
     * Bir dökümanı elde edebilmek için önce o
     * dökümanın açılması (openDocument)
     * ve checkDocument() ile kontrol edilmesi
     * gerekiyor. Bu işlemler ardından
     * döküman temin edilir.
     *
     */
    public final Document getDocument(String name) {
        Document dokument = openDocument(name);
        checkDocument(dokument);
```

```
        return dokument;
    }

    /**
     * Altsınıflar tarafından istekleri
     * dogrultusunda implemente edilir.
     *
     */
    public abstract void checkDocument(Document document);

    /**
     * Altsınıflar tarafından istekleri
     * dogrultusunda implemente edilir.
     *
     * @param s
     *            Document ismi
     * @return Document Document
     */
    public abstract Document openDocument(String s);
}
```

getDocument() metodunda kullanılan openDocument() ve checkDocument() metotları aynı sınıf bünyesinde soyut metotlar olarak tanımlanmışlardır. Buna göre bir dökümanı edinme algoritması için iki işlem yapılması gerekmektedir. Lakin DocumentManager sınıfı bu işlemlerin nasıl implemente edilmesi gerektiği hakkında bilgiye sahip değildir. Daha ziyade altsınıflar openDocument() ve checkDocument() metotlarını implemente ederek, bir döküman edinme algoritmasını (getDocument()) kendi istekleri doğrultusunda şekillendirebileceklerdir.

Bu örnekte algoritmanın işleyiş tarzı tanımlanmıştır. Altsınıflar bu algoritmanın ihtiva ettiği adımları değiştiremezler, çünkü getDocument() metodu final olarak tanımlanmıştır. Final olarak tanımlanan metotlar, altsınıflar tarafından reimplemente edilemezler. Geriye kalan sadece, algoritmayı oluşturan adımların altsınıflarca implemente edilmesidir.

```
// Kod 50

package com.pratikprogramci.designpatterns.bolum7.template;

/**
 * DocumentManager sınıfı implementasyonu
 *
 */
public class DocumentManagerImpl extends DocumentManager {

    @Override
```

```
    public void checkDocument(Document dokument) {
        System.out.println("Document checked...");

    }

    @Override
    public Document openDocument(String s) {
        System.out.println("Document " + s + " opened.");
        return new Pdf();
    }
}
```

checkDocument() ve openDocument() metotlarını DocumentManagerImpl sınıfında implemente ediyoruz. Bu metotlar DocumentManager soyut sınıf genişletildiği için (extends) altsınıfta implemente edilmek zorundadır. Bu iki metot, döküman edinme (getDocument()) algoritmasında yer aladığı için, yapılan implementasyon algoritmanın çalışma tarzını yönlendirecektir.

```
// Kod 51

package com.pratikprogramci.designpatterns.bolum7.template;

public abstract class Test {
    public static void main(String[] args) {
        DocumentManager manager = new DocumentManagerImpl();
        Document document = manager.getDocument("test_dok");
    }
}
```

İmplementasyonu test etmek için Test.main() metodunu kullanıyoruz. Ekran çıktısı şöyle olacaktır:

```
Document test_dok opened.
Document checked...
```

Pekte gösterişli bir ekran çıktısı olmamakla beraber, bir dökümanın elde edilişi için gerekli algoritmayı kendi isteklerimiz doğrultusunda implemente etmeyi başardık. Bunun için yapmamız gereken işlem, algoritmayı soyut olarak tanımlamak ve ihtiva ettiği işlemleri altsınıflarda implemente etmektir.

Template Method tasarım şablonu ne zaman kullanılır?

Dinamik ve altsınıflar tarafından değiştirilebilir algoritmalar oluşturmak için Template Method tasarım şablonu kullanılır. Algoritmanın değişken bölümleri altsınıflar tarafından gereksinimleri doğrultusunda implemente edilir.

İlişkili tasarım şablonları

Template method tasarım şablonunda algoritmayı değiştirmek için kalıtım (inheritance) kullanılmaktadır. Strategy tasarım şablonu altsınıflara delege ederek, başka algoritmaların kullanılmasını sağlar.

Template Method tasarım şablonunda Factory tasarım şablonu kullanılabilir. Örneğin openDocument() metotunda bir factory kullanılarak dökümanlar oluşturulabilir.

Yorumlayıcı (Interpreter Pattern)

Kelimenin tam anlamıyla yorumlanması gereken ve belli bir gramatiğe sahip yapıların işlenmesi için yorumlayıcı tasarım şablonu kullanılır. Örneğin bu yapı bir programlama dili, bir matematiksel formül ya da regüler ifade (regular expression) olabilir. Bu tasarım şablonunun en güzel örneğini yorumlanan programlama dillerinde yazılan program kodlarını çalışır hale getiren yorumlayıcılar oluşturmaktadır. Örneğin Java sınıfları derlenmiş olsalar bile, Java sanal makinesi (JVM - Java Virtual Machine) bünyesinde yorumlanırlar taki Hotspot derleyicisi çok kullanılan kod bölümlerini derleyerek, makine koduna dönüştürünceye kadar. Java sınıflarının derlemenme işlemi sırasında da bir yorumlayıcı kullanılmaktadır. Bu yorumlayıcı kodu yorumlayarak, byte koduna dönüştürür. Bunun için yorumlayıcının bir AST (Abstract Syntax Tree) oluşturması ve dilin gramatiğine uygun şekilde bu AST nesnesini yorumlaması gerekmektedir. Yorumlayıcının görevini yerine getirebilmesi için belli bir dil gramatiğinin kullanımı zaruridir. Bu gramatik yorumlayıcı için yorumlama işlemini kalıplara sokmasını ve böylece yorumlama işlemini gerçekleştirmesini kolaylaştırmaktadır.

Şimdi yorumlayıcı tasarım şablonunu matematiksel bir işlemi ihtiva eden bir cümlenin yorumlanarak, matematiksel işlemin gerçekleştiği bir örnek üzerinde inceleyelim.

Çıkış noktamış aşağıdaki cümle olacak:

```
10 çarpı 10
```

İki rakam üzerinde işlem yapmak istediğimiz taktirde, işlemi bir gramatik ya da söz dizimi (syntax) ile ifade etmemiz gerekmektedir. Bu söz dizimi "rakam1: işlem: rakam2" şeklindedir. Burada bir söz dizimi ağacı (synxtax tree) oluşturabiliriz:

```
                çarpı
                /   \
               /     \
```

```
                10     10
```

Bu ağacın kökünü "çarpı" ifadesi oluşturmaktadır. Yorumlayıcımız bu cümleyi temel alarak, gerekli olan matematiksel işlemi gerçekleştirecek ve neticeyi ekranda görüntüleyecektir.

İlk işlem olarak bir Test sınıfı oluşturuyoruz:

```
// Kod 52

package com.pratikprogramci.designpatterns.bolum7.interpreter;

public class Test {

    public static void main(String[] args) {
        String formul = "10 çarpı 10";
        new Interpreter().interpret(formul);
    }
}
```

Yorumlama işlemini Interpreter sınıfında bulunan interpret() metodu aracılığı ile yapıyoruz.

```
// Kod 53

package com.pratikprogramci.designpatterns.bolum7.interpreter;

public class Interpreter {

    public void interpret(String formul) {

        Expression expression = parse(formul);
        expression.calculate();
    }

    private Expression parse(String formul) {
        if (formul.contains(
                OperationIdentifier.MULTIPLICATION)) {
            return new MultiplicatioExpression(formul);
        }
        return null;
    }
}


package com.pratikprogramci.designpatterns.bolum7.interpreter;
```

```
public class OperationIdentifier {

    public static final String MULTIPLICATION = "çarpı";

}
```

Parse() metodunda cümlenin hangi işlemi ihtiva ettiğini bulup, o işlemi yerine getirecek bir Expression implementasyonunu seçiyoruz. Bizim örneğimizde bu çarpma işleminde sorumlu olan MultiplicationExpression sınıfıdır.

```
// Kod 54

package com.pratikprogramci.designpatterns.bolum7.interpreter;

public class MultiplicatioExpression
        extends BaseExpression {

    public MultiplicatioExpression(String formul) {
        int[] ops = parse(formul);
        this.a = ops[0];
        this.b = ops[1];
        this.formul = formul;
    }

    private int[] parse(String formul) {

        String[] ops = formul
                .split(OperationIdentifier.MULTIPLICATION);

        return new int[] { Integer.parseInt(ops[0].trim()),
                Integer.parseInt(ops[1].trim()) };
    }

    @Override
    public void calculate() {
        System.out.println(
                this.formul + " = " + (this.a * this.b));

    }
}
```

Parse() metodunda cümle içinde yer alan rakamları tespit ettikten sonra, bu rakamları this.a ve this.b sınıf değişkenlerine atıyoruz. Böylece MultiplicationExpression sınıfı gerekli rakamları bünyesinde taşır hale geldi.

Şimdi Kod 53 de yer alan interpret() metoduna tekrar bir göz atalım. Çıkış noktamız Kod 53 de yer alan bu metotdu. Interpreter.interpret() metodunda Expression sınıfının calculate() metodunun koşturulduğunu görmekteyiz. Expression bir interface sınıf:

```
// Kod 55
package com.pratikprogramci.designpatterns.bolum7.interpreter;

public interface  Expression {

    void calculate();
}
```

Bizim örneğimizde MultiplicationExpression sınıfı bu interface sınıfın implementasyonu konumundadır. Interpreter.interpret() metodu koşturulduğunda, kod 54 de yer alan MultiplicationExpression sınıfının calculate() metodu koşturulmuş olmaktadır. Bu işlemin ardından ekran çıktısı su şekilde olacaktır:

```
10 çarpı 10 = 100
```

Bu örnek bölme işlemi için DivisinExpression, toplama işlemi için AdditionExpression ve çıkarma işlemi için SubtractionExpression isminde Expression sınıfı implementasyonları oluşturularak, diğer matematiksel işlemler yorumlanabilecek şekilde genişletilebilir.

Yorumlayıcı tasarım şablonu ne zaman kullanılır?

Belli bir söz dizimi (syntax) ya da gramatik ihtiva eden yapıların yorumlanması gerektigi durumlarda yorumlayıcı tasarım şablonu kullanılır. Yorumlayıcının işlevini yerine getirebilmesi için kullanılan söz dizimini (syntax tree) tanıması gerekmektedir.

İlişkili tasarım şablonları

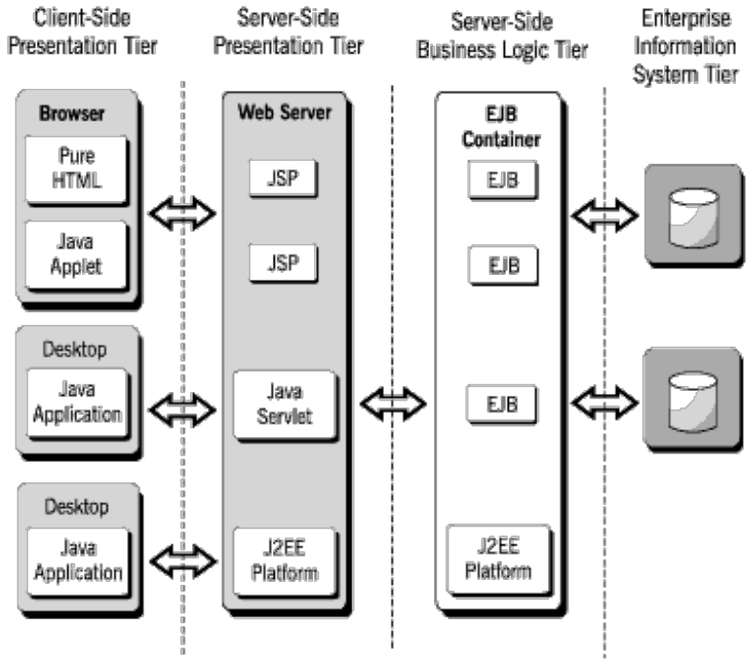
Iterator tasarım şablonu ile söz dizimi ağacı (syntax tree) döngüye alınabilir. Sineksiklet tasarım şablonu ile söz dizimi agacında yer alan ifadeler (çarpı, bölü, eksi, artı) tanımlanabilir. Söz dizimi ağacı kompozit tasarım şablonu ile ifade edilebilir. Visitor tasarım şablonu ile Expression sınıfı implementasyonlarının adedi düşürülebilir ya da sundukları davranış biçimleri değiştirilebilir hale getirilebilir.

8. Bölüm

JEE Tasarım Şablonları

Java Enterprise Edition (JEE) Platformu

Java Enterprise Edition anlamına gelen JEE birçok Java teknolojisini bir araya getiren bir Java platformudur. Bünyesinde JSP (Java Server Pages), Servlet3, EJB 3 (Enterprise Java Bean), JTA (Java Transaction API), JNDI (Java Naming and Directory Interface) ve daha birçok Java teknolojisini barındırır. Kurumsal projelerde JEE sıkça kullanılır. Tipik bir JEE mimarisini bir sonraki resimde görmekteyiz.



Resim 1

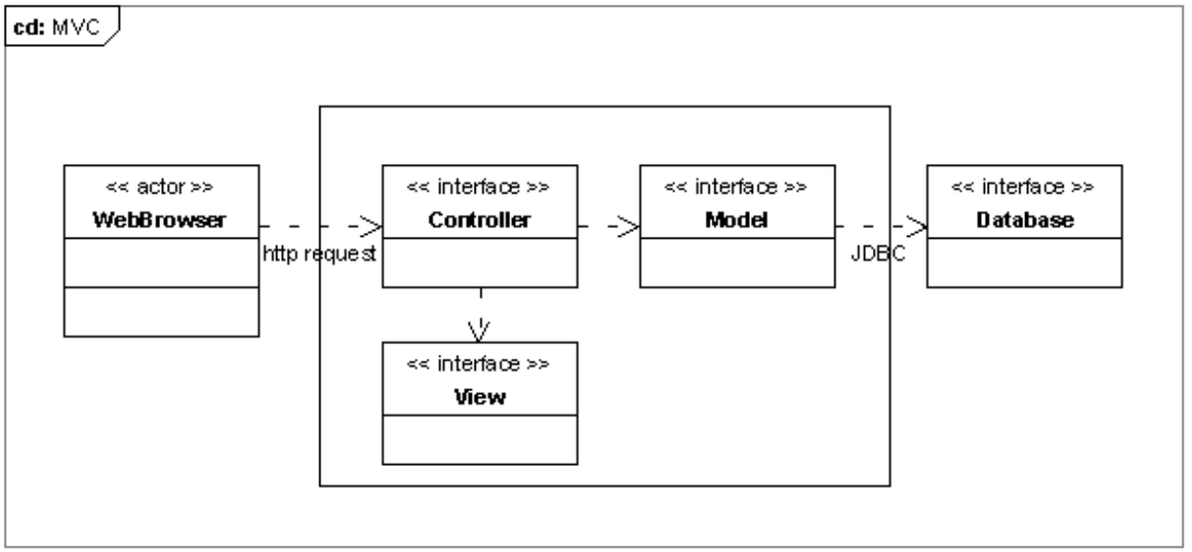
JEE projeleri değişik katmanlardan oluşur. Genel olarak üç katmanlı mimariler tercih edilmektedir. Her katmanın oluşturulması için JEE bünyesinde yer alan teknolojilerden faydalanılır. Örneğin ilk katman olan gösterim katmanı için JSP ya da Servlet teknolojisi kullanılır. İşletme katmanı (business) olarak bilinen orta katmanda EJB komponentler yer alır. JEE uygulamaları, Jboss ya da Weblogic gibi uygulama sunucusu (application container) olarak bilinen sunucular içinde çalışırlar. Uygulama sunucusu uygulama için gerekli altyapıyı sağlar (örneğin veri tabanı bağlantısı, transaksiyon yönetimi vs).

Bu bölümde JEE projelerinde kullanılan tipik JEE tasarım şanlonlarını yakından inceliyecegiz.

MVC Tasarım Tasarım Şablonu

Web tabanlı projelerin oluşturulmasında MVC (Model, View, Controller) tasarım şablonunun

kullanıldığı açık kaynaklı çatılar çok popüler olmuştur. Bunların başında Struts, Spring MVC ve JSF (Java Server Faces) gelmektedir.



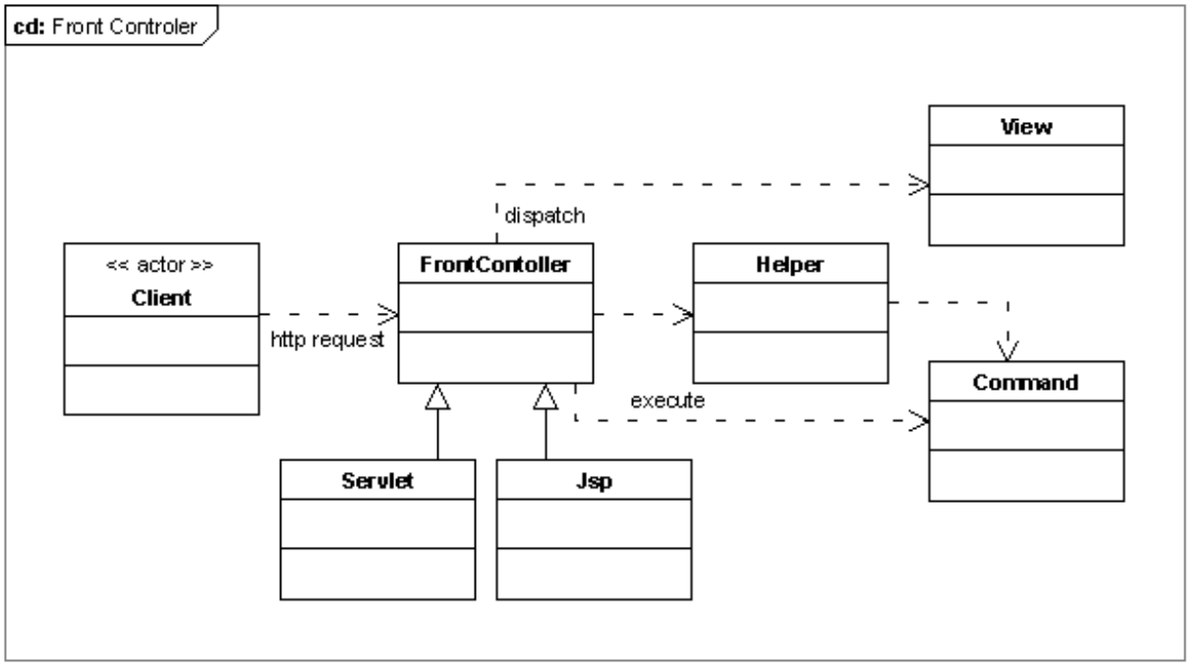
Resim 2

MVC kullanılan bir projede model sınıfları işlenen verileri ihtiva eder. Örneğin bir üyenin hesap bilgileri veri tabanından JDBC (Java Database Connectivity) ile edinildikten sonra bir model sınıfında tutulur. Model basit bir POJO (plain old java object) sınıfı olabilir. View gösterim katmanına aittir ve JSP ya da JSF teknolojileri kullanılarak programlanır. View elementinin görevi model sınıflarında yer alan verilerin kullanıcıya gösterilmesini sağlamaktır. Model değiştiği zaman, view buna göre kendisini adapte ederek, modelin ihtiva ettiği verileri gösterir. Controller sınıfları, model ve view arasındaki interaksiyonu koordine etmek için kullanılırlar.

MVC kullanılarak değişik görevleri olan katmanlar oluşturmak mümkündür. View gösterim katmanı ve model veri katmanı olabilir. MVC hakkında geniş bilgiyi Front Controller tasarım şablonu ve Otel Rezervasyon programı bölümünde detaylı inceliyecegiz.

Front Controller Tasarım Şablonu

MVC tasarım şablonu bünyesinde Controller sınıflarının hangi görevi yerine getirdiklerini gördük. MVC tasarım şablonunu implemente etmek için front controller tasarım şablonu kullanılabilir. Front controller tasarım şablonu ile sisteme yöneltilen tüm istekler (request) merkezi bir yerde toplanarak, işlem görürler.



Resim 3

Front controller ile yönlendirme ve işlem yapma fonksiyonlarının birden fazla view (bir JSP sayfası olabilir) elementine dağıtılması önlenmiş olur. Tüm view elementleri yönlendirme ve işlem yapma fonksiyonlarını ortak kullanırlar. Böylece front controller tasarım şablonunun kullanıldığı bir proje bakımı ve geliştirilmesi daha kolay bir hale gelir. Ayrıca front controller ile gösterim ve navigasyon fonksiyonları birbirinden ayrıldığı için gösterim katmanını etkilemeden navigasyon idaresi değiştirilebilir ya da tamamen yenilenebilir.

Front controller implementasyonunda Servlet ya da JSP teknolojisi kullanılabilir. Kitabın bir sonraki bölümünde yer alan otel rezervasyonu programı örneğinde front controller sınıfını Servlet teknolojisini kullanarak implemente edeceğiz. Bu sınıf aşağıda yer almaktadır.

```
// Kod 1

package com.pratikprogramci.designpatterns.bolum9
                 .otelrezervasyon.presentation.controller;

/**
 * FrontController tasarım şablonu ornegi. Http uzerinden
 * gelen tum isteklerler bu sınıf tarafından işlem gorur.
 *
 * Controller sınıfı gerekli command nesnesini bularak,
 * işlemi bu nesneye devreder ve neticeye gore gerekli jsp
 * sayfasına yonlendirme yapar.
```

```
 *
 */
public class FrontController extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * FrontController sınıfına gelen tum istekler bu metot
     * tarafından işlem gorur
     *
     */
    public void handle(final HttpServletRequest request,
            final HttpServletResponse response) {
        Logger.instance(this).debug("handle()");
        String nextPage = "";
        try {
            final RequestHelper helper = new HotelRequestHelper(
                    request, response);
            final Command command = helper.getCommand();
            nextPage = command.execute(helper);
            dispatch(request, response, nextPage);
        }
        // CommandException ve IOExcepton oluşabilir.
        // Bu durumda error.jsp sayfasına yï¿½nlendiriyoruz.
        catch (final Exception e) {
            e.printStackTrace();
            try {
                dispatch(request, response, "/error.jsp");
            } catch (final Exception e1) {
                e1.printStackTrace();
            }
        }
    }

    /**
     * Jsp sayfaları arasında yonlendirme yapmak için kullanılır.
     *
     */
    private void dispatch(final HttpServletRequest request,
            final HttpServletResponse response,
            final String page)
                    throws ServletException, IOException {
        Logger.instance(this).debug("dispatch()");
        final RequestDispatcher dispatcher = getServletContext()
                .getRequestDispatcher(page);
        dispatcher.forward(request, response);
    }
```

```
    /**
     * FrontController bir servlet sınıfı olduğu için GET
     * metodu ile gelen tum istekler bu metot tarafından
     * işlem gorur.
     */
    @Override
    public void doGet(final HttpServletRequest request,
            final HttpServletResponse response) {
        Logger.instance(this).debug("doGet()");
        handle(request, response);
    }

    /**
     * FrontController bir servlet sınıfı olduğu için POST
     * metodu ile gelen tum istekler bu metot tarafından
     * işlem gorur
     */
    @Override
    public void doPost(final HttpServletRequest request,
            final HttpServletResponse response) {
        Logger.instance(this).debug("doPost()");
        handle(request, response);
    }
}
```

FrontController isminde bir sınıf tanımlıyoruz. Bu sınıf HttpServlet sınıfını genişlettiği (extends HttpServlet) için bir Servlet sınıfı haline gelir. Bir servlet sınıfının kullanıcıdan gelen GET ve POST metotlarına cevap vermek üzere doGet() ve doPost() isimlerinde iki metoda sahip olması gerekmektedir. FronController sınıfında doGet() ve doPost() metodları içinde handle(request, response) metodunu kullanarak gelen isteği (http request) merkezi bir metoda yönlendiriyoruz. Otel rezervasyonu programında kullandığımız bu front controller sınıfı bir Helper sınıfı yardımı ile üyenin isteğine cevap vermek üzere gerekli Command (komut) nesnesini edinir. Command bünyesinde gerekli business metodları çalıştırılır ve oluşan sonucun gösterimi için bir sonraki sayfa (view) geri verilir. FrontController sahip olduğu dispatch() metodu yardımı ile kontrolü RequestDispatcher sınıfına verir ve bir sonraki jsp sayfasının gösterilmesini sağlar.

Front controller tasarım şablonunun sağladığı diğer bir avantaj da merkezi hata yönetimi (exception handling) yapılmasını kolaylaştırıyor olmasıdır. handle() metodunda yer alan bir try-catch bloğu ile altkatmanlarda oluşan hatalar yakalanır ve örneğin log4j ile bir log dosyasına yazılır ya da bir veri tabanına daha sonra incelenmek üzere eklenir.

FrontController sınıfını bir web uygulaması içinde kullanabilmek için aşağıdaki şekilde web.xml kaydının yapılması gerekmektedir.

```
// Kod 2

<?xml version="1.0" encoding="UTF-8"?>

<web-app>

    <servlet>
        <servlet-name>FrontController</servlet-name>
        <servlet-class>
            com.pratikprogramci.designpatterns.bolum9
                .otelrezervasyon.presentation.controller
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>FrontController</servlet-name>
        <url-pattern>/FrontController/*</url-pattern>
    </servlet-mapping>

    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>
</web-app>
```

Web.xml içinde elementini kullanarak FrontController isminde bir servlet tanımlıyoruz. Bu servlet http://localhost/FrontController adresinden erişilebilir hale gelir. <servlet-mapping> elementi ile http://localhost/FrontController adresine yönlendirilmiş tüm istekler (request) FrontConller sınıfına iletilir. GET metodu ile gelen istekler FrontController.doGet(), POST metodu ile gelen istekler FrontController.doPost() metoduna yönlendirilir. Aşağıda yer alan html formunda post metodu kullanılmaktadır. Ara butonuna tıklandığında bu istek (request) /FrontController sınıfına iletilir.

```
// Kod 3

<form name="form1" method="post" action="FrontController">
<table width="300" border="0" cellspacing="0" cellpadding="2">
    <tr>
        <td colspan="2"><img src="pics/arama.jpg" width="305" height="40" /></td>
        </tr>
    <tr>
        <td>
        <div align="left"><span class="text10 Stil1">Ülke:</span></div>     </td>
        <td><span class="text10"> <input type="Text" name="country"
            size="15"
            class=Stil1 maxlength="15" value="" > </span></td>
```

```
    </tr>
    <tr>
        <td><span class="Stil1">Sehir:</span></td>
        <td><input type="text" name="city" size="15" class=Stil1
            maxlength="50" value="" />
          <input type="submit" name="Submit2" value="   Ara   "
            class=Stil1 />
          <input type="hidden" name="action" value="ara" /></td>
    </tr>
    <tr>
        <td> </td>
        <td> </td>
    </tr>
</table>
</form>
```

Front Controller tasarım şablonu ne zaman kullanılır?

- Sisteme ulaşan isteklerin (http request) merkezi bir yerde toplanıp, işlenmesi gerektiği durumlarda Front Controller kullanılır.
- JSP sayfalarında navigasyon ve diğer işlemler için oluşabilecek kod tekrarlarını önlemek için Front Controller kullanılır.
- Gösterim katmanını oluşturan JSP sayfalarında yer alan verinin gösterilmesi için gerekli koddan daha fazla kodun yer almasını engellemek için Front Controller kullanılır. JSP sayfaları sadece verileri göstermek için programlanmalıdır. Bir JSP sayfası örneğin kesinlikle JDBC ile veri tabanına bağlanıp, veri edinmemeli ya da veri üzerinde işlem yapmamalıdır. Bu model komponentlerinin görevidir.
- Güvenlik uygulamaları yapabilmek için merkezi bir giriş noktası gerektiği durumlarda Front Controller kullanılır.

İlişkili tasarım şablonları

Intercepting filter ve front controller tasarım şanlonları veri akışını merkezi bir yerden kontrol etmek amacıyla kullanılır. View Helper tasarım şablonu ile kullanılan business metotlar helper sınıflarında bir araya getirilir. Front controller helper sınıflarını kullanarak, gerekli işlemlerin yapılmasını sağlar.

DAO (Data Access Objects) Tasarım Şablonu

Birçok programın var olma nedeni veriler üzerinde işlem yapmak, verileri veri tabanlarında depolamak ve bu verileri tekrar edinmektir. Bu böyle olunca, verilerin program tarafından nasıl

veri tabanlarında tutulduğu ve tekrar edinildiği önem kazanmaktadır. Data Access Objects (DAO) tasarım şablonu ile kullanılan veri tabanına erişim ve veri depolama-edinme işlemi daha soyutlaştırılarak, diğer katmanların veri tabanına olan bağımlılıkları azaltılır. DAO ile diğer katmanlar etkilenmeden veri tabanı değiştirilebilir. Daha öncede belirttiğim gibi, amacımız birbirini kullanan ama birbirine bağımlılıkları çok az olan katmanlar oluşturmak ve gerekli olduğu zaman bir katmanı, diğer katmanlar etkilenmeden değiştirebilmek olmalıdır. Katmanlar arası bağimlılık interface sınıfları üzerinden olduğu sürece bu amacımıza her zaman ulaşabiliriz.

Belki inanmayabilirsiniz, lakin aşağıda yer alan jsp sayfasına birçok kurumsal denebilecek önemli projelerde bile rastlamak mümkündür. Zaman yetersizliğinden dolayı birçok programcı, kendilerine verilen görevleri aşağıdaki şekilde çözmek zorunda kalabilirler. Bu programcının hatası değildir! Proje yöneticisinin yapılacak görevler için yeterince zaman ayrılmasına dikkat etmesi gerekmektedir. Doğru tahminler üzerinde kurulu olmayan bir proje planının çalışmamazlılığının acısı programcı ekipten çıkartılmamalıdır!

```
<%
    Connection con = getConnection();
    PreparedStatement pstmt = con.prepareStatement("select.....");
    ResultSet rs = pstmt.executeQuery();
    While(rs.next()) {
        Out.print(rs.getString(1));
    }
    rs.close();
    pstmt.close();
    con.close();
%>
```

Bu örnekte yapılmaması gereken hatalar bulunmaktadır. Öncelikle bunları inceliyelim:

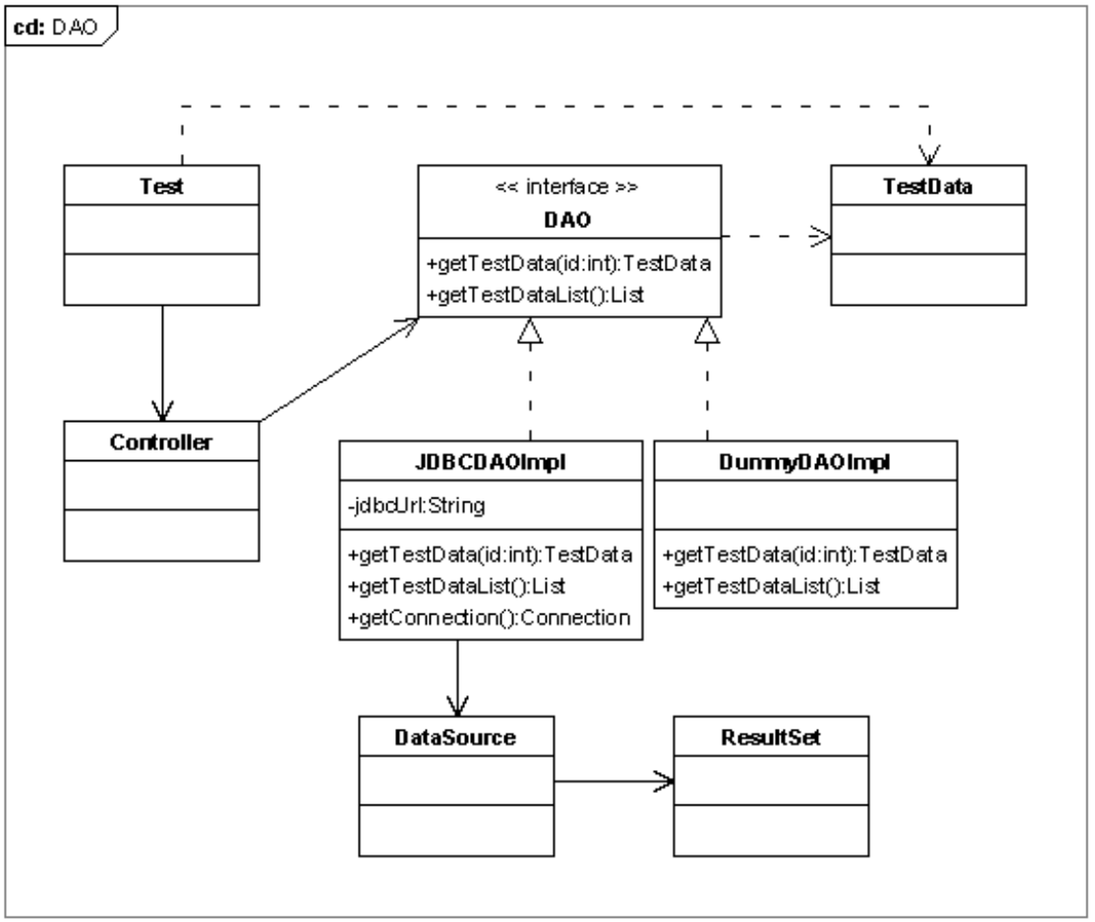
Bir JSP sayfası kesinlikle veri tabanına bağlanıp, verileri edinmemelidir. JSP sayfası başka bir katmandan edinilen verileri sadece göstermekle yükümlüdür. Bir JSP sayfası içine yukarıda yer aldığı şekilde gösterim mantığı harici başka kod eklendiğinde, JSP sayfasının bakımı ve geliştirilmesi güçleşir. Try-catch-finally bloğu kullanılmamıştır. Veri tabanına bağlanmak için bir Connection nesnesi oluşturulmakta ve con.close() ile kapatılmaktadır. getConnection() operasyonundan sonra bir hata (Exception) oluştuğunda, con.close() satırına hiçbir zaman ulaşılamıyacak ve veri tabanına olan baglantı kapatılamıyacaktır! Veri tabanına olan bağlantı con.close() ile kapatılmadığı durumunda, diğer kullanıcılar tarafından kullanılamaz hale gelir. Veri tabanında yeralan veriler bir ORM (object relational mapping) çatı ile nesnelere dönüştürülmeli ve bu nesneler kullanılmalıdır.

JSP sayfasının görevi sadece verileri kullancıya göstermek ise, veri tabanına bağlanma ve diğer işlemleri bir Controller sınıfına devretmelidir. Örnek aşağıdaki şekilde değiştirilebilir:

```
<%
    Controller controller = new ApplicationController();
    List list = controller.getMemberList();

    for(int i=0; i < list.size(); i++) {
        Out.print(((Member)list.get()).getName());
        Out.print("<p>");
    }
%>
```

ApplicationController sınıfı arka planda JDBC ya da başka bir teknoloji kullanarak gerekli verileri veri tabanından edinir. JSP sayfası getMemberList() metodu üzerinden bu verilere ulaşır.

Bu açıklamaların ardından DAO tasarım şablonu kullanımını yakından inceliyelim.



Resim 4

UML diyagramında yer alan Test sınıfı gösterim katmanında yer alan bir JSP sayfası olarak düşünülebilir. Bu sınıfın görevi veri tabanındaki test_data tablosunda yer alan verileri okuyarak, ekranda görüntülemektir.

Test_data tablosu aşağıdaki yapıya sahiptir:

id	test1	test2
1	Abc	def
2	123	455
3	xxx	yyy

Bu tablo anlam taşıyan veriler ihtiva etmese de, DAO tasarım şablonunun çalışma tarzını göstermek açısından yeterli olacaktır. Test_data tablosunun her satırını (record) bir nesne bünyesinde tutabilmek için TestData isminde bir sınıf tanımlıyoruz:

```
// Kod 4

package com.pratikprogramci.designpatterns.bolum8.dao;

/**
 * TestData sınıfı
 *
 */
public class TestData {
    private String test1;
    private String test2;
    private int id;

    public int getId() {
        return id;
    }

    public void setId(final int id) {
        this.id = id;
    }

    public String getTest1() {
        return test1;
    }

    public void setTest1(final String test1) {
```

```
        this.test1 = test1;
    }

    public String getTest2() {
        return test2;
    }

    public void setTest2(final String test2) {
        this.test2 = test2;
    }
}
```

TestData sınıfında, test_data tablosunun her kolonunda yer alan verileri tutabilmek için id, test1 ve test2 isimlerinde değişkenler tanımlıyoruz. id int, test1 ve test2 String veri tipindendir. Test sınıfı ApplicationController sınıfından sadece TestData nesnelerini alacak ve bu nesnelerin sahip olduğu değerleri ekrada görüntüleyecektir.

Test sınıfı aşağıdaki yapıya sahiptir:

```
// Kod 5

package com.pratikprogramci.designpatterns.bolum8.dao;

import java.util.List;

/**
 * DAO tasarım şablonu için oluşturulan Test sınıfı.
 *
 */
public class Test {

    /**
     * İki değişik DAO implementasyon sınıfını test eder.
     *
     * @param args
     */
    public static void main(final String[] args) {
        // TODO: JdbcDAOImpl implementasyonunun
        // çalışabilmesi için bilgisayarinizda
        // bir mysql sunucusunun çalısması gerekmektedir.
        // Detaylar için JdbcDAOImpl sınıfına bakınız.
        final ApplicationController controller = new ApplicationController(
                new JdbcDAOImpl());
        List<TestData> list = controller.getTestDataList();
```

```
        for (int i = 0; i < list.size(); i++) {
            final TestData data = list.get(i);
            System.out.println(data.getTest1());
            System.out.println(data.getTest2());
        }

        controller.setDao(new DummyDAOImpl());

        list = controller.getTestDataList();

        for (int i = 0; i < list.size(); i++) {
            final TestData data = list.get(i);
            System.out.println(data.getTest1());
            System.out.println(data.getTest2());
        }
    }
}
```

Test sınıfı, ApplicationController sınıfı üzerinden tüm TestData nesnelerini ihtiva eden bir liste edinir. Test sınıfı kendi başına JDBC ya da başka bir teknoloji kullanarak veri tabanına bağlanıp, verileri edinmiyor. Bu şekilde olması, Test sınıfını JDBC ya da kullanılan başka bir API (Application Programming Interface) ye bağımlı kılardı ve değiştirilmesi ve bakımı çok zorlaşırdı. Bir ApplicationController nesnesi oluşturmak için konstrüktör parametresi olarak bir DAO interface implementasyon sınıfı kullanmamız gerekiyor. DAO interface sınıfı aşağıdaki yapıya sahiptir:

```
//Kod 6

package com.pratikprogramci.designpatterns.bolum8.dao;

import java.util.List;

/**
 * DAO tasarım şablonu için bir interface sınıf
 * tanımlıyoruz.
 *
 */
public interface DAO {
    /**
     * Bilgibankasindan testdata listesini almak
     * için tanimlanan metot.
     *
     * @return List<TestData> testdata listesi
     */
    List<TestData> getTestDataList();
```

```
    /**
     * ID kolon değeri verilen bir testdata
     * recordunu bulmak için kullanılan metot.
     *
     * @param id
     *            TestData record id
     * @return TestData bulunan testdata
     */
    TestData getTestData(int id);
}
```

DAO interface sınıfı getTestDataList() ve getTestData() isimlerinde iki metot tanımlar. getTestDataList() metodu ile bir veri tabanında yer alan tüm test data veriler edinilir. getTestData() ile id si belli bir test data elde edilir. ApplicationController sınıfı sadece bu interface sınıfını kullanacak şekilde programlanır. Nasıl olduğunu daha sonra görecegiz.

Test.main() metodunun ilk bölümünde JdbcDAOImpl ikinci bölümünde DummyDAOImpl sınıflarını kullanarak ApplicationController nesneleri üretiyoruz. ApplicationController sınıfı aşağıdaki yapıya sahiptir:

```
// Kod 7

package com.pratikprogramci.designpatterns.bolum8.dao;

import java.util.List;

/**
 * DAO tasarım şablonu için bir interface sınıf
 * tanımlıyoruz.
 *
 */
public interface DAO {
    /**
     * Bilgibankasindan testdata listesini almak
     * için tanimlanan metot.
     *
     * @return List<TestData> testdata listesi
     */
    List<TestData> getTestDataList();

    /**
     * ID kolon değeri verilen bir testdata
     * recordunu bulmak için kullanılan metot.
     *
```

```
     * @param id
     *            TestData record id
     * @return TestData bulunan testdata
     */
    TestData getTestData(int id);
}
```

ApplicationController sınıfında dao isminde ve DAO tipinde bir sınıf değişkeni tanımlıyoruz. Test sınıfı tarafından controller.getTestData() ve controller.getTestDataList() metodları kullanıldığında, controller sınıfı bünyesinde bu istekler DAO.getTestData() ve DAO.getTestDataList() metotlarına delege edilecektir. DAO interface sınıfını kullanarak değişik implementasyon sınıfları oluşturduğumuz taktirde, Test sınıfının veri edinme tarzını istediğimiz şekilde değiştirme imkanına sahip olabiliriz. Yapmamız gereken tek şey bir ApplicationController nesnesi oluştururken, sınıf konstrüktörüne istediğimiz DAO implementasyon sınıfını belirmek olacaktır:

```
ApplicationController controller =
            new ApplicationController(new JdbcDAOImpl());
List<TestData> list =  controller.getTestDataList();
```

Yukarıda yer alan satırda dolaylı olarak (controller getTestDataList() üzerinden) JdbcDAOImpl sınıfının getTestDataList() metodu işlem görür. JdbcDAOImpl sınıfı DAO sınıfını implemente eder ve aşağıdaki yapıya sahiptir:

```
// Kod 8

package com.pratikprogramci.designpatterns.bolum8.dao;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

/**
 * DAO interface sınıfını implemente eden altsınıf.
 * JDBC teknolojisini kullanarak DAO interface sınıfını
 * implemente eder.
 *
 */
public class JdbcDAOImpl implements DAO {
```

```
    /**
     * JDBC Mysql URL
     */
    private final String url = "jdbc:msql://localhost:4333/"
            + "test_db";

    /**
     * ID kolon değeri verilen bir testdata recordunu
     * bulmak için kullanılan
     * motod. JDBC kullanır.
     *
     * @param id
     *            TestData record id
     * @return TestData bulunan testdata
     */
    @Override
    public TestData getTestData(final int id) {
        Connection con = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;
        TestData data = null;
        try {
            con = getConnection();
            pstmt = con.prepareStatement(
                    "select test1, test2, id"
                            + " from test_db where id=?");
            pstmt.setInt(1, id);
            rs = pstmt.executeQuery();
            data = new TestData();
            if (rs.next()) {
                data.setTest1(rs.getString(1));
                data.setTest1(rs.getString(2));
                data.setId(rs.getInt(3));
            }
            rs.close();
            pstmt.close();
        } catch (final Exception e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        } finally {
            if (con != null) {
                try {
                    con.close();
                } catch (final SQLException e) {
                    throw new RuntimeException(e);
                }
            }
```

```
        }
        return data;
    }

    /**
     * Veri tabanından testdata listesini almak
     * için tanimlanan metot. JDBC kullanır.
     *
     * @return List<TestData> testdata listesi
     */
    @Override
    public List<TestData> getTestDataList() {
        Connection con = null;
        final ArrayList<TestData> list = new ArrayList<TestData>();
        PreparedStatement pstmt = null;
        ResultSet rs = null;
        try {
            con = getConnection();
            pstmt = con.prepareStatement(
                    "select test1, test2, id"
                            + " from test_db");
            rs = pstmt.executeQuery();

            while (rs.next()) {
                final TestData data = new TestData();
                data.setTest1(rs.getString(1));
                data.setTest1(rs.getString(2));
                data.setId(rs.getInt(3));
                list.add(data);
            }
            rs.close();
            pstmt.close();
        } catch (final Exception e) {
            throw new RuntimeException(e);
        } finally {
            if (con != null) {
                try {
                    con.close();
                } catch (final SQLException e) {
                    throw new RuntimeException(e);
                }
            }
        }
        return list;
    }

    /**
```

```
     * Veri tabanına bağlanmak için gerekli Connection
     * nesnesini oluşturur.
     *
     * @return Connection db bağlantısı
     */
    public Connection getConnection() {
        Connection con = null;
        try {
            Class.forName("com.mysql.jdbc.Driver");
            con = DriverManager.getConnection(url, "test",
                    "test");
        } catch (final Exception e) {
            e.printStackTrace();
            throw new RuntimeException(e);
        }
        return con;
    }

}
```

JdbcDAOImpl sınıfı isminden de belli olduğu gibi JDBC teknolojisini kullanarak verilere ulaşmaktadır.

```
controller.setDao(new DummyDAOImpl());
list =  controller.getTestDataList();
```

Yukarıda yer alan satırda dolaylı olarak (controller getTestDataList() üzerinden) DummyDAOImpl sınıfının getTestDataList() metodu işlem görür. DummyDAOImpl sınıfı DAO sınıfını implemente eder ve aşağıdaki yapıya sahiptir:

```
// Kod 9

package com.pratikprogramci.designpatterns.bolum8.dao;

import java.util.ArrayList;
import java.util.List;

/**
 * DAO interface sınıfı için dummy implementasyon örnegi.
 *
 */
public class DummyDAOImpl implements DAO {

    /**
     * Bir dummy TestData oluşturarak geri verir. JDBC örneğinde bu metot veri
```

```
     * tabanına bağlanarak edindiği veriler ile bir TestData oluşturur.
     */
    @Override
    public TestData getTestData(final int id) {
        final TestData data = new TestData();
        data.setTest1("test1");
        data.setTest2("test2");
        return data;
    }

    /**
     * Dummy TestData nesneleri oluşturarak bir ArrayList içinde geri verir.
     * JDBC örneğinde bu metot veri tabanına bağlanarak, edindiği verilerde
     * TestData nesneleri oluşturur ve bir ArrayList içinde geri verir.
     */
    @Override
    public List<TestData> getTestDataList() {
        final ArrayList<TestData> list = new ArrayList<TestData>();

        TestData data = new TestData();
        data.setTest1("test1");
        data.setTest2("test2");

        list.add(data);

        data = new TestData();
        data.setTest1("test11");
        data.setTest2("test22");

        list.add(data);

        return list;
    }
}
```

DummyDAOImpl bünyesinde rastgele TestData nesneleri oluşturulur. Örneğin DummyDAOImpl sınıfı Junit testleri bünyeside DAO implementasyon sınıfı olarak kullanılabilir. DummyDAOImpl implementasyonu veri tabanına ihtiyaç duymadığı için gerekli TestData nesneleri kolay bir şekilde oluşturulup, diğer katmanlar tarafından kullanılabilir.

Örnekte görüldüğü gibi Test ve ApplicationController sınıfları etkilenmeden DAO sınıf implementasyonunu değiştirerek, veri edinme metotlarını değiştirebiliyoruz. DAO tasarım şablonunun kullanılma amacı işte budur! DAO kullanılarak değişik tarzda verilere ulaşım mekanizmaları implemente edilebilir. Diğer katmanlar etkilenmeden değişik DAO

implementasyonları kullanılabilir.

DAO tasarım şablonu ne zaman kullanılır?

- Veri tabanlarında yer alan verilere ulaşmak için DAO kullanılır.
- Verilere ulaşmayı sağlayan katman ile verileri kullanan katmanlar arasındaki bağımlılığı azaltmak ve bu katmanların hangi veri tabanı tipinin kullanıldığını bilmesine gerek kalmadan işlem yapabilmelerini kolaylaştırmak için DAO kullanılır.
- Değişik tipte veri tabanı sistemlerine (örneğin LDAP, XML, RDBMS) aynı metotları kullanarak erişimi sağlamak için DAO kullanılır. Böylece veri tabanı operasyonları için kullanılan metotlar standardize edilmiş olur ve tüm sistem içinde tekil olarak kullanılabilirler.

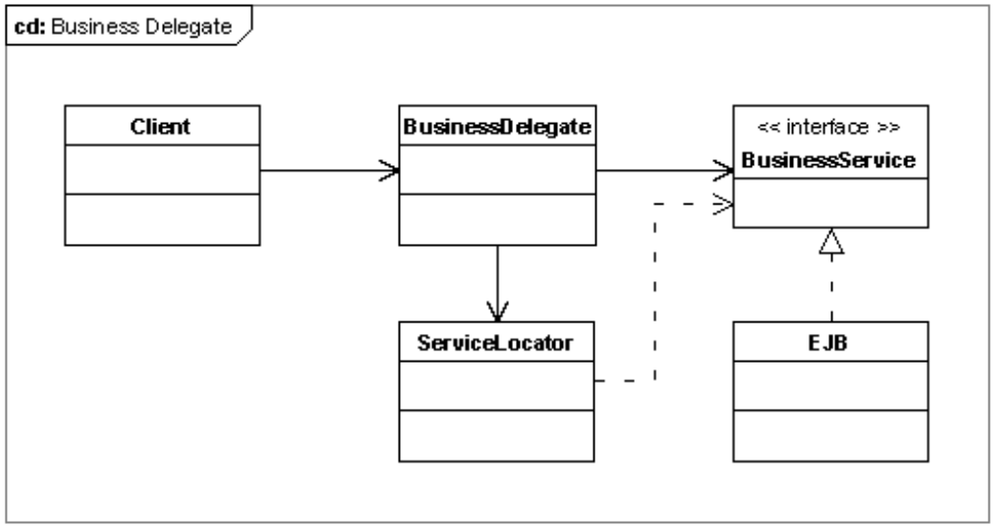
İlişkili tasarım şablonları

DAO implementasyon sınıfları DTO (Data Transfer Objects) tasarım şablonunu kullanarak verilerin diğer katmanlar tarafından kullanılmasını sağlarlar. Fabrika tasarım şablonu kullanılarak istenilen tipte DAO implementasyon sınıfları oluşturulabilir.

Servise Yönlendirme Tasarım Şablonu (Business Delegate)

Modern yazılım sistemleri birden fazla katmandan oluşur. Bu katmanlar her zaman aynı sunucu üzerinde mevcut olmayabilir. Bu durumda bir katmandan diğer katmana ulaşmak için remote call olarak isimlendirilen RMI operasyonları yapılır. Örneğin EJB teknolojisi ile hazırlanan komponentler birden fazla sunucu üzerinde hizmet sunabilir. Bu komponentlere bağlanıp, işlem yapabilmek için RMI kullanılır.

EJB sistemlerinde bazı işlemler bilgisayar ağı üzerinden erişim gerektirebileceği için zamanla sistem performansı negatif etkilenebilir. Bunun yanı sıra gösterim katmanında bulunan sınıflar doğrudan EJB komponentler ile interaksiyona girdikleri taktirde, gösterim katmanı ile EJB lerden oluşan İşletme (business) katmanı arasında sıkı bir bağ oluşur. EJB komponentler üzerinde yapılan değişiklikler gösterim katmanını etkiler. Bu bağı azaltmak ve RMI performansını artırmak için business delegate tasarım şablonu kullanılır.



Resim 5

Business delegate tasarım şablonu ile karmaşık yapıda olabilecek işletme katmanı ile gösterim katmanı arasına, gösterim katmanının isteklerini işletme katmanına delege edecek BusinessDelegate isminde bir sınıf yerleştirilir. Bu sınıfın öncelikli görevi, işletme katmanında yer alan EJB gibi servis komponentlerinin lokalizasyonu için gerekli lookup işlemlerini gösterim katmanından gizlemektir. Bu işlem için BusinessDelegate sınıfı servis lokalizasyonu (service locator) tasarım şablonundan faydalanır. Bunun yanı sıra BusinessDelegate İşletme katmanında oluşan tüm hataları (exception handling) yakalayarak, gösterim katmanı için daha anlaşılır bir hale getirir. Örneğin EJB komponentleri ile çalışırken java.rmi.RemoteException oluşabilir. Gösterim katmanının catch(RemoteException) şeklinde oluşan remote hataları yakalamaya çalışması, gösterim katmanını EJB teknolojisine bağımlı kılar. BusinessDelegate kullanıldığı taktirde, servis katmanı ile gösterim katmanı arasında "loose coupling" olarak tabir edilen esnek bir bağ oluşur.

Business delegate tasarım şablonunun beraberinde getirdiği diğer bir avantaj ise, bir hızlı önbellek (caching) mekanizması kullanarak ağ (network) üzerinden yapılan işlemlerin neticelerini saklamak ve tekrar RMI operasyonuna gerek kalmadan gösterim katmanı tarafından kullanılmasını sağlıyor olmasıdır. Gerekli verilerin RMI üzerinden başka bir sunucudan edinilmesi yerine BusinessDelegate sınıfının sahip olduğu önbellekten alınması, sistem performansını büyük ölçüde artırır.

Resim 5 de yer alan UML diyagramında da görüldüğü gibi BusinessDelegate sınıfı Service locator tasarım şablonunu kullanarak, işletme katmanında bulunan EJB ve diğer servis veren sınıfların lokalize eder. Service locator tasarım şablonunu bir sonraki bölümde inceliyeceğiz.

Business Delegate tasarım şablonunun nasıl kullanılabileceğini bir EJB örneğinde görelim. İlk işlem

olarak EJB komponentini oluşturuyoruz.

```
// Kod 10

package com.pratikprogramci.designpatterns.bolum8.delegate.ejb;

import javax.ejb.Stateless;

/**
 * Stateless EJB komponenti.
 *
 */
@Stateless
public class ServiceBean implements ServiceBeanRemote {

    private static final long serialVersionUID = 1L;

    @Override
    public String getValue() {
        return "value";
    }
}
```

Servis sunucusu olan ServiceBean stateless tipi bir EJB komponentidir. @Stateless annotasyonu kullanılarak herhangi bir Java sınıfı stateless komponent olarak tanımlanabilir. ServiceBean sınıfı bünyesinde bulunan getValue() metodu, gösterim katmanı tarafından kullanılacak olan metotdur.

Bir EJB komponente RMI üzerinden ulaşabilmek için bir Remote Interface sınıfının tanımlanması gerekmektedir:

```
// Kod 11

package com.pratikprogramci.designpatterns.bolum8.delegate.ejb;

import java.io.Serializable;

import javax.ejb.Remote;

/**
 * Service için remote interface sınıfı.
 *
 */
@Remote
public interface ServiceBeanRemote extends Serializable {
    public String getValue();
```

```
}
```

EJB komponenti kullanmak isteyen bir sınıf, ServiceBeanRemote interface sınıfını implemente eden bir stub nesne edinmek zorundadır. Bu stub nesne, EJB komponentin deploy edildiği uygulama sunucusu tarafından (Jboss, Weblogic vs...) otomatik olarak oluşturulur. Bu stub nesnesini EJB sınıfının vekili (proxy) olarak düşünebilirsiniz. Uygulama sunucusu tarafından oluşturulan bu proxy otomatik olarak bizim sunduğumuz ServiceBeanRemote interface sınıfını implemente eder. Kullanıcı sınıf bu proxy nesnesinin getValue() metodunu kullandığı zaman, proxy bu isteği gerçek EJB komponentine yönlendirir. @Remote annotasyonunu kullanarak, herhangi bir interface sınıfını EJB Remote Interface haline getirebiliriz.

Şimdi business delegate tasarım şablonun uygulandığı BusinessDelegate sınıfını inceliyelim:

```
// Kod 12

package com.pratikprogramci.designpatterns.bolum8.delegate;

import com.pratikprogramci.designpatterns.bolum8
        .delegate.ejb.ServiceBeanRemote;

/**
 * Business Delegate Sınıfı
 *
 */
public class BusinessDelegate {
    /**
     * Kullanılan ejb komponenti
     */
    private final ServiceBeanRemote service;

    /**
     * Konstrüktör bunyesinde ServiceLocator sınıfı
     * kullanılarak ejb komponenti
     * oluşturuluyor.
     */
    public BusinessDelegate() {
        service = ServiceLocator.instance()
                .getService("ServiceBean/remote");
    }

    /**
     * getValue() metodu ejb komponentine delege ediliyor.
     *
```

```
     * @return String value
     */
    public String getValue() {
        return service.getValue();
    }
}
```

BusinessDelegate basit bir POJO (Plain Old Java Object) sınıftır. Gösterim katmanı tarafından gelen istekleri, EJB sınıfına delege edebilmesi için bünyesinde ServiceBeanRemote tipinde bir değişken barındırır. getValue() metodunda görüldüğü gibi, EJB komponentin getValue() metodunu kullanarak, gösterim katmanından gelen isteği EJB komponentine delege eder. Konstrüktör içinde ServiceLocator kullanılarak, EJB remote interface nesnesinin nasıl oluşturulduğunu görmekteyiz. BusinessDelegate sınıfı, ServiceLocator aracılığıyla kullanmak istediği EJB komponenti lokalize eder.

```
// Kod 13

package com.pratikprogramci.designpatterns.bolum8.delegate;

import java.util.HashMap;
import java.util.Map;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import com.pratikprogramci.designpatterns.bolum8.
            delegate.ejb.ServiceBeanRemote;

/**
 * Bir servisi lokalize etmek için kullanılan
 * ServiceLocator komponenti
 *
 */
public class ServiceLocator {

    /**
     * Oluşturulan ejb remote interface nesnesi cache
     * içinde tutulur. Böylece
     * her defasında lookup işlemi yapılmak
     * zorunda kalınmaz.
     */
    private Map<String, ServiceBeanRemote> cache;

    private InitialContext ctx;
```

```
    /**
     * Singleton tasarım şablonunu kullanarak sistemde
     * tek bir ServiceLocator
     * olacak şekilde oluşturuyoruz.
     */
    private static ServiceLocator locator = new ServiceLocator();

    private ServiceLocator() {
        try {
            /**
             * Lookup için gerekli ctx ve remote
             * interface nesnelerin tutulduğu
             * cache nesnesini oluşturuyoruz.
             */
            ctx = new InitialContext();
            cache = new HashMap<String, ServiceBeanRemote>();
        } catch (final NamingException e) {
            throw new RuntimeException(e);
        } catch (final Exception e) {
            throw new RuntimeException(e);
        }
    }

    public static ServiceLocator instance() {
        return locator;
    }

    /**
     * Lookup ismi verilen bir EJB komponentini geri verir.
     */
    public ServiceBeanRemote getService(
            final String lookup) {
        ServiceBeanRemote remote = null;
        if (cache.containsKey(lookup)) {
            remote = cache.get(lookup);
        } else {
            try {
                remote = (ServiceBeanRemote) ctx
                        .lookup(lookup);
                cache.put(lookup, remote);
            } catch (final NamingException e) {
                throw new RuntimeException(e);
            }
        }
        return remote;
    }
}
```

Bir EJB komponenti kullanabilmek için InitialContext sınıfı yardımıyla lookup işlemi yapılmak zorundadır. Bu bir nevi mahalleye gelen bir yabancının "Ahmet beyin evini arıyorum, 35 nolu evde oturuyormuş" söylemine eşittir. Lookup işlemi ile ismi belli olan bir EJB komponent lokalize edilirek EJB nin sahip olduğu remote interface nesnesi elde edilir. ServiceLocator sınıfı EJB komponentini lokalize etmek için kullanılır. ServiceLocator lookup için gerekli tüm metotları ihtiva ettiği için BusinessDelegate sınıfının spesifik lookup operasyonlar ile ugraşma zorunluğuğu ortadan kalkmaktadır.

ServiceLocator bünyesinde bir önbellek (cache) oluşturarak, her defasından lookup işleminin yapılmasını engellemiş oluyoruz. Her lookup, RMI üzerinden başka bir sunucuya bağlantı yapmak anlamına gelebildiği için önbellek içinde bulunan bir remote interface nesnesini doğrudan önbellekten alarak kullanmak, performansı artıracaktır.

ServiceLocator sınıfını singleton tasarım şablonunu kullanarak implemente ediyoruz, çünkü tüm sistem bünyesinde sadece bir ServiceLocator ve bununla bağlantılı olarak sadece bir önbelleğin olması gerekmektedir.

```
// Kod 14

package com.pratikprogramci.designpatterns.bolum8.delegate;

/**
 * Test sınıfı
 *
 */
public class Test {
    /**
     *
     * Bu test sınıfı sadece ornek olarak programlanmistir.
     * EJB 3 komponent kullanıldığı için, test sınıfının çalısması
     * imkansızdır. Örneğin calistirilabilmesi için EJB
     * komponentin JBOSS gibi bir uygulama sunucusu uzerinde deploy
     * edilmesi gerekmektedir.
     */
    public static void main(final String[] args) {
        final BusinessDelegate delegate = new BusinessDelegate();
        System.out.println(delegate.getValue());
    }
}
```

Test.main() bünyesinde bir BusinessDelegate nesnesi oluşturarak, getValue() metodunu

kullanıyoruz. Gösterim katmanını simüle eden Test sınıfı, EJB komponentin varlığından bile haberdar olmadan, servis katmanında yer alan bir EJB nin sunduğu servisi kullanabilmektedir. Eğer business delegate tasarım şablonun kullanmasaydık, Test sınıfı aşağıdakı yapıda olurdu:

```
// Kod 15

package com.pratikprogramci.designpatterns.bolum8.delegate;

import javax.naming.InitialContext;
import javax.naming.NamingException;

import com.pratikprogramci.designpatterns.bolum8.
            delegate.ejb.ServiceBeanRemote;

public class Test2
{
    public static void main(final String[] args)
            throws NamingException {
        final InitialContext ctx = new InitialContext();
        final ServiceBeanRemote remote = (ServiceBeanRemote) ctx
                .lookup("ServiceBean/remote");
        System.out.println(remote.getValue());
    }
}
```

Böyle bir implementasyonun beraberinde getirdiği sorunlar şöyle olacaktır:

- Gösterim katmanı EJB teknolojine bağımlı olacaktır, çünkü RMI operasyonlarını kullanarak EJB komponenti lokalize etmek zorunda.
- Gösterim katmanı oluşacak javax.rmi.RemoteException ile baş etmek zorundadır. Sunucuya olan baglantı koptuğunda, tekrar baglantı kurmak zorundadır.
- Gösterim katmanı caching uygulamadığı için her getValue() metodunu kullandığında, bu RMI üzerinden EJB komponentin deploy edildiği sunucuya bağlantı kurmak anlamına geldiği için yapılmak istenen işlemler uzun sürecek ve performans iyi olmayacaktır.

Görüldüğü gibi business delegate tasarım şablonu ile hem servis sağlayan katman gösterim katmanı için daha transparen bir hale dönüştürülüyor hem de RMI operasyonlarında performans artırılabiliyor.

Business Delegate tasarım şablonu ne zaman kullanılır?

- Gösterim katmanı ile işletme katmanı arasındaki bağlantı için business delegate kullanılır.
- Servis katmanında uluşan hataları, gösterim katmanı için daha anlaşılır bir hale dönüştürmek

için business delegate kullanılır.

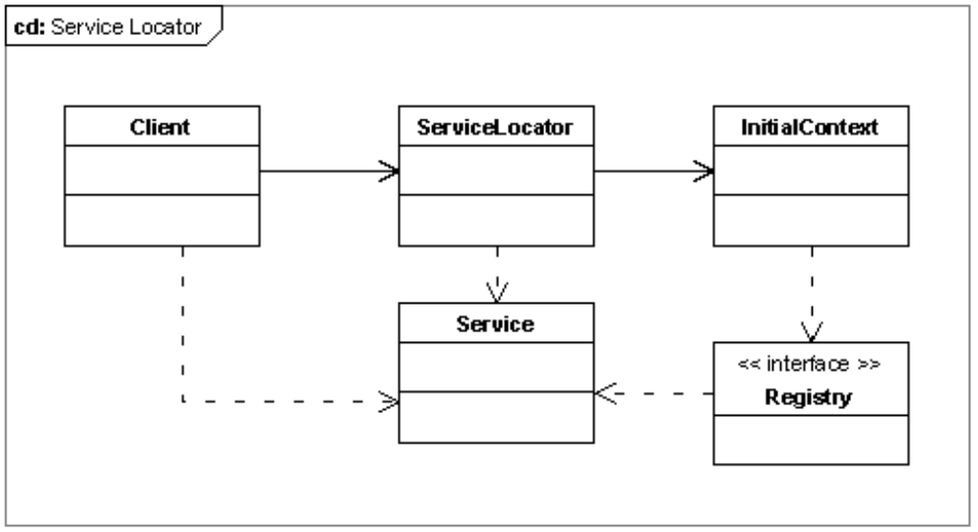
- RMI operasyonlarında performans çok önemli bir unsurdur. Business delegate caching mekanizmaları implemente ederek, performansı artırır.

İlişkili tasarım şablonları

Business delegate servis katmanında bulunan komponentleri lokalize etmek için service locator tasarım şablonunu kullanır. Business felegate bir nevi proxy nesnesi oluşturarak, gösterim katmanını servis katmanı ile bir araya getirir.

Servis Lokalizasyonu Tasarım Şablonu (Service Locator)

Business Delegate örneğinde service locator tasarım şablonunun nasıl uygulandığını görmüştük. Service locator işletme (business) katmanında bulunan komponentlerin lokalizasyonu için kullanılır.



Resim 6

JEE projelerinde kullanılan EJB gibi komponentler lokalize edilebilmeleri için merkezi bir kütükte (registry) kayıtlanırlar. Kullanıcı sınıflar JNDI API sini kullanarak, bu kütüklerde kayıtlı bulunan komponentleri edinebililer. JNDI ile lokalizasyon işlemi yapılabilmesi için InitialContext sınıfının kullanılması gerekmektedir. Resim 6 da görüldüğü gibi ServiceLocator sınıfı InitialContext yardımı ile Registry ismini taşıyan kütüğe bağlanarak, EJB komponentin remote interface nesnesini edinir.

Kullanılmak istenen komponentlerin lokalizasyonu için birçok yerde InitialContext ile çalışmak yerine, bir service locator sınıfı oluşturularak, lookup işlemleri merkezi bir yerde toplanmış olur.

Kullanıcı sınıflar InitialContext sınıfını kullanmak yerine, mevcut ServiceLocator sınıfını kullanarak, istedikleri komponentleri edinirler. Böylece lookup için kod tekrarı (code dublication) oluşmaz ve sadece bir sınıf (ServiceLocator) bu işlemden sorumlu olur.

ServiceLocator sınıfı bünyesinde bir önbellek barındırabilir. Bu önbellek içinde daha önce lookup işlemi yapıldıktan sonra oluşturulmuş olan remote interface nesneleri yer alır. InitialContext ile tekrar lookup işlemini yapmadan, istenen remote interface nesnenin önbellek içinde olup, olmadığına bakılır. Önbellek içinde bulunan bir remote interface nesnesi, yapılması pahalı olan bir RMI lookup operasyonunu engeller ve genel olarak sistem performansını artırır.

Service Locator implementasyonu için business delegate tasarım şablonuna bakınız.

Service Locator tasarım şablonu ne zaman kullanılır?

Komponentlerin lokalizasyonu ve lookup için gerekli kodun merkezi bir yerde oluşturulması için service locator tasarım şablonu kullanılır.

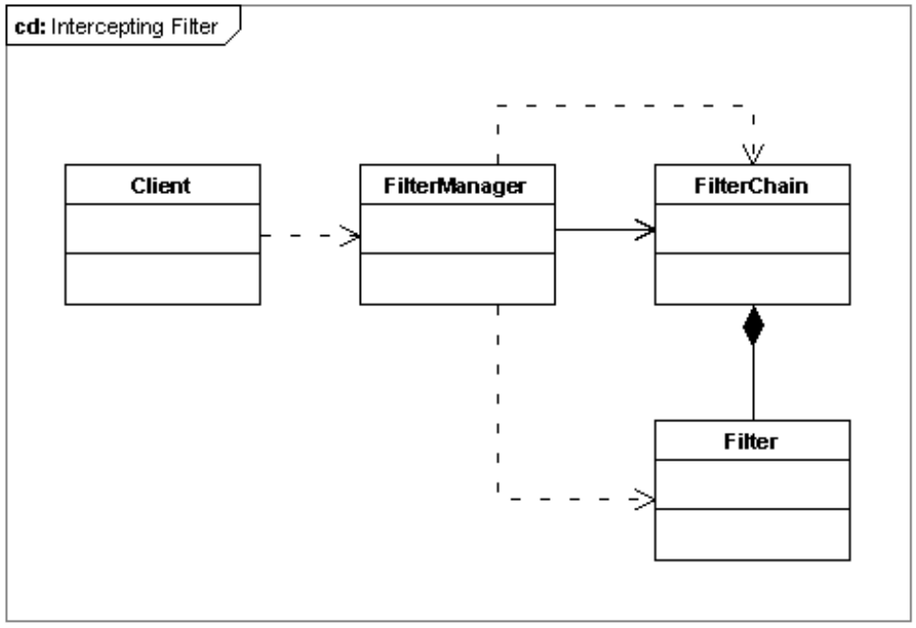
İlişkili tasarım şanlonları

Business delegate servis katmanında bulunan komponentleri lokalize etmek için service locator tasarım şablonunu kullanır. Data Access Object tasarım şablonu veri kaynaklarına (datasource) ulaşmak için service locator tasarım şablonunu kullanır.

Filtreleme Tasarım Şablonu (Intercepting Filter)

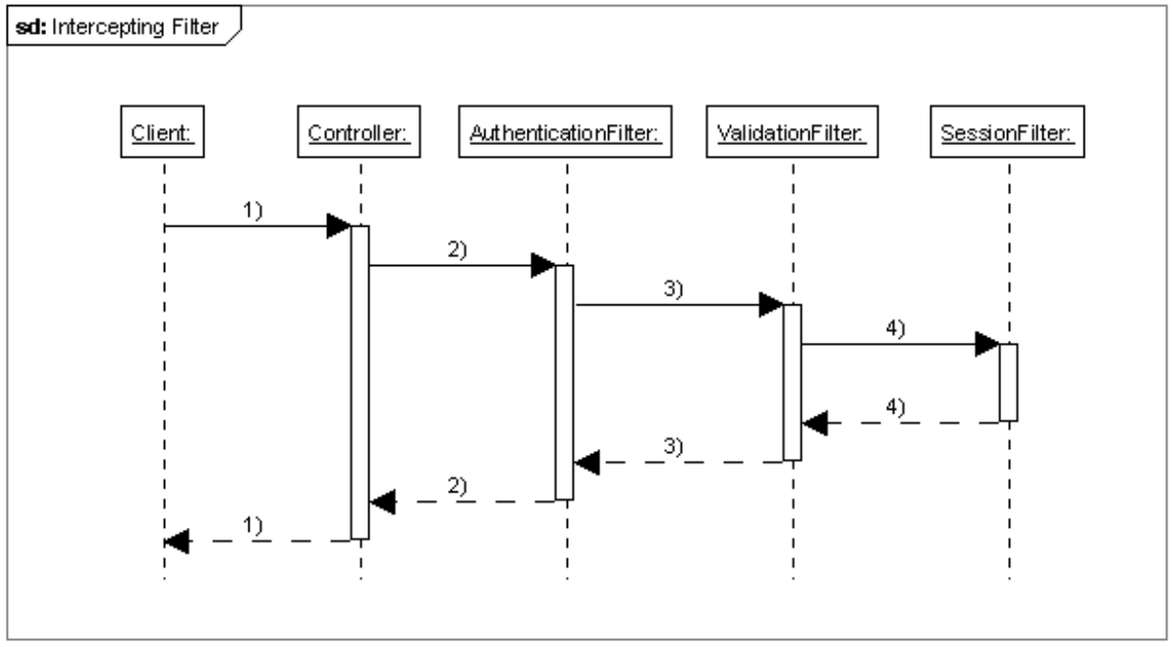
Front controller tasarım şablonunda kullanıcıdan gelen isteklerin (request) merkezi bir yerde toplanarak, işlem yapıldığını daha önce görmüştük. intercepting filter tasarım şablonu ile kullanıcının isteği (request) işleme alınmadan önce filtreler kullanılarak süzgeçten geçirilir. Örneğin bir filtre ile kullanıcının işlem öncesi login yaptığını konrol edebiliriz. Filtremiz, session (HttpServletSesion) içinde login bilgilerini bulamadığı taktirde, kullanıcıyı login sayfasına yönlendirebilir.

Aynı şekilde işlem tamamlandıktan sonra kullanıcıya gönderilecek cevap (response) filtreler yardımı ile modifike edilebilir.



Resim 7

Intercepting filter tasarım şablonunun merkezinde filtreler bulunur. Bunlar belirli özelliklere sahip normal Java sınıflarıdır. JEE dünyasında servlet filter olarak bilinen bu sınıflar, filter chaining mekanizması ile bir kolyenin üzerinde bulunan boncuklar gibi, arka arkaya dizilerek, işleme tabi tutulabilirler.



center>
Resim 8

UML diyagramında görüldüğü gibi birden fazla filtre arka arkaya dizilerek işlem yapılabilir. Sıradaki her filtre kendi görevini yerine getirdikten sonra kontrolü FilterManager yardımı ile kendinden sonraki filtreye bırakır. Eğer işlem esnasında filtrelerden birisi hatalı bir durum tespit ederse (örneğin session içinde login bilgilerinin bulunamaması) kendinden sonra gelen filtreler devreye girmeden işlemi durdurur ve filtre içinde tanımlanmış aksiyonu gerçekleştirir (örneğin login sayfasına yönlendirme).

Web uygulamalarında filtreler web.xml dosyasında tanımlanır.

Üyelerin login yaptığını kontrol etmek için AuthenticationFilter isminde bir filtre örneğini inceliyelim:

```
// Kod 15

package com.pratikprogramci.designpatterns.bolum8.intercept;

import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

/**
 * Login yapmış uye bilgilerini session içinde
 * kontrol eder. Login yapmamış bir
 * kullanıcıyı login.jsp sayfasına yonlendirir.
 *
 */
public final class AuthenticationFilter implements Filter {

    /**
     * Bir kullanıcı login yaptıktan sonra, session
     * içine USER etiketi altında
     * login ismi yerleştirilir.
     *
     * Eğer session içinde USER isminde bir
     * etiket yoksa, o zaman kullanıcı
     * login yapmamış demektir.
     */
    private static final String USER = "user";
```

```
    @Override
    public void doFilter(final ServletRequest req,
            final ServletResponse res,
            final FilterChain chain)
                    throws IOException, ServletException {

        final HttpServletRequest request =
                (HttpServletRequest) req;
        final HttpServletResponse response =
                (HttpServletResponse) res;
        final HttpSession session = request.getSession();

        if (session.getAttribute(USER) != null) {
            /**
             * Kontrolü bir sonraki filtreye vermek için
             * chain.doFilter() metodu
             * kullanılır.
             */
            chain.doFilter(req, res);
        } else {
            response.sendRedirect("login.jsp");
        }
    }

    /**
     * Filter init()
     */
    @Override
    public void init(final FilterConfig arg0)
            throws ServletException {
    }

    /**
     * Filter destroy()
     */
    @Override
    public void destroy() {
    }
}
```

Oluşturduğumuz filtrelerin javax.servlet.Filter interface sınıfını implemente etmeleri gerekmektedir. Filter interface sınıfında, kendi filtre sınıfımızın implemente etmesi gereken üç metot bulunmaktadır. Bunlar init(), doFilter() ve destroy() metotlarıdır.

init() metodu ile filtre tarafından kullanılacak kaynaklar oluşturulur. init() metodunu bir sınıf

konstrüktörü olarak düşünebilirsiniz. Filtre görevine başlamadan önce uygulama sunucusu (örneğin Tomcat) tarafından init() metodu koşturulur.

Filtrenin asıl görevi doFilter() metodu bünyesinde implemente edilir. İşlem tamamlandıktan sonra chain.doFilter() ile kontrol bir sonraki filtreye verilir. Eğer sırada başka bir filtre yoksa, uygulama sunucusu gelen isteği (request) gerçek sahibine iletir (örneğin FrontController). Örneğimizde görüldüğü gibi session içinde gerekli bilgi bulunamadığı taktirde, üye login.jsp sayfasına yönlendilir. Bu noktadan itibaren sıradaki filtreler işlem görmez ve kullanıcı login yapabilmesi için login.jsp sayfasına yönlendirilir. Böylece sadece login yapmış üyelerin kullanabileceği fonksiyonların login yapmamış bir üye tarafından kullanımı engellenmiş olur.

Filtre görevini yerine getirdikten sonra destroy() metodu ile yok edilir. Bunu uygulama sunucusu otomatik olarak gerçekleştirir.

Filtremizin Tomcat altında çalışabilmek için aşağıdaki şekilde web.xml dosyasına kayıtlanması gerekmektedir:

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/JEE"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/JEE/web-app_2_4.xsd">

    <filter>
        <filter-name>AuthenticationFilter</filter-name>
        <filter-class>
                  package com.pratikprogramci.designpatterns.pattern.interceptingfilter
                  .AuthenticationFilter
        </filter-class>
    </filter>

</web-app>
```

Intercepting Filter tasarım şablonu ne zaman kullanılır?

Kullanıcı isteği (request) işleme alınmadan önce ve işlem tamamlandıktan sonra filtreler aracılığı ile logging, üyelik ve veri kontrolü gibi işlemler için intercepting filter tasarım şablonu kullanılır.

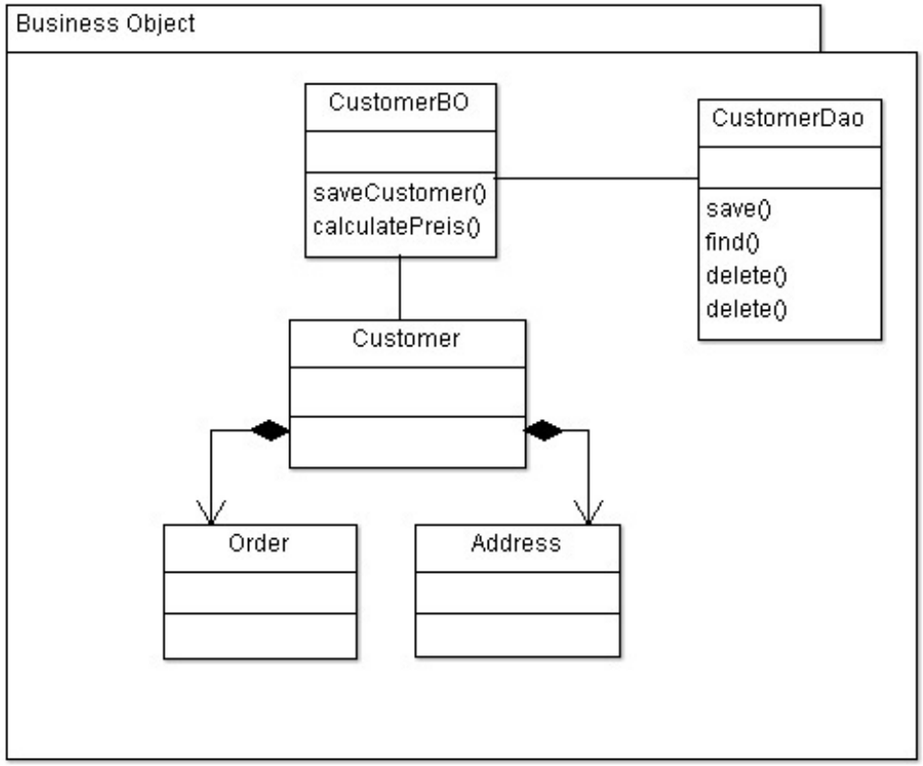
İlişkili tasarım şablonları

Front controller tasarım şablonu da intercepting filter tasarım şablonu gibi verilerin merkezi bir yerde işlem görmesini kolaylaştırır.

Business Object Tasarım Şablonu

Uygulamanın sahip olduğu alan modeli (domain model; bknz. 9. bölüm DataMapper tasarım şablonu) uygulama bünyesinde kullanılan sınıflar yanı sıra bu sınıfların birbirleriyle olan ilişkilerini modellemek için de kullanılmaktadır. Bu ilişkiler ve bu ilişkilerden doğan davranış biçimleri alan sınıflarında işletme mantığı olarak implemente edilirler.

Alan modelindeki sınıfları verileri modelleyen sınıflar olarak düşünecek olursak (Örneğin Müşteri, Sipariş, Adres gibi), bu sınıflardan oluşturulan nesneler belli bir durumu (state) ihtiva ederler. Örneğin veri tabanında bulunan musteri isimli tablodaki 1 numaralı veriyi DataMapper tasarım şablonu yardımıyla bir Musteri sınıfı nesnesine dönüştürdüğümüzde, bu nesne belli bir duruma sahip bir nesne haline gelmektedir. Bunun yanı sıra müşteri nesnesine bu durum ve diğer sınıflarla olan ilişkiler için gerekli işletme mantığını da yükledigimizde, sınıf hem durumunu hem sahip olduğu işletme mantığını hem de diğer sınıflarla olan ilişkilerini yönetmek zorunda kalacaktır. Birden fazla sorumluluğu olan sınıfların kırılganlığı artmaktadır. BusinessObject tasarım şablonu yardımı ile alan nesnelerinin sahip oldukları durum ve işletme mantığının ayrılması mümkün hale gelmektedir.



Resim 9

Resim 9 da Business Object tasarım şablonunun kullanım şeklini görmekteyiz. CustomerBO Customer sınıfı için gerekli işletme mantığı ihtiva etmektedir. CustomerBO aynı zamanda CustomerDao aracılığı ile Customer nesneleri için gerekli gerekli veri tabanı işlemlerini gerçekleştirmektedir. CustomerDao sadece veri tabanı işlemlerine odaklanırken, CustomerBO sınıfı Customer ve ilişki içinde olduğu Order ve Address için gerekli işletme mantığını ihtiva etmektedir. Customer sınıfı kendi bünyesinde hiçbir işletme mantığına sahip değildir. Customer sınıfı sadece sahip olduğu ilişkileri tanımlamak ve yönetmekten sorumludur.

```
// Kod 16

package com.pratikprogramci.designpatterns.bolum8.bo;

public class CustomerBO {

    public double calculateOrderTotal(Customer customer) {

        double result = 0;

        for (Order order : customer.getOrders()) {
            result += order.getTotal();
        }
        return result;
    }
}
```

Kod 16 da CustomerBO sınıfının örnek bir implementasyonu görmekteyiz. Bu örnekte Customer bünyesindeki tüm sipariş tutarlarını toplayan ve calculateOrderTotal() ismini taşıyan bir metot yer almaktadır. Bunun gibi Customer sınıfı ile ilgili daha değişik işletme mantığı metotları düşünülebilir. Eğer business object tasarım şablonunu kullanmamış olsaydık, calculateOrderTotal() gibi metotlar Customer ve benzeri alan sınıfları bünyesinde yer almak zorunda kalacaklar ve bu sınıfların zaman içinde genişlemelerini ve birden fazla sorumluluğa sahip olmalarını mecbur kılacaklardı.

Business Object tasarım şablonu ne zaman kullanılır?

Durum (state) ve işletme mantığının ayrılması gerektigi durumlarda bu tasarım şablonu kullanılabilir. Bu bazı sınıfların işletme mantığına, bazı sınıfların ise sınıflar arası ilişkilere ve bu ilişkilerden doğan durumlara konsantre olmalarını mümkün kılmaktadır. Her şeyi yapan bir sınıf yerine, görev dağılımı prensibine dayanarak, kendi alanında uzman sınıf oluşumu tercih edilmelidir.

İlişkili tasarım şablonları

Onuncu bölümde inceleyeceğimiz business facade implementasyonu ile işletme katmanına olan erişim kontrol erilirken, business object tasarım şablonu ile alan nesneleri için gerekli işletme mantığı implemente edilir. Business object alan nesnelerine erişimi sağlamak için kullanılmaz.

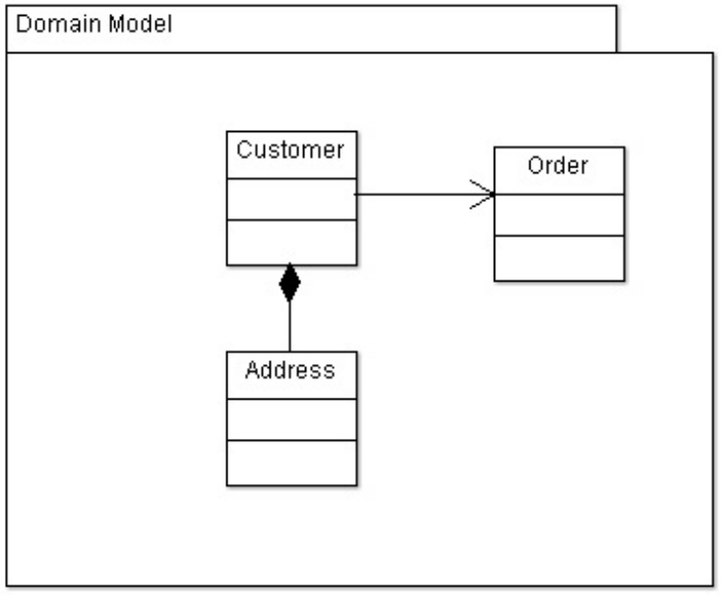
9. Bölüm

Daha Fazla Tasarım Şablonu

DataMapper Tasarım Şablonu

Alan modelleri (domain model) bir uygulama bünyesindeki sınıflar ve bu sınıfların birbirleriyle olan ilişkilerini modellemek için oluşturulurlar. Resim 1 de böyle bir alan modeli görülmektedir.

Bu alan modelinde bir Customer sınıfı kendi bünyesinde bir Address nesnesini barındırmaktadır ve bir Order (sipariş) sınıfı ile ilişkilidir.



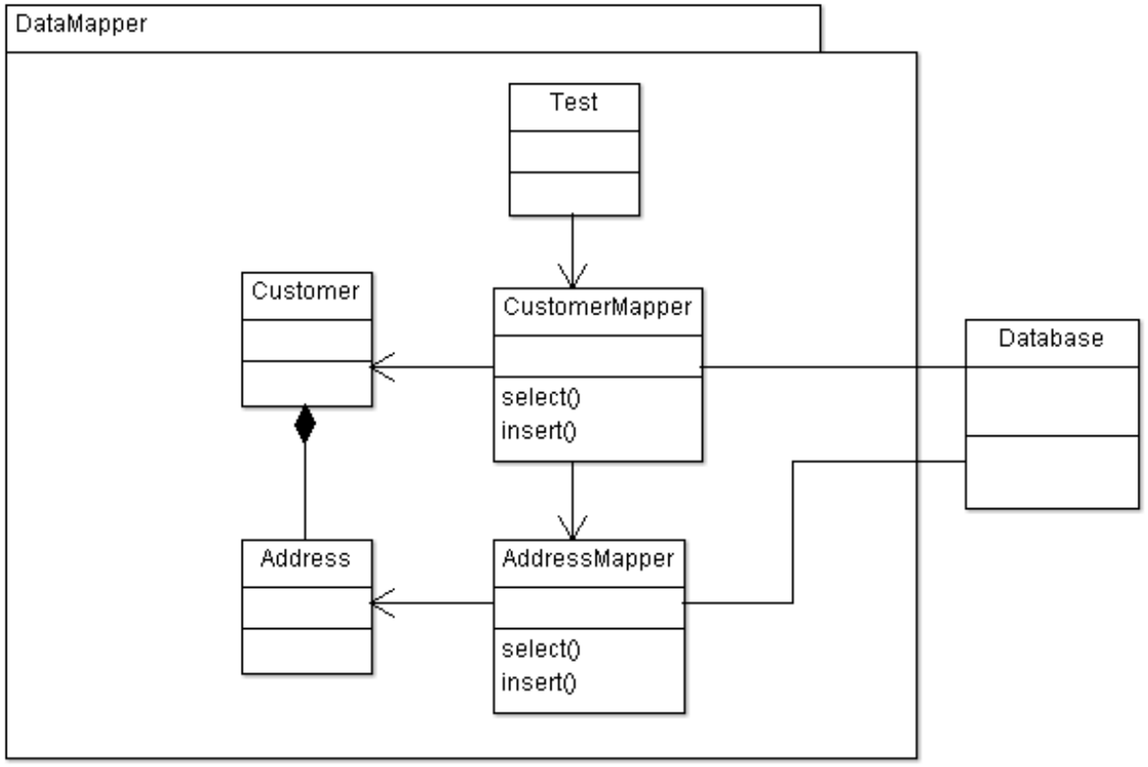
Resim 1

Bu tür sınıflar genelde bir uygulamanın veri tabanında tutulan verilerini modellemek için kullanılmaktadırlar. Bu sınıflardan oluşturulan nesneleri veri tabanındaki verilerin uygulama bünyesindeki canlı halleri olarak düşünebiliriz. Durum böyle iken bu nesneler ve veri tabanı arasında senkronizyona ihtiyaç duyulmaktadır. Bu örneğin Java dünyasında JDBC (Java Database Connectivity) kullanılarak yapılabilir.

Bu sınıfların veri tabanı ile yapılması gereken senkrozisyon kodunu ihtiva etmeleri durumunda, onlar için yeni bir sorumluluk alani açılmış olur. Bu onların daha çok kod ve sorumluluk sahibi olmalarını da beraberinde getiren bir durum olacaktır. Birden fazla sorumlulugu olan sınıfların kırılganlık seviyesi her yeni sorumlulukla ile daha da artmaktadır. DataMapper tasarım şablonu aracılığı ile veri tabanı sekronizasyonu sorumluluğunu bu işten sorumlu ve bu işte uzman sınıflara verebiliriz.

DataMapper tasarım şablonu için gerekli sınıfları resim 2 de görmekteyiz. Customer ve Address sınıfları için bu verilerin veri tabanından JDCB ile nasıl alındığı bilgisine ve Customer ve Address

sınıflarından nasıl nesne olusturulacağını bilen Mapper implementasyonlarına ihtiyaç duymaktayız. Şimdi böyle bir senaryonun nasıl implemente edildigini bir örnek üzerinde inceleyelim.



Resim 2

```
// Kod 1

package com.pratikprogramci.designpatterns.bolum9.datamapper;

import java.sql.Date;

public class Customer {

    private String name;
    private String firstname;
    private Date birthday;
    private Address address;

    public String getName() {
        return name;
    }
```

```
    public void setName(String name) {
        this.name = name;
    }

    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }
}

// Kod 2

package com.pratikprogramci.designpatterns.bolum9.datamapper;

public class Address {

    private String street;
    private String no;
    private String city;
    private String country;

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }
```

```
    public String getNo() {
        return no;
    }

    public void setNo(String no) {
        this.no = no;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }
}
```

Kod 1 ve 2 de Customer ve Address sınıflarını görmekteyiz. Customer sınıfı kendi bünyesinde bir Address nesnesi ihtiva etmektedir. Veri tabanında customer isminde bir tablonun olduğunu düşünecek olursak, kod 3 de yer alan CustomerMapper sınıfı ile bu tabloyu sorgulayarak, bir Customer nesnesi oluşturabliriz.

```
// Kod 3

package com.pratikprogramci.designpatterns.bolum9.datamapper;

import java.sql.ResultSet;

public class DataMapper {

    public Customer map(ResultSet result) {

        Customer customer = null;
        try {
            if (result.next()) {
                customer = new Customer();
```

```
                customer.setName(result.getString("name"));
                customer.setFirstname(
                        result.getString("firstname"));
                customer.setBirthday(
                        result.getDate("birthday"));
                customer.setAddress(
                        new AddressMapper().map(result));
            } else {
                throw new RuntimeException(
                        "No customer recourd found");
            }

        } catch (Exception e) {
            throw new RuntimeException(e);
        }

        return customer;
    }
}
```

CustomerMapper sınıfındaki map() metodu java.sql paketinde yer alan bir ResultSet nesnesini parametre olarak almaktadır. ResultSet bir SQL select sonrasında oluşan veri tabanı iteratörüdür ve onu kullanarak bir Customer nesnesi oluşturabiliriz.

CustomerMapper.map() bünyesinde bir Customer nesnesi oluşturulurken, Customer bünyesindeki Address nesnenin oluşma işleminin AddressMapper sınıfına delege edildigini görmekteyiz. Bu Mapper implementasyonlarının birbirlerini kullanabildikleri anlamına gelmektedir. Burada kullanılan AddressMapper sınıfı CustomerMapper sınıfı ile benzer yapıdadır.

Veri tabanındaki veriden bir Customer nesnesi oluşturmak için kullandığımız map() metodunun ismi select() de olabilirdi. Aynı şekilde bir insert() metodu oluşturarak, bir Customer nesnesini veri tabanında bulunan customer isimli tabloya SQL insert komutuyla ekleyebiliriz.

```
// Kod 4

package com.pratikprogramci.designpatterns.bolum9.datamapper;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class Test {

    public static void main(String[] args)
```

```
            throws Exception {

        Connection con = ConnectionFactory.connection();
        PreparedStatement pstmt = con.prepareStatement(
                "select * from customer where id=?");
        pstmt.setInt(1, 100);
        ResultSet result = pstmt.executeQuery();

        Customer customer = new DataMapper().map(result);
        System.out.println(customer.getFirstname() + " "
                + customer.getName() + " loaded!");
    }
}
```

DataMapper tasarım şablonunun uygulanış şeklini kod 4 de yer alan Test sınıfında görmekteyiz. Veri tabanında bulunan customer tablosu id=100 değeri ile sorgulandıktan sonra, oluşan result nesnesi ile DataMapper.map() aracılığı ile bir Customer nesnesi oluşturulmaktadır.

DataMapper tasarım şablonu ne zaman kullanılır?

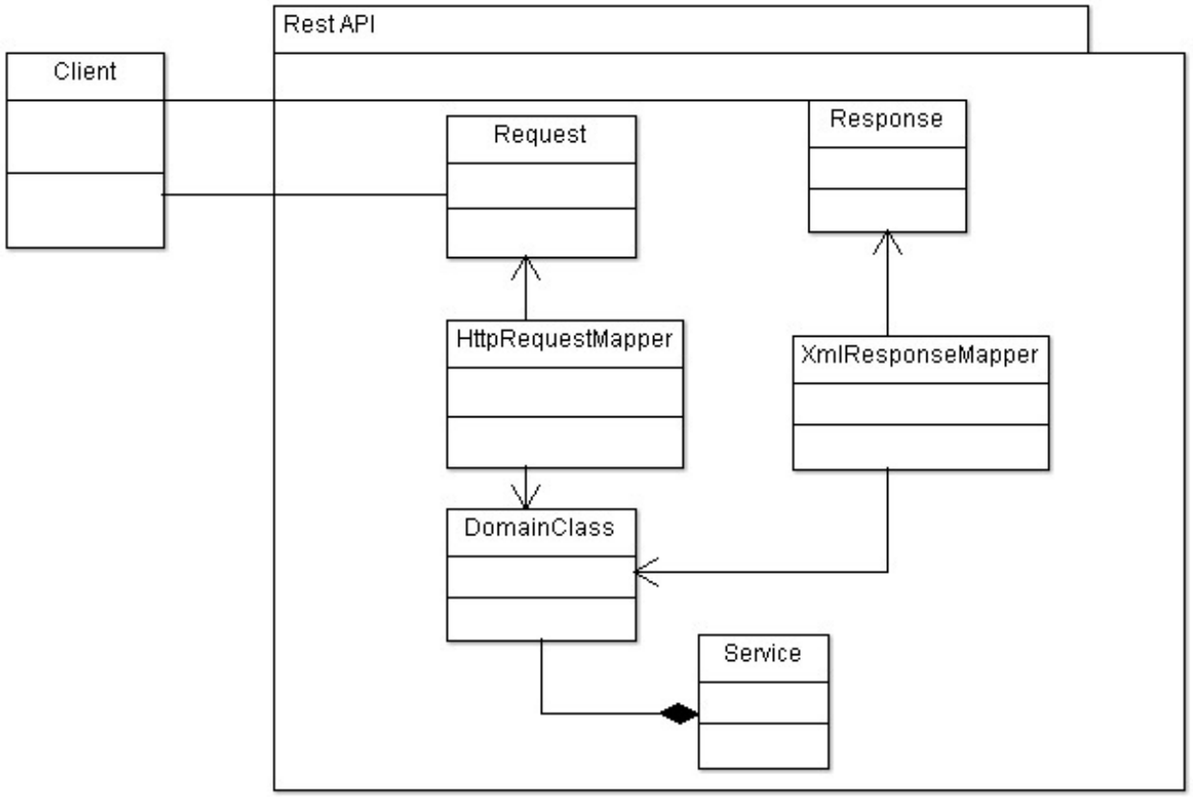
Veri tabanı işlemlerini alt bir katmanda gizlemek ve alan modelini oluşturan sınıfların bu sorumluluktan uzak durmalarını sağlamak için DataMapper tasarım şablonu kullanılmaktadır. Veri tabanı ile alan modelindeki sınıflar arasındaki senkronizasyon için DataMapper tasarım şablonu tercih edilmelidir.

İlişkili Tasarım Şablonları

Iterator tasarım şablonu aracılığı ile bir döngü içinde birden fazla nesne oluşturulabilir. DAO tasarım şablonu veri tabanı erişimini kapsüllemek için kullanılır.

RequestMapper ve ResponseMapper Tasarım Şablonları

DataMapper ve RequestMapper/ResponseMapper tasarım şablonları aynı işlevsel prensibe sahiptirler. DataMapper daha cok veri tabanı ile alan nesnesi arasındaki eşleme işleminden sorumlu iken, RequestMapper tasarım şablonu bir sisteme iletilen kullanıcı isteğinin (request) bir alan nesnesine dönüştürülmesi, ResponseMapper tasarım şablonu ise bir alan nesnesinin ihtiva ettiği bilgilerin kullanılan protokole uyumlu hale getirilerek, bu bilginin kullanıcıya geri gönderilmesi için kullanılmaktadır.



Resim 3

Resim 3 de yer alan örnekte kullanıcı bir REST tabanlı uygulamaya HTTP protokolü aracılığı ile veri girişinde bulunmaktadır. HttpRequestMapper sınıfı HTTP protokolü ile gelen verilerin nasıl dekode edilerek, bir DomainClass nesnesine dönüştürülebileceği bilgisine sahiptir ve bu işlemden sorumludur. DomainClass Service ismini taşıyan servis sınıfı tarafından kullanılmaktadır. Kullanıcı isteği tamamlandıktan sonra yine bir DomainClass nesnesi XmlResponseMapper aracılığı ile bir XML dosyasına dönüştürülerek, kullanıcıya bu veri HTTP transport protokolü üzerinden aktarılmaktadır.

RequestMapper/ResponseMapper tasarım şablonu ne zaman kullanılır?

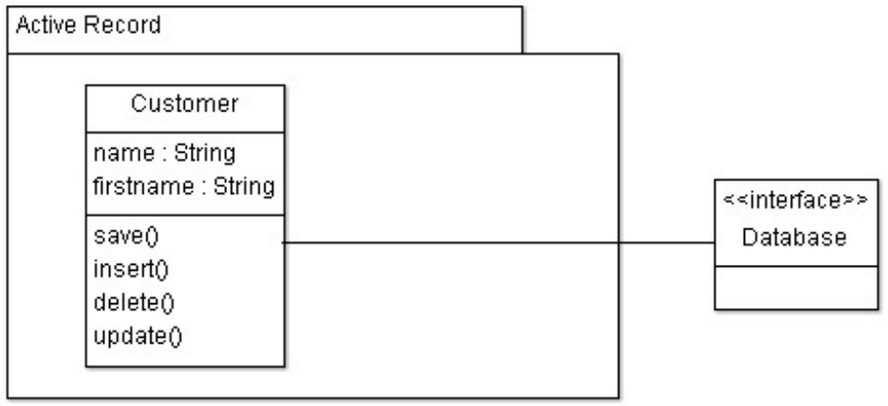
Kullanıcı ile uygulama arasındaki veri akışını alan nesneleri üzerinden yönetmek için bu tasarım şablonları kullanılabilir. Ayrıca çok katmanlı mimarilerde bir alt katmanın verilerini bir üst katmanın tanıdığı alan nesnelerine dönüştürmek için de kullanilabilirler.

İlişkili Tasarım Şablonları

- Bknz. DataMapper tasarım şablonu.

Active Record Tasarım Şablonu

DataMapper tasarım şablonunda veri tabanında bulunan verilerin bir nesneye nasıl dönüştürüldüğünü gördük. Active Record DataMapper tasarım şablonunun kardeşidir ve aynı işlemi yapmak için kullanılır. Active Record bir Mapper sınıfı kullanmak yerine, veri tabanı işlemlerini alan sınıfı bünyesinde gerçekleştirir.



Resim 4

Kullanılan alan sınıfının veri tabanında tablo olarak bir karşılığının bulunması gerekmektedir. Alan sınıfı bünyesindeki tüm veri tabanı işlemleri bu tablo üzerinde gerçekleştirilir. Bu tablodaki bir kayıt (record) alan nesnesine dönüştürülür, bir alan nesnesinin ihtiva ettiği bilgiler bu tabloya kayıt olarak eklenir ya da mevcut bir kayıt güncellenir.

Kod 5 de active record tasarım şablonunun nasıl implemente edilebileceğini görmekteyiz. Bu örnekte sadece select() metodu yer almaktadır. Buna alanog olarak insert(), delete() ve update() metotları implemente edilebilir.

```
// Kod 5

package com.pratikprogramci.designpatterns.bolum9.activerecord;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

public class Customer {

    private String name;
    private String firstname;
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    public Customer select() throws Exception {
        Connection con = ConnectionFactory.connection();
        PreparedStatement pstmt = con.prepareStatement(
                "select * from customer where id=?");
        pstmt.setInt(1, 100);
        ResultSet result = pstmt.executeQuery();

        try {
            if (result.next()) {
                setName(result.getString("name"));
                setFirstname(result.getString("firstname"));
            } else {
                throw new RuntimeException(
                        "No customer recourd found");
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
        return this;
    }
}
```

Active record tasarım şablonu yapısı itibari ile tek sorumluluk prensibine (SRP - Single Responsibility Principle) ters düşmektedir. Her alan sınıfı ihtiyaç duyduğu veri tabanı işlemlerini kendi bünyesinde taşımak zorundadır. Bu alan sınıflarının birden fazla sorumlulugu olacağı anlamına gelmektedir. Bu yüzden active record implementasyonları kırılgandırlar.

Bunun yanı sıra bir alan sınıfının veri tabanı işlemlerinin nasıl yapıldığını bilmesi, bu ve benzeri sınıfları veri tabanı teknolojisine bağımlı kılacaktır. Bu bağımlılık kullanılan sınıf adedinin

yükselmesiyle uygulamanın bakımı ve geliştirilmesini zora sokabilir.

Active record implementasyonlarının test edilmeleri (unit test - birim testi) zordur, çünkü test edilebilmeleri için bir veri tabanına ihtiyaç duyulmaktadır. Bu yüzden birim testleri implemente etmek mümkün değildir. Sadece entegrasyon testleri ile test edilebilirler.

Active Record tasarım şablonu ne zaman kullanılır?

Eğer insert(), select(), delete() gibi metotlar elden implemente edilecekse, active record tasarım şablonunun kullanımı tavsiye edilmemektedir. Bunun yerine DataMapper tasarım şablonu yardımı ile en azından veri tabanı erişimi ve eşleme işlemi başka bir sınıf bünyesinde yapılmalı ve alan sınıflarının veri tabanına olan bağımlılıkları azaltılmalıdır.

Ruby gibi dillerde alan sınıfları otomatik olarak active record implementasyonuna sahiptirler ve select(), delete(), update() gibi metotların elden implemente edilmeleri zorunluluğu bulunmamaktadır.

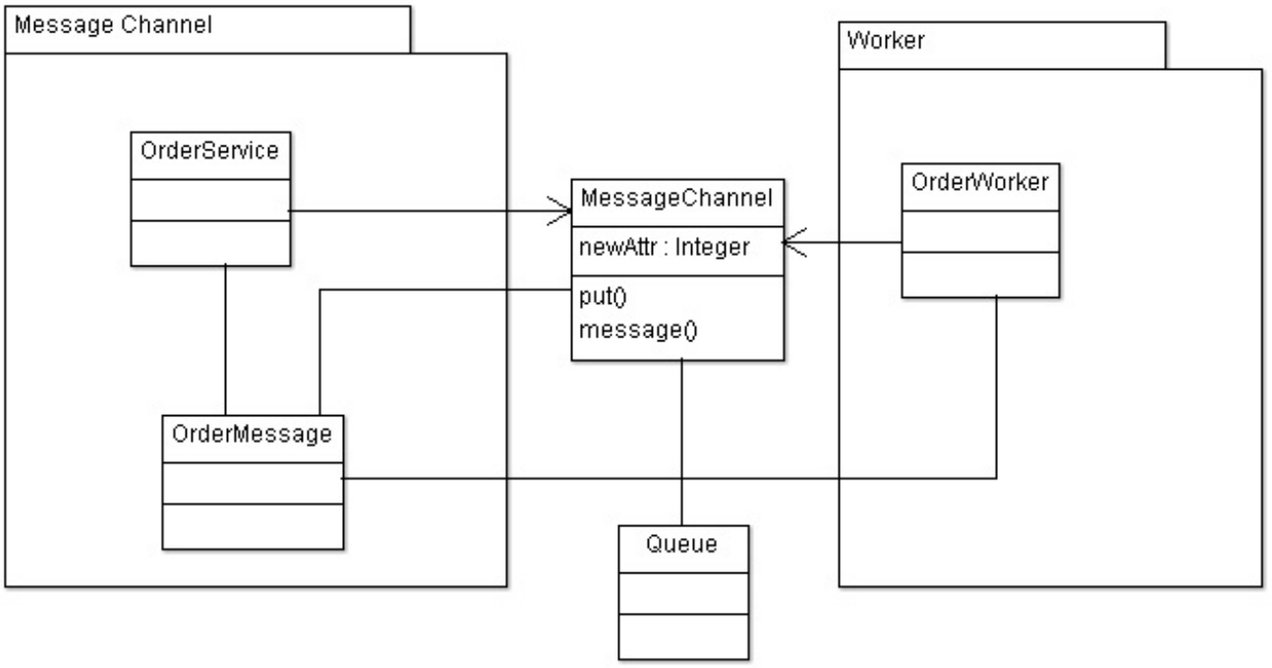
İlişkili Tasarım Şablonları

- Bknz. DataMapper tasarım şablonu.

Message Channel Tasarım Şablonu

Uygulamaların çoğunda sınıflar arası veri alışverişi senkron olarak gerçekleşir. Bir sınıf başka bir sınıfın metodunu koşturur ve bu metot son bulana kadar bloke olur. Metodun son bulmasıyla birlikte ilk sınıf kendi işlemlerine devam eder.

Uygulamanın senkron çalısması, birbirlerini kullanan kod parçalarının doğal olarak birbirlerini tanımaları bu yüzden bir nevi bağımlılığın oluştuğu anlamına gelmektedir. Eğer iki sınıf birbirini kullanarak bir işlem gerçekleştiriyorlarsa, bu iki sınıfın her zaman birlikte olmaları ve birlikte çalışmaları gerekmektedir. Bu bağımlılığı ortadan kaldırmak ve bahsettiğim iki sınıfın birbirlerini tanımadan, bilmeden birlikte çalışmalarını sağlamak için message channel tasarım şablonu kullanılabilir.



Resim 5

Message channel tasarım şablonunda iletişim içinde olan sınıflar arasına mesaj alıp, verebilmek için bir nevi tünel ya da kanal oluşturulur. Tünel gerekli mesajların sınıflar arasında taşınması için kullanılır. Tünelin iki ucundaki sınıflar birbirlerini tanımak zorunda değildirler. Bir sınıf tünele istediği mesajı gönderir, diğer sınıf tünelin diğer ucundan bu mesajı alarak, gerekli işlemi gerçekleştirir. Mesajın gönderilmesi ve alınması asenkron olarak implemente edilebilir. Bu her iki tarafın da bloke olmadan çalışabildikleri anlamına gelmektedir.

Resim 5 de message channel tasarım şablonunun implementasyon şekli görülmektedir. OrderService ismindeki sınıf kendisine iletilen Order nesnelerini OrderMessage nesnesine dönüştürerek, MessageChannel bünyesinde yer alan kanala göndermektedir. Kanalın diğer ucunda OrderWorker ismini taşıyan bir sınıf bu mesajları alarak, işlemektedir. Bu örnekte görüldüğü gibi OrderService ve OrderWorker sınıfları birbirlerini tanımadan, Order nesneleri üzerinde birlikte çalışabilmektedirler.

Şimdi message channel tasarım şablonunun nasıl implemente edildigini bir örnek üzerinde inceleyelim.

OrderService isminde, siparişleri işleyen bir servis sınıfımız bulunuyor. Bu sınıf kendisine gönderilen Order nesnelerinden bir OrderMessage nesnesi oluşturmaktadır. Bu mesaj MessageChannel sınıfında bulunan kanala gönderilmektedir.

```
// Kod 6

package com.pratikprogramci.designpatterns.bolum9.channel;

public class OrderService {

    public void newOrder(Order order) {
        OrderMessage orderMessage = new OrderMessage(order);
        MessageChannel.put(orderMessage);
    }
}
```

MessageChannel sınıfının yapısını kod 7 de görmekteyiz.

```
// Kod 7

package com.pratikprogramci.designpatterns.bolum9.channel;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class MessageChannel {

    private static BlockingQueue<OrderMessage> orderQueue =
            new ArrayBlockingQueue<OrderMessage>(10);

    public static void put(OrderMessage message) {
        orderQueue.add(message);
    }

    public static OrderMessage message() throws Exception {
        return orderQueue.take();
    }
}
```

MessageChannel sınıfı bünyesinde gerekli kanalı oluşturmak için BlockingQueue yapısı kullanılmıştır. Bu sınıfın implementasyonu olan ArrayBlockingQueue ile gerçek bir kanal oluşturulmaktadır. Bu kanal içinde mesaj olmadığı zaman bloke olan bir yapıdır. Bunun ne anlama geldiğiniz OrderWorker sınıfında yakından inceleyeceğiz.

OrderMessage sınıfı kod 8 de görmekteyiz.

```
// Kod 8

package com.pratikprogramci.designpatterns.bolum9.channel;
```

```
public class OrderMessage {

    private Order order;

    public OrderMessage(Order order) {

        this.order = order;
    }

    public Order getOrder() {
        return order;
    }
}
```

Kod 9 da yer alan Test sınıfı ile bir Order nesnesinin sistem tarafından işlenişini inceleyebiliriz.

```
// Kod 9

package com.pratikprogramci.designpatterns.bolum9.channel;

public class Test {

    public static void main(String[] args) {

        new Thread(new OrderWorker()).start();

        Order order = new Order();
        Customer customer = new Customer();
        order.setCustomer(customer);
        order.setId(100);
        Item item = new Item();
        order.addItem(item);

        OrderService service = new OrderService();
        service.newOrder(order);
        service.newOrder(order);
        service.newOrder(order);
        service.newOrder(order);
        service.newOrder(order);

    }
}
```

Biraz sonra kodunu inceleyeceğiz OrderWorker sınıfı bir Thread olarak çalıştırılmaktadır.

OrderWorker sürekli kanalın kendi tarafında yeni mesajları kontrol etmektedir. Eger kanal içinde mesaj varsa işlem yapılmakta, mesaj yoksa OrderWorker bloke olmakta ve yeni veri için beklemektedir. OrderWorker MessageChannel.message() metodunu koşturduğu anda kanalda mesaj yoksa bloke olmaktadır.

Test sınıfında yeni bir Order nesnesi oluşturulduktan sonra, bu nesne OrderService sınıfına iletilmektedir. OrderService sınıfı kod 6 da görüldüğü gibi bu nesneyi OrderMessage olarak paketleyerek, MessageChannel bünyesindeki kanala göndermektedir.

Şimdi kanala gönderilen bir OrderMessage nesnesinin OrderWorker tarafından nasıl işlendiğini inceleyelim. OrderWorker sınıfı kod 10 da yer almaktadır.

```
// Kod 10

package com.pratikprogramci.designpatterns.bolum9.channel;

import java.util.Date;

public class OrderWorker implements Runnable {

    @Override
    public void run() {
        try {
            while (true) {
                OrderMessage message = MessageChannel
                        .message();
                System.out.println("("
                        + new Date(
                                System.currentTimeMillis())
                        + ") New order with id "
                        + message.getOrder().getId()
                        + "is processed!");
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

OrderWorker sınıfı Runnable interface sınıfını implemente ettiği için bir Thread olarak çalıştırılabilmektedir. Bunun nasıl yapıldığı kod 9 da yer alan Test sınıfında görmekteyiz. OrderWorker sınıfının bir Thread olarak implemente edilmesi bir gereklilik. Sadece bu şekilde kod 9 da yer alan ana program akışı (main) ile OrderWorker çalışma sürecini bibirbirinden

ayrıştırabilmekteyiz. Aksi taktirde kanal içinde mesaj olmadığından, OrderWorker bloke olur ve main() metodunda bir ilerleme olmaz.

Run() metodu bünyesinde bir döngü içinde kanaldaki mesajlar işlenmektedir. MessageChannel.message() metodu koşturulduğunda kanalın içinde bir mesaj varsa, message() metodu bu mesajı geri verecektir. Eğer kanal içinde mesaj yoksa, message() metodu kanala yeni bir mesaj eklenene kadar bloke olacaktır.

Test sınıfını koşturduğumuzda, ekran çıktısı su şekilde olacaktır:

```
(Thu Mar 17 18:26:23 CET 2016) New order with id 100 is processed!
(Thu Mar 17 18:26:23 CET 2016) New order with id 100 is processed!
(Thu Mar 17 18:26:23 CET 2016) New order with id 100 is processed!
(Thu Mar 17 18:26:24 CET 2016) New order with id 100 is processed!
(Thu Mar 17 18:26:24 CET 2016) New order with id 100 is processed!
```

Görüldüğü gibi OrderWorker Test sınıfı tarafından OrderService üzerinden kanala gönderilen tüm mesajları tüketmiştir.

Message Channel tasarım şablonu ne zaman kullanılır?

İki sınıf arasında esnek bag oluşturmak için message channel tasarım şablonu kullanılabilir. Loose coupling olarak ifade edilen bu esnek bağ, her iki sınıfın birbirlerinden bağımsız olarak geliştirilmelerini mümkün kılacaktır.

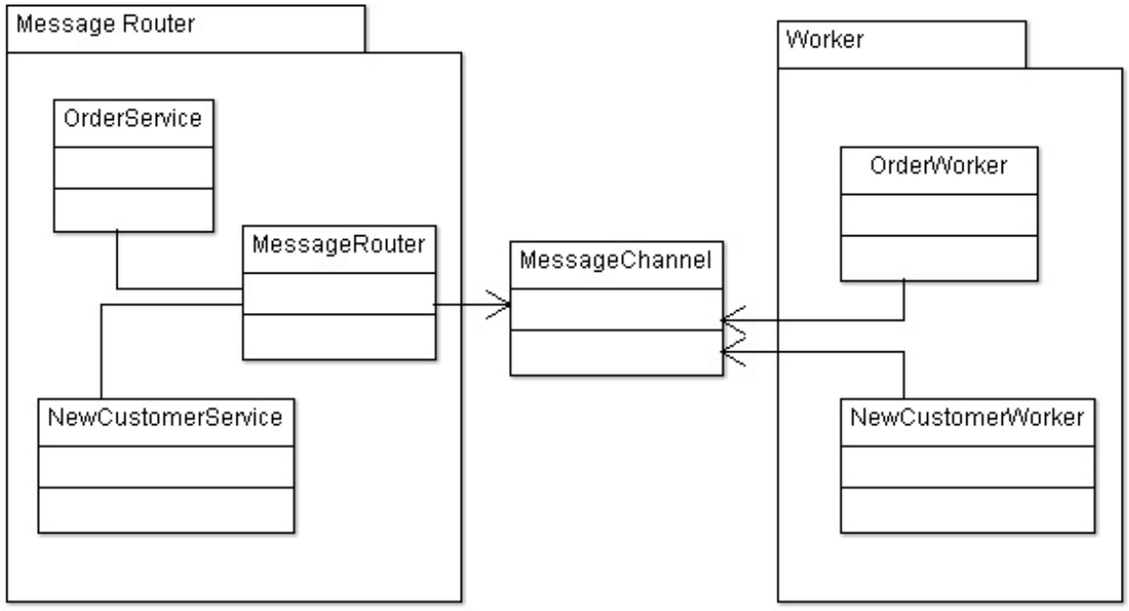
İlişkili Tasarım Şablonları

Komut tasarım şablonu message channel tasarım şablonunda olduğu gibi birbirini tanımayan iki sistem arasında veri taşıma işlemini gerçekleştirmek için kullanılmaktadır.

Message Router Tasarım Şablonu

Message channel tasarım şablonunda iki sistemin bir kanal aracılığı ile bloke olmadan nasıl birlikte çalışabildiklerini inceledik. Message Router tasarım şablonu bu fikri bir adım daha ileri götürerek, sistem bünyesinde kullanılan mesajların sahip oldukları veri tipine gore gerekli kanala yönlendirilmelerini mümkün kılmaktadır.

Bunun nasıl yapıldığını şimdi bir örnek üzerinde inceleyelim.



Resim 6

MessageRouter ismindeki sınıf kendisine gönderilen mesajları ilgili kanallara aktarmaktadır. Kod 11 de OrderService MessageRouter sınıfına oluşturduğu mesajı aktarmaktadır.

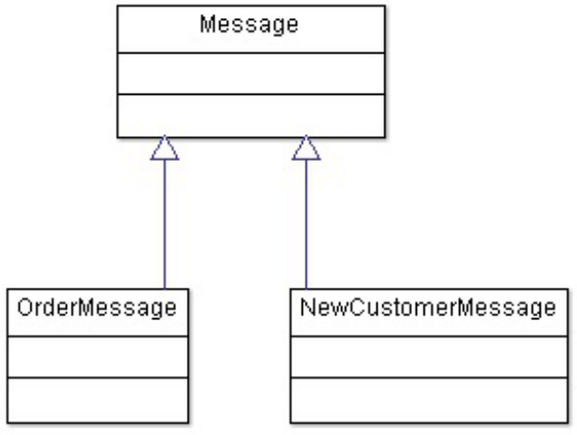
```
// Kod 11

package com.pratikprogramci.designpatterns.bolum9.router;

public class OrderService {

    public void newOrder(Order order) {
        OrderMessage orderMessage = new OrderMessage(order);
        MessageRouter.route(orderMessage);
    }
}
```

Aynı işlemi NewCustomerService sınıfı da gerçekleştirmektedir:

```
// Kod 12

package com.pratikprogramci.designpatterns.bolum9.router;

public class NewCustomerService {

    public void newCustomer(Customer customer) {
        NewCustomerMessage message = new NewCustomerMessage(
                customer);
```

```
        MessageRouter.route(message);
    }
}
```

Sistemde değişik tipte Message nesneleri kullanılmakta. Message ve altsınıflarını resim 7 de görmekteyiz.



Resim 7

MessageRouter sınıfı kullanılan mesaj tipine göre kendisine aktarılan mesaj nesnesini gerekli kanala aktarmaktadır. Bu işlemi kod 13 de görmekteyiz.

```
// Kod 13

package com.pratikprogramci.designpatterns.bolum9.router;

public class MessageRouter {

    public static void route(Message message) {
        if (message instanceof OrderMessage) {
            MessageChannel.order(message);
        } else if (message instanceof NewCustomerMessage) {
            MessageChannel.newCustomer(message);
        }
    }
}
```

MessageChannel bünyesinde kullanılan mesaj tipi kadar kanal bulunmaktadır.

```
// Kod 14

package com.pratikprogramci.designpatterns.bolum9.router;
```

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class MessageChannel {

    private static BlockingQueue<OrderMessage> orderQueue =
            new ArrayBlockingQueue<OrderMessage>(10);
    private static BlockingQueue<NewCustomerMessage>
    newCustomerQueue = new ArrayBlockingQueue<NewCustomerMessage>(10);

    public static OrderMessage orderMessage()
            throws Exception {
        return orderQueue.take();
    }

    public static NewCustomerMessage newCustomerMessage()
            throws Exception {
        return newCustomerQueue.take();
    }

    public static void order(Message message) {
        orderQueue.add((OrderMessage) message);
    }

    public static void newCustomer(Message message) {
        newCustomerQueue.add((NewCustomerMessage) message);
    }
}
```

Worker implementasyonları kendilerine ait olan mesajları hangi kanaldan alacaklarını bilmektedirler.

```
// Kod 15

package com.pratikprogramci.designpatterns.bolum9.router;

import java.util.Date;

public class NewCustomerWorker implements Runnable {

    @Override
    public void run() {
        try {
            while (true) {
                NewCustomerMessage message = MessageChannel
```

```
                        .newCustomerMessage();
                System.out.println("("
                        + new Date(
                                System.currentTimeMillis())
                        + ") New customer with id "
                        + message.getCustomer().getId()
                        + " is processed!");
                Thread.sleep(1000);
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

// Kod 16

package com.pratikprogramci.designpatterns.bolum9.router;

import java.util.Date;

public class OrderWorker implements Runnable {

    @Override
    public void run() {
        try {
            while (true) {
                OrderMessage message = MessageChannel
                        .orderMessage();
                System.out.println("("
                        + new Date(
                                System.currentTimeMillis())
                        + ") New order with id "
                        + message.getOrder().getId()
                        + "is processed!");
                Thread.sleep(1000);
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Bu örnekte gördüğümüz gibi message router tasarım şablonu message channel tasarım şablonu kullanılarak implemente edilmiştir. İki tasarım şablonu arasındaki farklılık, message router tasarım şablonunda kullanılan mesaj tipine göre kanal yönlendirme işleminin yapılmasıdır.

Message Router tasarım şablonu ne zaman kullanılır?

Sistemleri bir kanal aracılığı ile birbirlerinden ayırmak ve kullanılan mesaj tiplerine göre kanal yönlendirmesi yapılması gerektigi durumlarda message router tasarım şablonu kullanılabilir.

İlişkili tasarım şablonları

Bknz. Message channel tasarım şablonu.

Registry Tasarım Şablonu

Hafızadaki bir nesneye ulaşmak için o nesnenin referansını elimizde tutmamız gerekmektedir. Aşağıdaki örnekte bunu görmekteyiz:

```
// Kod 17

Customer cust = new Customer();
```

Cust isimli referansa sahip olduğumuz sürece, bu referansın altındaki müşteri nesnesine her zaman ulaşabiliriz. Eğer bu referansa sahip değilsek, bu müşteri nesnesi bizim için kayıp konumundadır. Bu durumda örneğin DAO ve DataMapper tasarım şablonları yardımı ile gerekli nesnenin tekrar veri tabanından edinilmesi gerekmektedir.

Daha önce oluşturduğumuz nesnelere olan erişimi kaybetmemek için yine hafızada bulunan merkezi bir kütük (registry) oluşturabiliriz. Mevcut nesneleri bu kütüğe ekleyerek, tekrar edinebiliriz.

Kütük bünyesinde nesneleri tutabilmek ve nesnelere tekrar ulaşabilmek için bir anahtara ihtiyaç duymaktayız. Bu anahtar kütüğe yerleştirmek istediğimiz nesneyi tekil olarak adreslemek için kullandığımız bir değişken değeri olabilr.

Kod 18 de yer alan örnekte bir müşteri nesnesi oluşturuyoruz. Daha sonra bu nesneyi CustomerRegistry isimli kütüğe addCustomer() metodu aracılığı ile ekliyoruz. Kütüğe nesne ekleme işlemi esnasında tüm nesneyi kütüğe gönderiyoruz. Aynı nesneyi tekrar kütükten edinmek istediğimiz taktirde, nesnenin sahip olduğu anahtarı bilmemiz gerekiyor. Aşağıdaki örnekte bu müşteri nesnesinin 100 rakamıdır.

```
// Kod 18

package com.pratikprogramci.designpatterns.bolum9.registry;
```

```
public class Test {

    public static void main(String args[]) {

        Customer customer = new Customer(100L);

        CustomerRegistry.addCustomer(customer);

        Customer customer1 = CustomerRegistry
                .getCustomer(100L);

        if (customer1 == null) {
            throw new RuntimeException(
                    "Customer not found!");
        }

        System.out.println(customer.getId());
    }
}

// Kod 19

package com.pratikprogramci.designpatterns.bolum9.registry;

public class Customer {

    private Long id;

    public Customer(Long id) {
        this.id = id;
    }

    public Long getId() {
        return id;
    }
}
```

Kütük implementasyonu kod 20 de yer almaktadır. Kütük nesneleri bir Map nesnesine tutmaktadır.

```
// Kod 20

package com.pratikprogramci.designpatterns.bolum9.registry;

import java.util.HashMap;
import java.util.Map;
```

```
public class CustomerRegistry {

    private static Map<Long, Customer> registry
    = new HashMap<Long, Customer>();

    public static Customer getCustomer(Long id) {
        return registry.get(id);
    }

    public static void addCustomer(Customer customer) {
        registry.put(customer.getId(), customer);
    }
}
```

Registry tasarım şablonu ne zaman kullanılır?

Hafızada yer alan nesnelere olan erişimi merkezi bir kütük üzerinden yönetmek için registry tasarım şablonu kullanılabilir.

İlişkili tasarım şablonları

Registry singleton tasarım şablonu kullanılarak implemente edilebilir. Facade tasarım şablonu servis sunan katmanlara erişimi merkezi bir yerden sağlarken, registry nesnelerin merkezi bir yerde bulunmalarını mümkün kılar.

Null Object Tasarım Şablonu

Birçok bilgisayar dilinde nil, nill ya da null gibi anahtar kelimeler bir nesne referansının hafızada bulunan hiçbir nesneye işaret etmediğini ve bir değere sahip olmadığını ifade etmek için kullanılırlar.

Doğru gibi görünen bu yaklaşım, kod yazarken bu kondisyonun tekrar tekrar control edilmesi zorunluluğunu beraberinde getirmektedir. Aşağıdaki kod parçasında bu problemi görmekteyiz.

```
// Kod 21

double salaryTotal = 0;
List<Customer> customers = DAO.getCustomers();
if (customers != null) {
    for (Customer customer : customers) {
        if (customer != null) {
            salaryTotal += customer.getSalary();
```

```
        }
    }
}
System.out.println("Total salary: " + salaryTotal);
```

Ne yazık ki ne zaman bir customer nesnesi ya da listesi edinsek, null olup, olmadığını kontrol etmek zorundayız. Aksı taktirde NullPointerException gibi bir hata oluşacaktır. Bu program akışını değiştirici niteliktedir. Ama gerçekten bir müşteri bulunamadığında, bunu bir program hatası olarak mı algılamak zorundayız?

Null object tasarım şablonu ile null değerlerine bakış açımız değişmektedir. Yukarıdaki örnekte DAO.getCustomers() metodu null değerini geriye vermek yerine, içinde en az bir adet NullCustomer veri tipinde nesne bulunan bir listeyi geriye verebilir. Bu listesinin boş olması da sorun teşkil etmemektedir. Önemli olan listenin null olmamasıdır.

Eğer liste içinde bir Customer nesnesi yerine null değeri yer alacak olursa, bu durumda yine if(customer != null) ile bu durumun kontrol edilmesi gerekmektedir. Null değeri yerine listede NullCustomer veri tipinde nesneler yer alabilir. NullCustomer, Customer sınıfının bir altsınıfı olup, bünyesinde boş ya da sıfır değerler taşımaktadır. Edindiğimiz nesnelerin null değerinde olamayacağını bildiğimiz taktirde, kodu aşağıdaki şekilde yeniden yapılandırabiliriz.

```
// Kod 22

double salaryTotal = 0;
List<Customer> customers = DAO.getCustomers();
for (Customer customer : customers) {
    salaryTotal += customer.getSalary();
}
System.out.println("Total salary: " + salaryTotal);
```

NullCustomer sınıfı aşağıdaki yapıya sahiptir:

```
// Kod 23

package com.pratikprogramci.designpatterns.bolum9.nullobj;

public class NullCustomer extends Customer {

    @Override
    public double getSalary() {
        return 0;
    }
}
```

Görüldüğü gibi getSalary() bünyesinde 0 değerini geriye vererek, maaş hesaplama işleminde geçerli bir değer sunmuş oluyoruz. Bu null kontrolü gerekliliğini ortadan kaldırmaktadır.

Null Object tasarım şablonu ne zaman kullanılır?

Null kontrolünü ve NullPointerException oluşumunu engellemek için null object tasarım şablonu kullanılabilir.

İlişkili tasarım şablonları

Null object singleton olarak implemente edilebilir. Bu null object nesnesinin hafızada sadece bir kopyasının bulunduğu anlamına gelmektedir. Bu şekilde null object için daha az hafıza alanı kullanılmış olur.

Dependency Injection Tekniği

Başlık olarak "Dependency Injection Tekniği" ni seçtim, çünkü bağımlılıkların enjeksiyonu olarak tercüme edebileceğimiz bu teknik bir tasarım şablonu değil, daha ziyade bağımlılıkların kontrolü ve yönetimi için kullanılmaktadır.

Java uygulamalarında bağımlılıklar nasıl oluşur? Bunun bir örneğini kod 24 de görmekteyiz.

```
// Kod 24

package com.pratikprogramci.designpatterns.bolum9.depinj;

public class ClassA {

    private ClassB classB = new ClassB();

    public void print(String msg) {
        classB.print(msg);
    }
}
```

ClassA ile ClassB arasında doğrudan bir ilişki mevcuttur ve ClassA bu ilişkiyi new operatörünü kullanarak hayata geçirmiştir. New operatörü ne yazık ki iki sınıfı beton dökmüşcesine birbirine bağlamaktadır. Bu durumda ClassA sınıfı ClassB sınıfını kullanmaya mecburdur ve ClassB olmadan derlenmesi bile mümkün değildir. Dependency injection tekniğini kullanarak, bu bağımlılığı daha esnek bir hale getirebiliriz.

```
// Kod 25

package com.pratikprogramci.designpatterns.bolum9.depinj.orn1;

public class ClassA {

    private ClassB classB;

    public ClassA(ClassB b) {
        this.classB = b;
    }

    public void print(String msg) {
        classB.print(msg);
    }
}
```

Kod 25 de dependency injection tekniğinin en basit uygulanış şekli yer almaktadır. Dependency injection tekniğinde bağımlılıkları new operatörü ile oluşturmak yerine, bu bağimlılıkların sınıf konstrüktörü ya da bir set() metodu üzerinden sınıfa enjekte edilmesi tercih edilmektedir. Kod 25 de ClassA sınıfının ihtiyaç duyduğu classB nesnesi sınıf konstrüktörü üzerinden ClassA sınıfına dışarıdan enjekte edilmektedir.

ClassA sınıfından nesne oluşturma işlemi kod 26 da yer almaktadır.

```
// Kod 26

package com.pratikprogramci.designpatterns.bolum9.depinj.orn1;

public class Test {

    public static void main(String[] args) {
        ClassA classA = new ClassA(new ClassB());
        classA.print("Hello World");
    }
}
```

Dependency injection tekniği uygulandığında, ClassA sınıfına ClassB sınıfı ve bu sınıfın altsınıfları enjekte edilebilir. Bu şekilde ClassA sınıfı birden fazla ClassB implementasyonu ile çalışır hale gelmektedir.

Dependency injection tekniğinin kullanıldığı uygulamalarda bağımlılıklar genelde interface sınıflar olarak tanımlanır. Bunun bir örneğini kod 27 ve 28 de görmekteyiz.

```
// Kod 27

package com.pratikprogramci.designpatterns.bolum9.depinj.orn2;

public interface Service {

    public void birMethot();

}

// Kod 28

package com.pratikprogramci.designpatterns.bolum9.depinj.orn2;

public class Uygulama {

    private Service service;

    public Uygulama(Service service){
        this.service = service;
    }

}
```

Kod 28 de yer alan Uygulama sınıfı kendi bünyesinde service isminde bir değişkene sahiptir. Bu değişkenin veri tipi bir interface sınıfı olan Service sınıfıdır. Bu uygulamaya Service sınıfını implemente eden herhangi bir sınıf nesnesi enjekte edilebilir. Bunu kod 29 da görmekteyiz.

```
// Kod 29

package com.pratikprogramci.designpatterns.bolum9.depinj.orn2;

public class ServiceImpl implements Service {

    @Override
    public void birMethot() {
    }

}


// Kod 30

package com.pratikprogramci.designpatterns.bolum9.depinj.orn2;
```

```
public class Test {

    public static void main(String[] args) {
        Service service = new ServiceImpl();
        Uygulama uygulama = new Uygulama(service);
    }

}
```

Kod 30 da ServiceImpl sınıfından olan nesneyi new ile oluşturduk ve Uygulama sınıfına enjekte ettik. Bu işlemi elden gerçekleştirmiş olduk. Dependency injection işleminin tam anlamıyla hayata geçirilebilmesi için bu işlemin de otomatize edilmesi ve kod dışında yapılması gerekmektedir. Spring gibi çatılarda bağımlılıklar bir XML ya da bir Java sınıfı bünyesinde tanımlanırlar. Bu şekilde uygulamayı tüm bağımlılıklar oluşturulmuş ve enjekte edilmiş bir şekilde kullanmak mümkündür. Böyle bir Spring konfigürasyon dosyasını kod 31 de görmekteyiz.

```
// Kod 31 - applicationContext.xml

<beans>
    <bean id="ClassA"
        class="com.pratikprogramci.designpatterns.bolum9.depinj.ClassA">
        <property name="classB" ref="ClassB" />
    </bean>

    <bean id="ClassB"
        class="com.pratikprogramci.designpatterns.bolum9.depinj.ClassB">
    </bean>
</beans>
```

Spring bu dosya yardımı ile bir ClassA nesnesi oluşturmadan önce bir ClassB nesnesi oluşturarak, bu nesneyi oluşturduğu ClassA nesnesine enjekte edecektir. Spring in yönettiği ClassA nesnesine şu şekilde ulaşabiliriz:

```
// Kod 32

ApplicationContext ctx = new ApplicationContext("applicationContext.xml");
ClassA classA = ctx.getBean(ClassA.class);
```

Dependency Injection tekniği ne zaman kullanılır?

Bağımlılık yönetiminin uygulama dışına taşınması gerektigi durumlarda ki bu her uygulama için uygulanması gereken bir işlemdir, dependency injection tekniği uygulanabılır.

Dependency injection tekniği implemente edilmemeli, bunun Spring ya da Guice gibi çatılar kullanılmalıdır.

İlişkili tasarım şablonları

Dependency injection tekniği fabrika tasarım şablonu kullanılarak implemente edilebilir. Service locator tasarım şablonu aracılığı ile bir bağımlılığın lokalize ve enjekte edilmesi sağlanabilir.

10. Bölüm

Yazılım Mimarisi ve Tasarım Şablonlarının Pratik Kullanımı

Bu bölümde şimdiye kadar incelediğimiz birçok tasarım şablonunu kullanarak, nasıl yazılım mimarisi üzerinde çalışabileceğimizi uygulamalı olarak göreceğiz. Bunun için hayal ürünü bir otel rezervasyonu programı tasarlıyacağız.

Daha öncede belirttiğim gibi tasarım şablonları yazılım esnasında karşılaşılan sorunlardan yola çıkarak, edinilen tecrübelerle oluşturulmuş çözümlerdir. Yazılım yaparken amacımız karşılaştığımız her sorunu kendi başımıza çözmek yerine, zaman kazanmak ve bakımı kolay kod yazabilmek için mevcut tasarım şablonlarından yararlanmak olmalıdır.

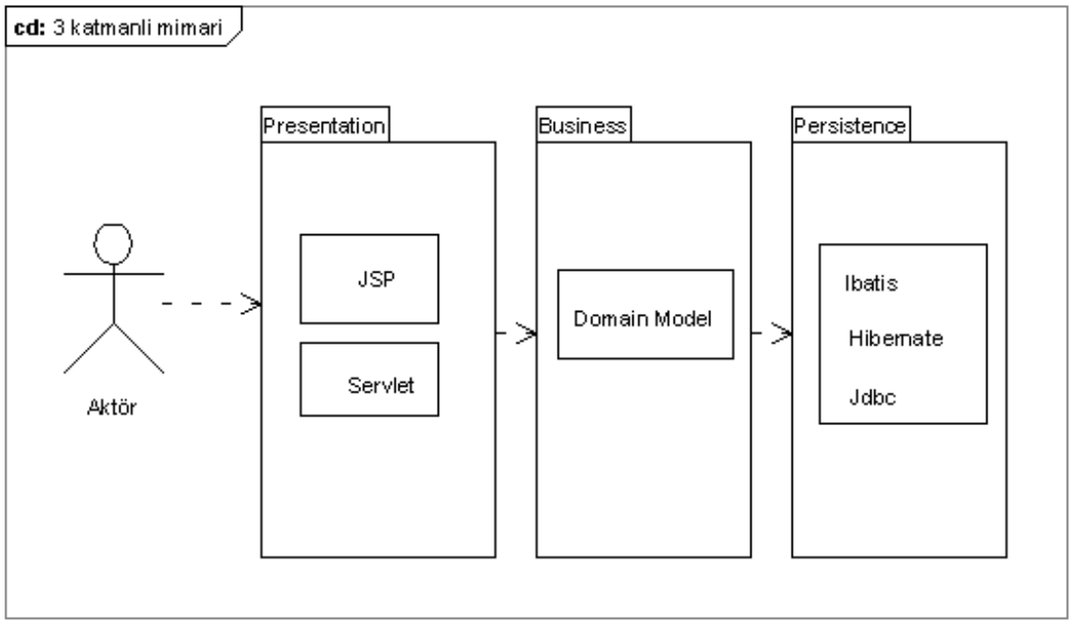
Doğru tasarım şablonunun doğru yerde kullanılmasını bilmek tecrübe gerektiren bir işlemdir. Bu yüzden programcıların bu konuda devamlı kendilerini geliştirmeleri ve uygulama yapmaları gerekmektedir. Tasarım şablonlarının başarılı bir şekilde ilk uygulanışını gerçekleştirdikten sonra, kodun ne kadar daha kolay okunur ve bakımı kolay hale geldiğini göreceğiz.

Bir evin yapımı için nasıl bir mimar tarafından teknik çizimler ve bir mühendis tarafından statik hesaplar yapılıyorsa, yazılımı yapılacak bir programın da yazılıma başlamadan önce kağıt üzerinde tasarımı yapılması gerekmektedir. Bu yapıldığı taktirde, nasıl bir program oluşturulacağı, karşılaşılabilecek muhtemel sorunlar ve çözümleri hakkında önceden fikir sahibi olmuş oluruz.

Oluşturulacak programın gelecekte oluşabilecek değişikliklere açık olabilmesi adına gerekli tasarım şablonlarının uygulanması büyük önem taşımaktadır. Esnek bir program mimarisi oluşturmanın temelinde doğru yerde kullanılmış tasarım şablonları yatmaktadır.

3 Katmanlı Mimari

Günümüzde yapılan kurumsal projelerin temelinde üç ya da daha fazla katmanlı mimariler yatmaktadır. Önce üç katmanlı mimarinin ne olduğu görelim. Daha sonra tasarım şablonları kullanılarak böyle bir mimarinin nasıl uygulanabileceğini otel rezervasyonu örneğinde görecegiz.



Resim 1

Program yazılımının amacı, veri oluşturmak, bu verileri depolamak, istendiği zaman depolanmış verileri elde edip, değerlendirmek ve belirli sonuçlara ulaşmaktır. Buradaki sihirli kelime „veri" dir ve bilgisayarın ve internetin icat edilmesinde başrolü oynamıştır.

Bir firmanın günlük faaliyetlerinde her gün birçok veri oluşur, bu veriler değerlendirilir ve firmanın bir veya birden fazla verı tabanında depolanır. Her firmanın stratejik faaliyetlerinden birisi, bu veri oluşumunun kontrollü bir şekilde yapılmasını sağlamak olmalıdır. Veri kaybı, aynı zamanda firmanın kazanç kaybı anlamına gelebileceği için birçok firma, sahip oldukları verilere bilgisayar ve bu bilgisayarlarda çalışan bir takım programlar aracılığı ile sahip çıkmaktadırlar.

Program yazılım amacının veriler üzerinde işlem yapmak olduğunun altını çizdik. Peki program yazma kriterleri nelerdir? Her programcı istediği şekilde, gelen istekler doğrultusunda program yazabilir mi? Evet yazabilir, ama programcının profesyonelliği, oluşan kodun kalitesi ile doğrudan orantılıdır. Edindiğim tecrübeler doğrultusunda, bir programın bakımı ve geliştirilmesi, yazılımından daha pahalıdır diyebilirim. Tasarım şablonlarının kullanılmadığı ve profesyonel olmayan yazılımcılar tarafından oluşturulan programların bakımı imkansız ya da çok zordur. Bir firma için sahip olduğu veriler ne kadar önemli ise, verileri işlemek için kullandığı programlar ve bu programların bakımı ve geliştirilmesi de bir o kadar önemlidir.

Günümüzde firmalar sahip oldukları verileri kullanmak, saklamak ve işlemek için web tabanlı programlar kullanmaktadırlar. Web tabanlı programların bakımı ve geliştirilmesi masaüstü programlardan daha kolay ve ulaşılan kullanıcı kitlesi daha büyük olduğu için (kullanıcılar genelde

bir web tarayıcı (browser) ile çalışabilirler) tercih edilmektedirler. Web tabanlı programlar bugünkü standartlara göre üç katmandan oluşacak şekilde hazırlanır. MVC (model – view – controller) tasarım şablonunda da gördüğümüz gibi, sorumluluk alanları tanımlanmış katmanlar oluşturulur. Her katman kendisi için tanımlanmış görevi yerine getirmekle yükümlüdür ve işlevini yerine getirmek için diğer katmanlardan faydalanır.

Web projelerde uygulanan ilk katman gösterim (presentation) katmanıdır. Bu katmanda veriler üzerinde işlem yapılmaz. Veriler üzerinde işlem diğer katmanlar tarafından gerçekleştirilir. Gösterim katmanı başka bir katmanda hazırlanmış olan verilerin kullanıcıya gösterimi için kullanılır. Bu katmanda JSP ve Servlet gibi gösterim teknolojileri kullanılarak edinilen veriler sunulur.

Gösterim katmanına veriler işletme katmanı (business) tarafından sağlanır. İşletme katmanında veriler üzerinde yapılacak işlemler tanımlanır. Bunlar Java sınıflarında oluşturulan metotlardır. Business metodları olarak bilinen bu birimlerde, firmanın veriler üzerinde yapmak istediği işlemler implemente edilerek, istenilen neticeler elde edilir.

Gösterim katmanı için gerekli veriler veri depolama / edinme (persistence) katmanı tarafından sağlanır. Bu katmanın görevi JDBC ya da Hibernate gibi teknoloji ile veri tabanında yer alan verileri edinmek ve istenilen verileri veri tabanında depolamaktır.

Katmanlar arası iletişim tanımlanmış interface sınıflar üzerinden gerçekleşir. Örneğin gösterim katmanı işletme katmanında bulunan bir facade (cephe) interface üzerinden istediği verileri elde edebilir. Gösterim katmanı, işletme katmanı sadece bir interface sınıfından oluşuyormuş gibi düşünülerek, bu interface sınıfına karşı programlandığı taktirde, iki katman arasında esnek bir bağ oluşur. Bu durumda işletme katmanı, dış dünyaya sunduğu facade interface sınıfını istekleri doğrultusunda implemente ederek, gösterim katmanını etkilemeden çalışma tarzını tanımlayabilir. Facade interface sınıfında tanımlanmış metotlar değişmediği sürece, işletme katmanında yapılacak değişiklikler gösterim katmanını etkilemez. Sadece bu şekilde hazırlanmış bir program, gelecekte meydana gelen değişikliklere ayak uydurabilir yapıda olabilir. Tüm kodun bir katman içinde implemente edilmesi, kullanılan sınıflar arasındaki bağı yükselteceği gibi, bu kodun ilerdeki bakımını güçleştirir.

Otel Rezervasyon Programı

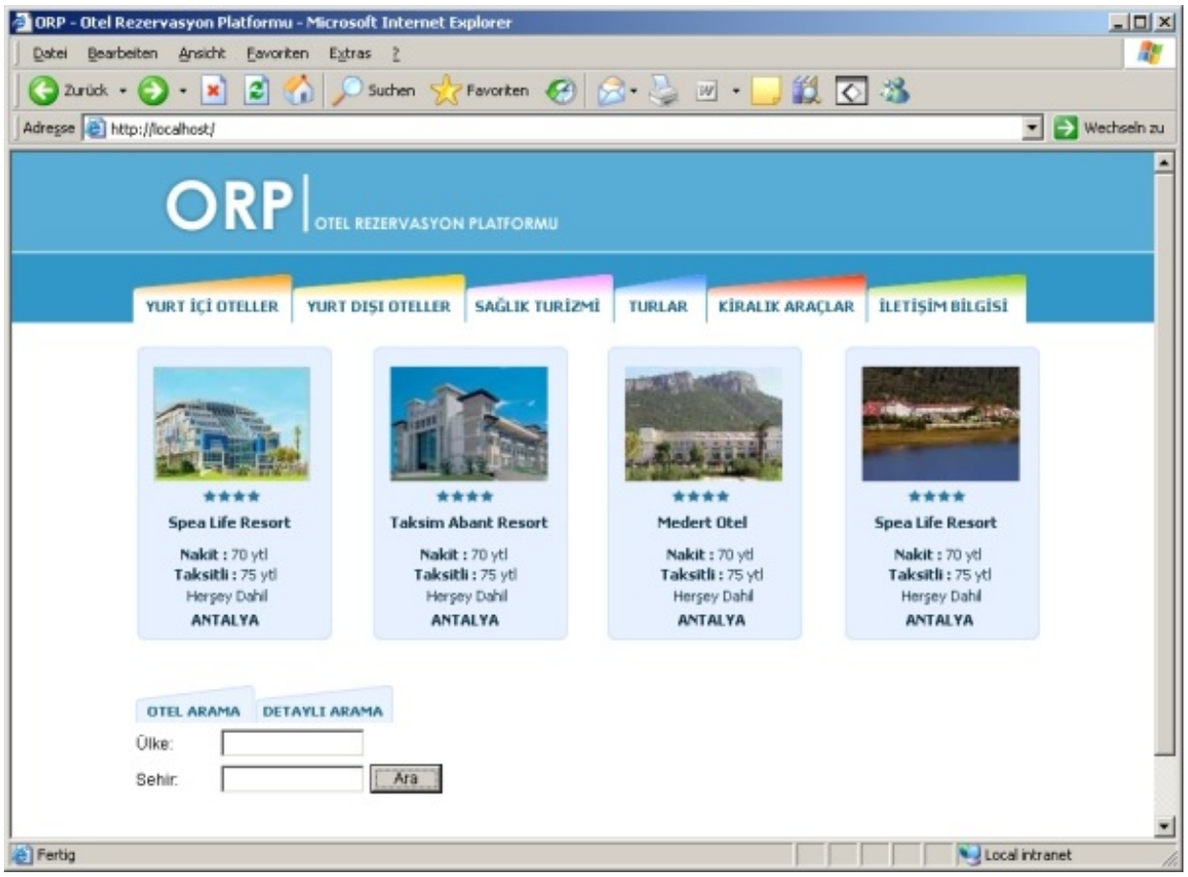
Bu bölümde web tabanlı bir otel rezervasyonu programgramı geliştireceğiz.

Tasarımını yapacağımız otel rezervasyon sistemi, bir otelin sahip olduğu ve müşterilerine oda

rezervasyonu için sunduğu bir program değildir. Program daha ziyade, birden fazla oteli bünyesinde barandıran, otellerden bağımsız, üçüncü bir şahıs/firma tarafından işletilen bir otel rezervasyon platformudur. Platform, müşteri bilgilerini ve rezervasyonlarını otomatik olarak arka planda istenilen otele iletebilir.

Otel rezervasyon programı aşağıda yer alan özelliklere sahip olacaktır:

- İlgi duyan şahıslar internet üzerinden otel rezervasyon websitesine bağlanarak, ülke, şehir ve fiyat kriterleri bazında otel arama yapabilirler.
- İstenilen kriterde bir otel bulunduğu taktirde, bu otel otomatik rezervasyon sistemi ile rezerve ettirilebilir. Otomatik rezervasyon esnasında otel rezervasyon platformu müşteri bilgilerini seçilen otele iletir. Bunu yapabilmesi için otel rezervasyon platformu ile seçilen otel arasında elektronik bir köprünün oluşturulası gerekmektedir. Bunu gerçekleştirmenin en kolay yolu, müşteri bilgilerini e-posta aracılığı ile seçilen otele göndermek olacaktır.
- Otel rezervasyon platform yöneticisi sisteme yeni oteller ekliyebilir ve silebilir.

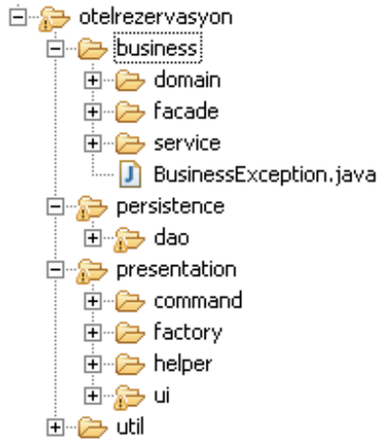


Resim 2

Not: Bu bölümde sadece otel arama modülünü implemente edeceğiz. Rezervasyon gibi diğer modüllerin implementasyonunu alıştırma olarak okuyucuya bırakıyorum. Arama modülü örnek

alınarak, çok rahatlıkla diğer modüller implemente edilebilir.

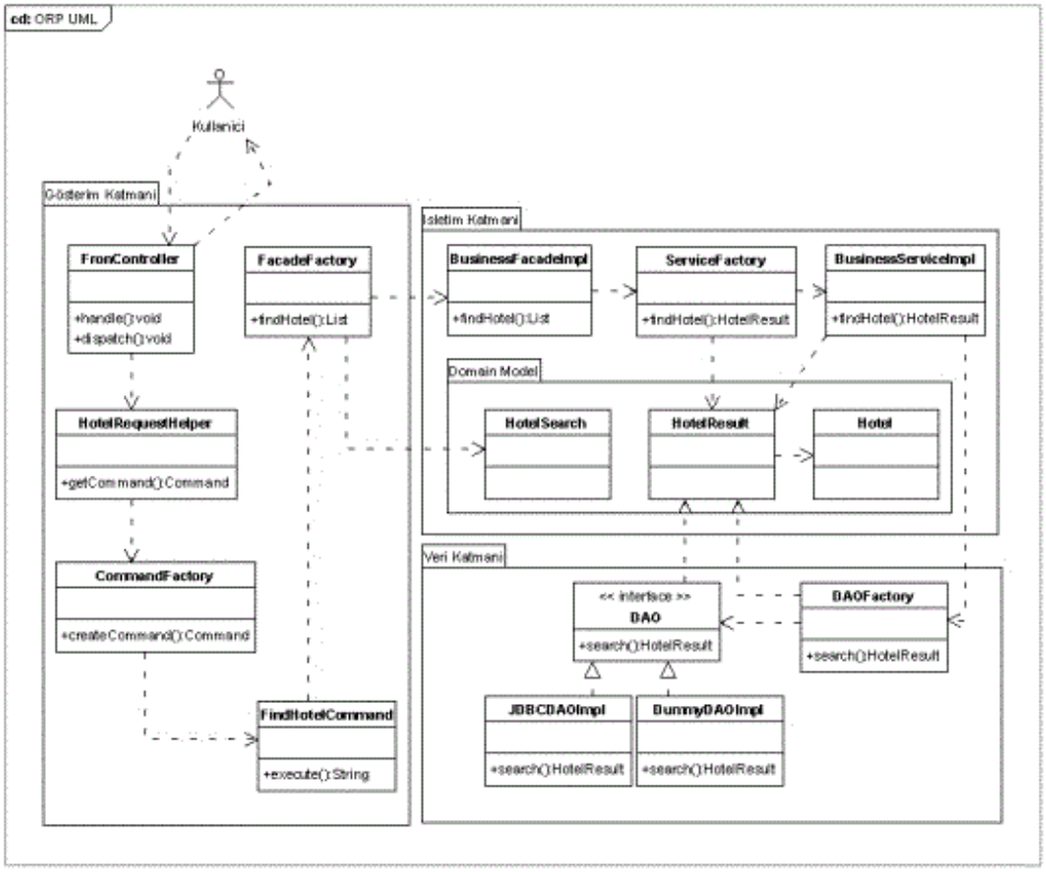
Program aşağıda yer alan Java paket yapısına sahiptir.



Resim 3

Üç katmanlı bir program yazılımı yapılacağı için, paket isimleride katmanlara göre seçildi.

UML Diagramı



Resim 4

UML diyagramında üç katman için üç ayrı Java paketin kullanıldığını görüyoruz. Bu şekilde proje için oluşturulan sınıfları ait oldukları katmanların paketleri içinde gruplamış oluyoruz. Şimdi kullanılan sınıfları yakından inceliyelim.

Gösterim Katmanı Sınıfları

Bu katman bünyesindeki sınıfların görevi, kullanıcının girmiş olduğu bilgileri, işletim katmanına iletmek ve işletim katmanından gelen veriler doğrultusunda, bu verilerin gerekli JSP sayfalarında gösterilmesini sağlamaktır. Bu katmanın sınıfları kesinlikle veriler üzerinde değişiklik yapmazlar ve işletim katmanında olması gereken işlemleri ihtiva etmezler. Örneğin JSP sayfaları içinde kesinlikle Java kodu kullanılarak, veri tabanından veri edinilmez, çünkü bu durum JSP sayfalarını doğrudan veri tabanına bağımlı kılar ve bakımı güçleştirir.

Gösterim katmanında yer alan sınıflar:

- ***FrontController***: Bir servlet sınıfı olarak implemente edilmiştir. Tüm kullanıcı istekleri (http request) handle() metoduna ulaşır. Dispatch() metodu ile herhangi bir JSP sayfasına yönlendirme yapabilir.
- ***RequestHelper***: İnterface sınıf olarak tanımlanmıştır. HotelRequestHelper sınıfı, RequestHelper interface sınıfının implementasyonudur. FronController RequestHelper interface sınıfını kullanarak, isteği cevaplamak için kullanılan command nesnesine ulaşır. Burada RequestMapper tasarım şablonu da kullanılabilir.
- ***CommandFactory***: HotelRequestHelper tarafından kullanıcı istediğine (request) cevap vermek için gerekli command nesnesini oluşturmak için kullanılır.
- ***Command***: Her kullanıcı isteği bir command nesnesine dönüştürülür ve execute() metodu ile gerekli işlem yapılır. Örneğin bir otel bulmak için FindHotelCommand implementasyon sınıfı kullanılır.
- ***FacadeFactory***: Command nesnesi FacadeFactrory sınıfını kullanarak, işletme katmanı üzerinde gerekli işlemleri yapar. FacadeFactory işletme katmanında bulunan BusinessFacade implementasyon nesnesini oluşturur ve iki katman arasındaki bağlantıyı gerçekleştirir.
- ***HotelSearch***: Domain modeline ait olan bu sınıf bir otel için arama kriterlerini ihtiva eder. Web arayüzü üzerinden arama yapıldığında girilen otel bilgileri HotelSearch içinde yer alır.
- ***HotelResult***: Domain modeline ait olan bu sınıf, arama işleminde bulunan otelleri bir ArrayList içinde tutar.
- ***Hotel***: Domain modeline ait olan bu sınıf bir oteli modellemek için kullanılır.

İşletme Katmanı Sınıfları

Gösterim katmanının kullanıcıdan edindiği veriler bu katmana iletilir ve kullanıcının istediği işlemler uygulanır. İşlem sonuçları tekrar gösterim katmanına iletir. Veri edinme, veri değiştirme ve veri saklama bu katman üzerinden gerçekleşir.

İşletme katmanında yer alan sınıflar:

- ***BusinessFacade***: İşletme katmanına sadece BusinessFacade üzerinden ulaşılır. Gösterim ve işletme katmanı arasındaki bağımlılığı azaltmak ve işletme katmanına girişleri kontrol etmek için kullanılır. BusinessFacadeImpl implementasyon sınıfıdır.
- ***BusinessService***: İşletme katmanının sunduğu hizmetler bu interface sınıfında tanımlanır. BusinessServiceIml implementasyon sınıfıdır. BusinessFacade, kendisine gelen istekleri BusinessService implementasyon sınıfına delege eder. Değişik implementasyonlar kullanılabileceği için bir interface olarak tanımlanmıştır. Böylece BusinessFacade ve BusinessService arasındaki bağ esnek tutulur. İmplementasyon sınıfında meydana gelebilecek değişiklikler BusinessFacade komponentini etkilemez. Görüldüğü gibi bir katman içinde de tasarım şablonları uygulanarak, katman içinde kullanılan sınıflar arası bağ azaltılabilir.
- ***ServiceFactory***: BusinessService implementasyonu oluşturmak için BusinessFacade tarafından kullanılır.

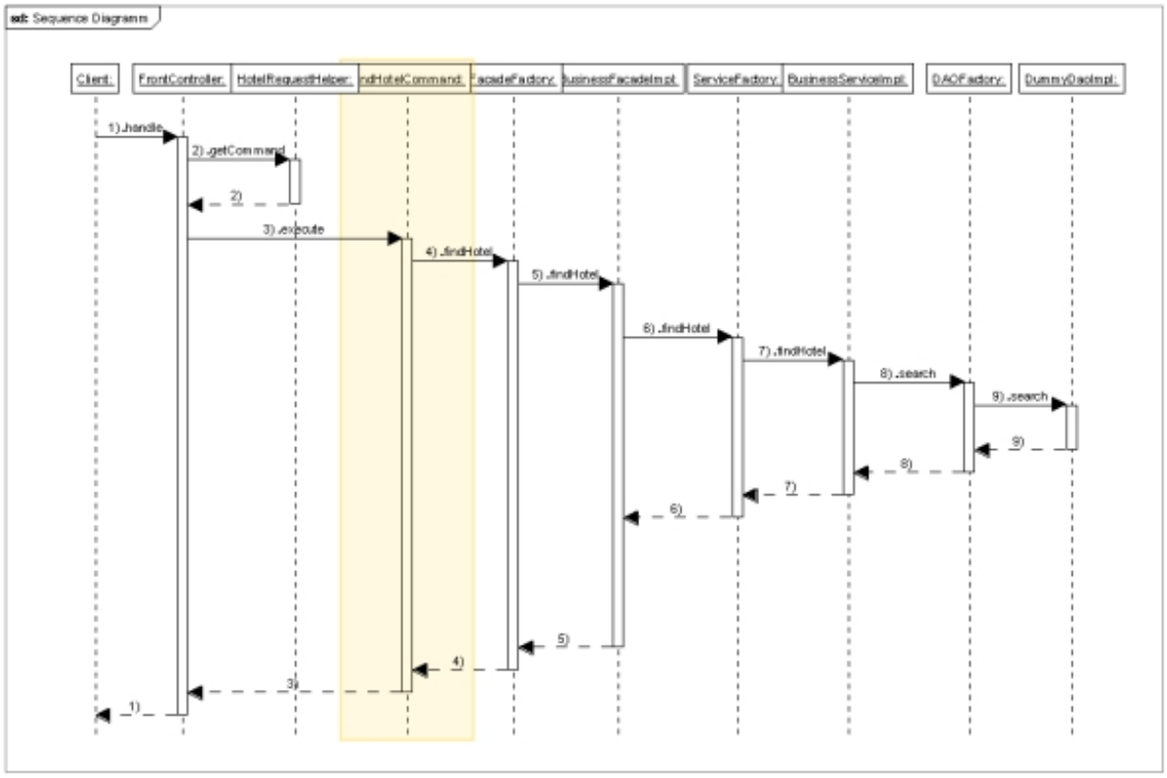
Veri Depolama / Edinme Katmanı

Veri tabanında verileri okuma, değiştirme ve depolama işlemleri bu katmanda gerçekleşir. DAO interface sınıfı sistem için kullanılan veri deposu mekanizmasını diğer katmanlar için transparen hale getirir. Veri deposu olarak bir veri tabanı ya da bir metin dosyası (txt) kullanılabilir. Nasıl bir veri deposu kullanıldığını işletme katmanı, DAO interface sınıfını kullandığı için bilmek zorunda değildir. Bu bize, işletme katmanını etkilemeden, sistem için kullanılan veri deposunu değiştirme imkanını tanır. DAO gibi tasarım şablonlarının neden kullanıldığını bu örnekte çok iyi görmüş oluyoruz. Amacımız, kısa zamanda karmaşık bir yapıya ulaşan projelerde, sistemin bütününü oluşturan modüleri birbirlerini etkilemiyecek şekilde modifike edebilir hale getirmektir. Tasarım şablonları yardımı ile her modül, diğerlerine zarar vermeden başka bir implementasyon ile değiştirilebilir hale geldiği taktirde, yazılan kodun geliştirilmesi ve bakımı çok daha kolaylaşır.

Bu katmanda kullanılan sınıflar:

- ***DAO***: Veri tabanı üzerinde yapılabilecek işlemler bu interface sınıfında tanımlanır. DummyDAOImpl implementasyon sınıfıdır.
- ***DAOFactory***: Bir DAO nesnesi oluşturmak için kullanılır. BusinessServiceImpl sınıfı DAOFactory üzerinden veri tabanı işlemlerini yapar.

UML Dizge (Sequence) Diagramı



Resim 5

UML dizge diyagramında sınıflar arası interaksiyonu görebiliyoruz. Buna göre bir kullanıcının isteği (request) FrontController sınıfının handle() metoduna ulaşır. FrontController sınıfı HotelRequestHelper sınıfından gerekli command nesnesini alır ve bu nesnenin execute() metodunu koşturur. Otel arama, ekleme, silme ve rezervasyon için değişik Command sınıfları mevcuttur. HotelRequestHelper sınıfı, HttpServletRequest içinde gelen action değişikeni ile hangi Command nesnesinin kullanılacağını tespit eder.

```
// Kod 1

<form name="form1" method="post" action="FrontController">

<table width="300" border="0" cellspacing="0" cellpadding="2">
  <tr>
    <td>
      <div align="left"><span class="text10">Ülke:</span></div>
    </td>
    <td><span class="text10">
      <input type="Text" name="country" size="15"
            class=text10 maxlength="15" value=""> </span>
```

```
    </td>
  </tr>
  <tr>
    <td> </td>
    <td> </td>
  </tr>
  <tr>
    <td>
      <div align="left">Sehir:</div>
    </td>
    <td><input type="text" name="city" size="15" class=text10
            maxlength="50" value=""></td>
  </tr>
  <tr>
    <td> </td>
    <td> </td>
  </tr>
  <tr>
    <td> </td>
    <td><input type="submit" name="Submit2" value="   Ara   "
            class=button></td>
  </tr>
  <tr>
    <td><input type="hidden" name="action" value="ara"></td>
    <td> </td>
  </tr>
</table>
</form>
```

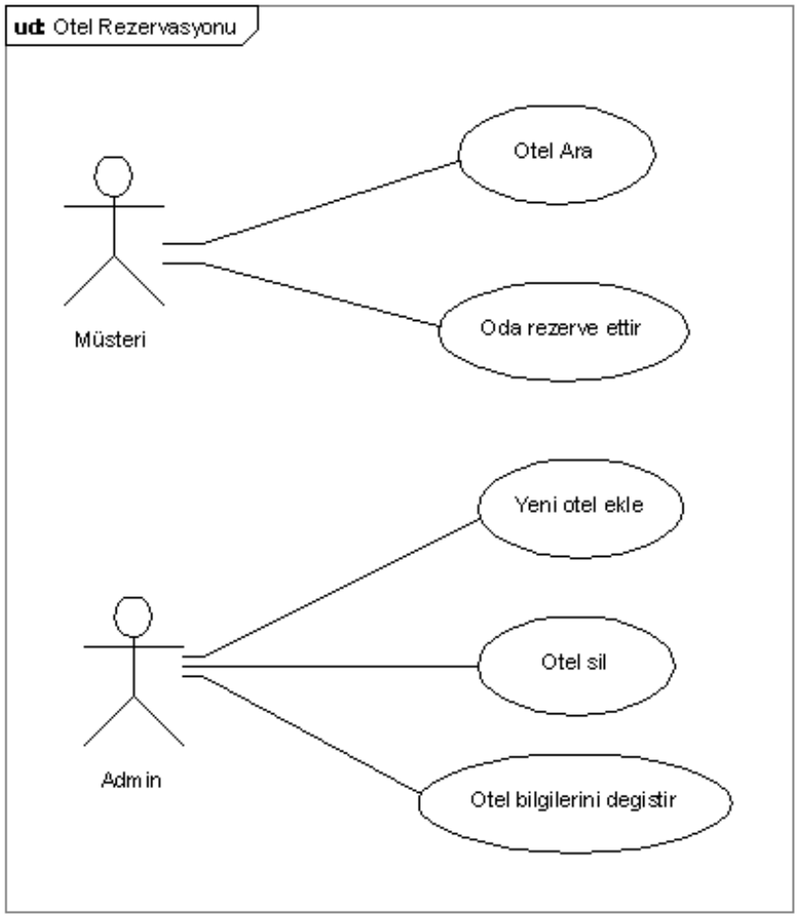
Otel arama işlemi yukarıda yer alan html formu ile yapılır. Form içinde action isminde gizli (hidden) bir parametre, hangi Command nesnesinin kullanılması gerektğini belirtir.

```
<input type="hidden" name="action" value="ara">
```

Bir otel arama işlemi yaptığımızı düşünürsek, HotelRequestHelper sınıfı request.getParameter("ara") verisi yardımı ile FindHotelCommand nesnesini oluşturur ve kullanılmak üzere FrontController sınıfına geri verir.

UML Kullanım Senaryosu (Use Case) Diagramı

UML kullanım senaryosu diyagramı otel rezervasyon programının dış dünyaya sunduğu hizmetleri göstermekte. Müşteri ve Admin isminde iki aktörümüz bulunuyor. Müşteri, arama ve rezerve ettirme fonksiyonlarını kullanabilir. Admin, sisteme yeni bir otel ekliyebilir, silebilir ya da mevcut bir otelin bilgilerini değiştirebilir.



Resim 6

Otel rezervasyonu platformunu üç katmanlı mimariyi örnek olarak tasarlıyacağız. İlk olarak gösterim (presentation) katmanı tasarımı ile başlıyoruz.

Gösterim (Presentation) Katmanı

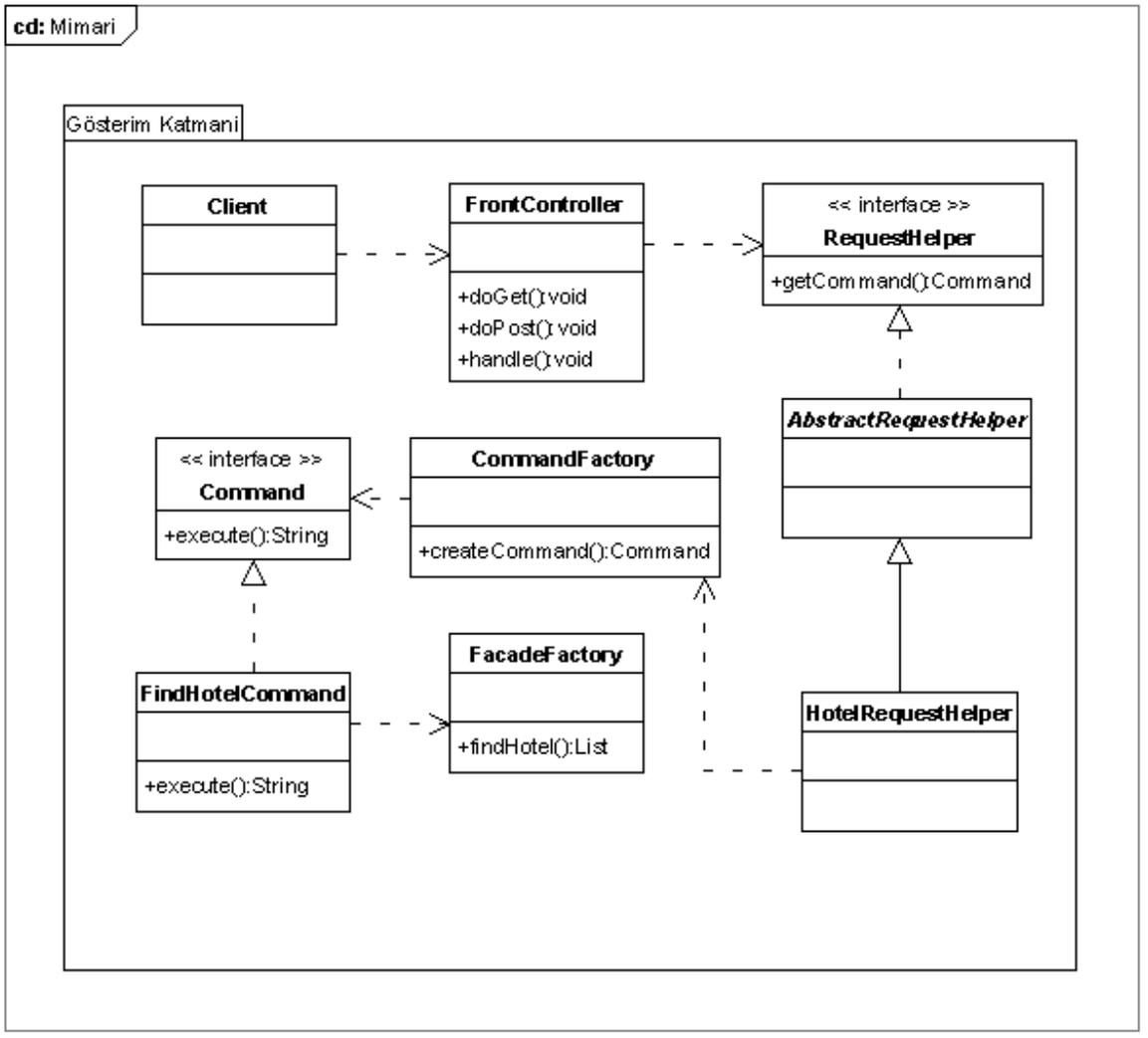
Klasik PHP veya JSP projelerinde genelde bütün kod sayfaların içine gömülür ve kısa bir zaman sonra sayfaların ve kodun bakımı imkansız hale gelir, çünkü yapılan en ufak bir değişiklik diğer sayfaları da etkiler. Kod, kopyala/ekle (copy/paste) metodu ile diğer sayfalara da sıçradığı için bir sayfa üzerinde yapılan değişiklikler, diğer sayfalar üzerinde de yapılmak zorundadır. Bunu önlemenin en kolay yolu, gösterim katmanını oluşturmak ve bu katmanın gösterim harici başka bir şey yapmasını engellemektir.

Buradan da anlaşılacağı gibi, amacımız, sorumluluk alanları ve işlevleri tanımlanmış katmanlar oluşturup, bu katmanlar arası "servis sunan – servis kullanan" ilişkisini oluşturmaktır. Ancak bu durumda katmanları ve işlevlerini isole edip, ilerde bakım ve genişletilmelerini, katmanın sunduğu hizmetleri kullanan diğer katmanları etkilemeden yapabiliriz.

Gösterim katmanının görevi, sistem kullanıcılarına sunduğu arayüzlerler aracılıği ile kullanıcılar ve sistem arasındaki interaksiyonu sağlamaktır. Bu katman JSP, Servlet ile yazılmış bir webtabanlı arayüz, ya da Swing, SWT ile yazılmış bir masaüstü programı olabilir. Kullanıcının istekleri, girmiş olduğu veriler işletme katmanına iletilir ve bu katmandan gelen sonuçlar kullanıcıya gösterilir. Bu katman sadece gösterimden sorumlu olduğu için gösterim (ingilizce "presentation") katmanı ismini taşır.

Gösterim katmanı adı üstünde sadece veri gösteriminden sorumludur. Kesinlikle kullanıcı verilerini değiştirmez, değerlendirmez ve veri tabanına bağlantı kurup, veri alıp, depolamaz.

Gösterim katmanı içinde, bu katmanın görevini yapabilmesi için Uml diyagramında gösterilen sınıflar yer almaktadır.



Resim 7

Bu katmanın merkezinde FrontController sınıfı yer almaktadır:

```
// Kod 2

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            presentation.controller;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            presentation.command.Command;
import com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            presentation.helper.HotelRequestHelper;
import com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            presentation.helper.RequestHelper;
import com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            util.Logger;

/**
 * FrontController tasarım şablonu ornegi. Http uzerinden
 * gelen tum isteklerler bu sınıf tarafından işlem gorur.
 *
 * Controller sınıfı gerekli command nesnesini bularak,
 * işlemi bu nesneye devreder ve neticeye gore gerekli jsp
 * sayfasına yonlendirme yapar.
 *
 */
public class FrontController extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * FrontController sınıfına gelen tum istekler bu metot
     * tarafından işlem gorur
     *
     */
    public void handle(final HttpServletRequest request,
            final HttpServletResponse response) {
        Logger.instance(this).debug("handle()");
        String nextPage = "";
        try {
```

```
            final RequestHelper helper = new HotelRequestHelper(
                    request, response);
            final Command command = helper.getCommand();
            nextPage = command.execute(helper);
            dispatch(request, response, nextPage);
        }
        // CommandException ve IOExcepton oluşabilir.
        // Bu durumda error.jsp sayfasına yï¿½nlendiriyoruz.
        catch (final Exception e) {
            e.printStackTrace();
            try {
                dispatch(request, response, "/error.jsp");
            } catch (final Exception e1) {
                e1.printStackTrace();
            }
        }
    }

    /**
     * Jsp sayfaları arasında yonlendirme yapmak için kullanılır.
     *
     */
    private void dispatch(final HttpServletRequest request,
            final HttpServletResponse response,
            final String page)
                    throws ServletException, IOException {
        Logger.instance(this).debug("dispatch()");
        final RequestDispatcher dispatcher = getServletContext()
                .getRequestDispatcher(page);
        dispatcher.forward(request, response);
    }

    /**
     * FrontController bir servlet sınıfı olduğu için GET
     * metodu ile gelen tum istekler bu metot tarafından
     * işlem gorur.
     */
    @Override
    public void doGet(final HttpServletRequest request,
            final HttpServletResponse response) {
        Logger.instance(this).debug("doGet()");
        handle(request, response);
    }

    /**
     * FrontController bir servlet sınıfı olduğu için POST
     * metodu ile gelen tum istekler bu metot tarafından
```

```
     * işlem gorur
     */
    @Override
    public void doPost(final HttpServletRequest request,
            final HttpServletResponse response) {
        Logger.instance(this).debug("doPost()");
        handle(request, response);
    }
}
```

FrontController sınıfı javax.servlet.http.HttpServlet sınıfını genişlettiği için bir servlettir. Kullanılan Http GET ya da POST metoduna göre otomatik olarak servletin doGet() ya da doPost() metodu işlem görür. FrontController.doGet() ve doPost() metotlarında görüldüğü gibi handle() metodu kullanılarak, kullanıcı isteğine cevap verilmektedir.

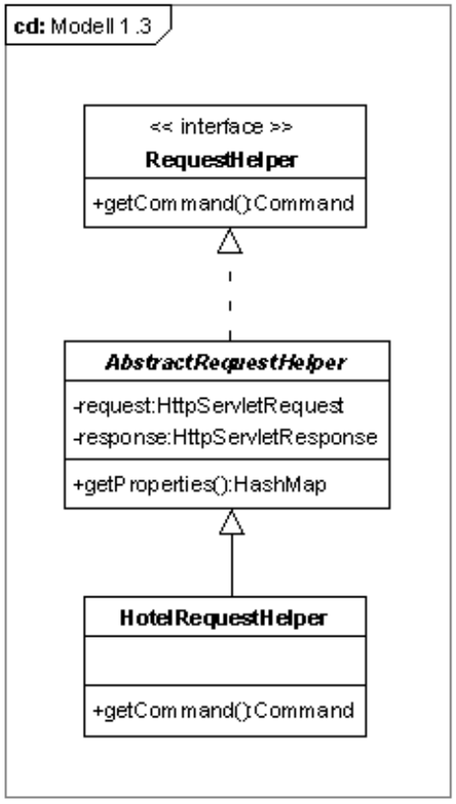
Kullanıcının isteklerini cevaplayabilmek için Command tasarım şablonunu kullandık. FrontController command nesnelerinin oluşumu RequestHelper isimli sınıf tarafından yapılmasını sağlar. Burada da tekrar gördüğümüz gibi her işlemi FrontController sınıfı bünyesinde yer alan metotlar ile yapmak yerine, belirli işlemleri RequestHelper gibi başka sınıflara taşıyarak, sorumluluk alanlarını daraltıyor ve bakımı ve test edilmesi kolay, daha az satırdan oluşan sınıflar oluşturuyoruz.

```
// Kod 3

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            presentation.helper;

import java.util.HashMap;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            presentation.command.Command;
import com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            presentation.command.CommandException;

/**
 * RequestHelper interface sınıfı
 *
 */
public interface RequestHelper {

    public HttpServletRequest getRequest();
```

```
    public HttpServletResponse getResponse();

    public Command getCommand() throws CommandException;

    public HashMap<String, String> getRequestProperties();
}
```

RequestHelper sınıfını bir interface olarak tanımlıyoruz. İleride değişik implementasyonlar kullanabilmek için bu gerekli bir işlemdir. RequestHelper interface sınıfında yer alan getCommand() metodu ile Command nesneleri oluşturulur.

Bir interface sınıfı değişik şekillerde altsınıflarca implemente edilebilir. Birden fazla implementasyon olabileceği için kullanılan ortak değişkenlerin ve metotların bir soyut sınıf bünyesinde toplanmasında fayda vardır. Bu amaçla interface sınıfı ile implementasyon sınıfları arasına, ortak değişken ve metotları ihtiva eden bir soyut sınıf yerleştirilir.



Resim 8

AbstractRequestHelper sınıfı bünyesinde request ve response gibi sınıf değişkenleri yanı sıra, request içinde bulunan değerlerin yer aldığı bir HashMap oluşturan getProperties() metodu yer almaktadır. İmplementasyon sınıfı olan HotelRequestHelper sınıfı bu değişken ve metotları

AbstractRequestHelper sınıfını genişlettiği için kullanabilir.

```
// Kod 4

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            presentation.helper;

import java.util.Enumeration;
import java.util.HashMap;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * AbstractRequestHelper sınıfı.
 *
 */
public abstract class AbstractRequestHelper implements RequestHelper {
    private HttpServletRequest request;
    private HttpServletResponse response;
    private HashMap<String, String> requestProperties;

    public AbstractRequestHelper(final HttpServletRequest request,
            final HttpServletResponse response) {
        setRequest(request);
        setResponse(response);
        setRequestProperties(getProperties(request));
    }

    @Override
    public HttpServletRequest getRequest() {
        return request;
    }

    public void setRequest(final HttpServletRequest request) {
        this.request = request;
    }

    @Override
    public HttpServletResponse getResponse() {
        return response;
    }

    public void setResponse(final HttpServletResponse response) {
        this.response = response;
    }
```

```
    @Override
    public HashMap<String, String> getRequestProperties() {
        return requestProperties;
    }

    private HashMap<String, String> getProperties(
            final HttpServletRequest request) {
        final HashMap<String, String> properties =
                new HashMap<String, String>();
        try {
            final Enumeration<String> en = request.getParameterNames();
            for (; en.hasMoreElements();) {
                final String key = en.nextElement();
                final String value = request.getParameter(key);
                properties.put(key, value);
            }
        } catch (final Exception e) {
            e.printStackTrace();
        }
        return properties;
    }

    public void setRequestProperties(final HashMap<String, String>
                    requestProperties) {
        this.requestProperties = requestProperties;
    }
}


// Kod 4.1

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
presentation.helper;


/**
 * HotelRequestHelper sınıfı
 *
 */
public class HotelRequestHelper extends AbstractRequestHelper {

    public HotelRequestHelper(final HttpServletRequest request,
            final HttpServletResponse response) {
        super(request, response);
    }
```

```
    @Override
    public Command getCommand() throws CommandException {
        Logger.instance(this).debug("getCommand()");
        return CommandFactory.instance().createCommand(
                getRequestProperties().get("action"), getRequest());
    }

}
```

HotelRequestHelper bünyesinde getCommand() metodunu implemente ederek, gerekli command nesnesinin oluşturulmasını sağlıyoruz. Bu amaçla CommandFactory sınıfı kullanılmaktadır.

```
// Kod 5

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
                presentation.command;

public class CommandFactory {

    private static CommandFactory factory = new CommandFactory();

    private final HashMap<String, String> commands =
            new HashMap<String, String>();

    private CommandFactory() {
        try {
            final ResourceBundle bundle = ResourceBundle
                    .getBundle("otelrezervasyon/command");
            final Enumeration<String> en = bundle.getKeys();
            for (; en.hasMoreElements();) {
                final String key = en.nextElement();
                commands.put(key, bundle.getString(key));
            }
        } catch (final Exception e) {
            e.printStackTrace();
            throw new RuntimeException("");
        }

    }

    public static CommandFactory instance() {
        return factory;
    }

    public Command createCommand(final String action,
            final HttpServletRequest request)
```

```
                    throws CommandException {
        Command command = null;
        try {
            command = (Command) Class
                    .forName(commands.get(action))
                    .newInstance();

            Logger.instance(this).debug("Command type: "
                    + command.getClass().getName());
            command.setRequest(request);
        } catch (final Exception e) {
            e.printStackTrace();
            throw new CommandException(e);
        }
        return command;
    }
}
```

Kullandığımız fabrika tasarım şablonu ile bir command nesnesinin oluşturulması HotelRequestHelper sınıfı için transparen bir hale gelmiştir. CommandFactory sınıfı bünyesinde singleton tasarım şablonunu kullanarak, sistem bünyesinde sadece bir tane CommandFactory nesnesinin kullanılmasını sağlıyoruz. Bir command nesnesinin oluşturulabilmesi için kullanıcı sınıfın bir String değişkeni ile istediği command tipini belirtmesi gerekmektedir. HotelRequestHelper sınıfı getCommand() metodu bünyesinde bu işlemi şu şekilde yapmaktadır:

```
return CommandFactory.instance().createCommand(
    (String)getRequestProperties().get("action"),getRequest());
```

Kullanıcı tarafından otel araması yapıldığı zaman, request parametresi olarak "action = ara" değeri FrontController sınıfına ulaşır. HotelRequestHelper.getRequestProperties().get("") ile action anahtarının değerine ulaşabiliriz. Action anahtarının değeri kullanılarak, CommandFactory tarafından gerekli command nesnesi oluşturulur.

CommandFactory sınıfının konstrüktörüne baktığımız zaman

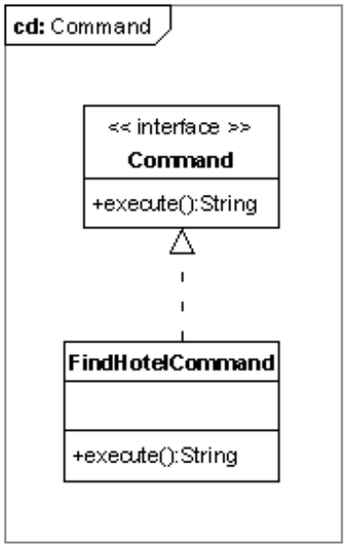
```
ResourceBundle.getBundle("otelrezervasyon/command");
```

şeklinde bir satırın olduğunu görmekteyiz. properties/otelrezervasyon dizininde command.properties isminde bir dosya bulunmaktadır. Bu dosya şu yapıya sahiptir:

```
ara = com.pratikprogramci.designpatterns.otelrezervasyon
            .presentation.command.FindHotelCommand
```

Değişik işlemler için değişik tipte command sınıfları tanımlamış olabiliriz. Bunlardan bir tanesi arama işleminde kullanılan FindHotelCommand sınıfıdır. CommandFactory sınıfını tekrar derlemek zorunda kalmadan, command.properties dosyası içinde değişiklik yaparak, sistemde kullanılan command sınıflarının adedi çoğaltabilir ya da azaltabiliriz. Bu değişikliklerden CommandFactory sınıfının etkilenmesini önlemek için command.properties şeklinde bir dosya tanımlayarak, program çalıştığı zaman bu dosyada yer alan değerlerin CommandFactory tarafından okunmasını sağlıyabiliriz. Bu dosya içinde yer alan anahtar ve değer tanımları ile gerekli command sınıfı bulunur ve command nesnesi oluşturulur. CommandFactory sınıfının konstrüktöründe command.properties dosyasında bulunan tüm satırlar okunarak, sınıf değişkeni olan commands isimli HashMap içine yerleştirilir. Daha sonra commands.get(anahtar) ile istenilen command sınıfı tipi edinilebilir.

CommandFactory bünyesinde command nesneleri createCommand() metodunda oluşturulur. CommandFactory sınıfı kendisinin tanımladığı commands sınıf değiskeni üzerinden command.properties içinde tanımlanmış olan anahtar ve değerlere ulaşabilir. Örneğin otel arama işlemi için "action = ara" şeklinde bir değer gönderildi ise, CommandFactory sınıfı, commands değişkeninde ara isminde bir değerin olup, olmadığına bakar ve Class.forName() işlemi ile command nesnesini oluşturur. commands.get("ara") com.pratikprogramci.designpatterns. otelrezervasyon.presentation.command.FindHotelCommand değerini geri verecektir. Bu aranan Command sınıfının ismidir. ClassForName() yapılarak, FindHotelCommand sınıfından bir nesne üretilebilir.



Resim 9

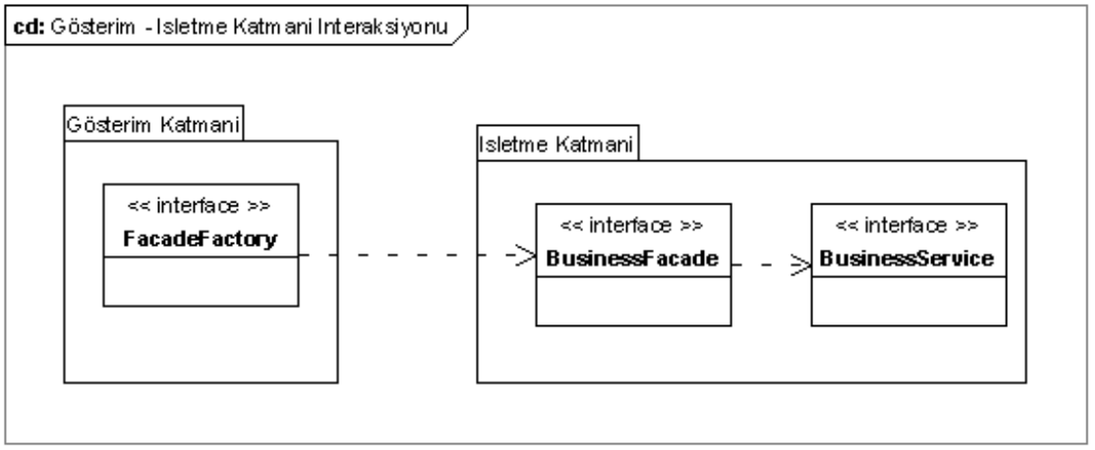
Command isminde bir interface sınıf tanımlıyoruz. Bu interface sınıfında execute() isminde bir

metot bulunmaktadır. Tüm altsınıflar (örneğin FindHotelCommand) bu metodu implemente ederler.

```
// Kod 6

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            presentation.command;

public interface Command {
    public String execute(RequestHelper helper)
            throws CommandException;

    public void setRequest(HttpServletRequest request);

}

// Kod 7

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            presentation.command;

public class FindHotelCommand implements Command {

    private HttpServletRequest request = null;

    public HttpServletRequest getRequest() {
        return request;
    }

    @Override
    public void setRequest(
            final HttpServletRequest request) {
        this.request = request;
    }

    @Override
    public synchronized String execute(
            final RequestHelper helper)
                    throws CommandException {
        Logger.instance(this).debug("execute()");
        String result = "";
        try {
            final HotelSearch hotel = new HotelSearch();
            hotel.setCountry(
                    getRequest().getParameter("country"));
            hotel.setCity(
```

```
                    getRequest().getParameter("city"));
            final List<Hotel> list = FacadeFactory
                    .instance().findHotel(hotel);
            getRequest().setAttribute("result", list);
            result = "/result.jsp";
        } catch (final Exception e) {
            e.printStackTrace();
            throw new CommandException(e);
        }
        return result;
    }
}
```

Gösterim katmanından işletme katmanına geçiş FacadeFactory sınıfı üzerinden gerçekleşir.



Resim 10

```
// Kod 8

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
        presentation.factory;

public class FacadeFactory {
    public static FacadeFactory factory;
    private BusinessFacade businessFacade = null;

    public BusinessFacade getBusinessFacade() {
        return businessFacade;
    }

    public void setBusinessFacade(
            final BusinessFacade businessFacade) {
        this.businessFacade = businessFacade;
```

```
    }

    private FacadeFactory() {
        initFactory();
    }

    public static FacadeFactory instance() {
        if (factory == null) {
            factory = new FacadeFactory();
        }
        return factory;
    }

    private void initFactory() {
        try {
            final String facade = ResourceBundle
                    .getBundle(
                            "otelrezervasyon/presentation")
                    .getString("businessfacade");
            businessFacade = (BusinessFacade) Class
                    .forName(facade).newInstance();
        } catch (final Exception e) {
            e.printStackTrace();
            throw new RuntimeException();
        }
    }

    public List<Hotel> findHotel(final HotelSearch search)
            throws BusinessException {
        return businessFacade.findHotel(search);
    }

}
```

FindHotelCommand implemente ettiği execute() metodunda FacadeFactory sınıfını kullanarak, işletim katmanı ile bağlantı kurar. FacadeFactory sınıfı, değişik tipte BusinessFacade implementasyonları olabileceği bilgisine sahiptir. Hangi tip implementasyon kullanılacağı presentation.properties dosyasında yer alır. Bu dosya properties/otelrezervasyon dizininde yer almaktadır ve aşağıdaki içeriğe sahiptir:

```
businessfacade = com.pratikprogramci.designpatterns.otelrezervasyon
            .business.facade.BusinessFacadeImpl
```

FacadeFactory.initFactory() metodunda görüldüğü gibi, ResourceBundle sınıfı aracılığı ile

presentation.properties dosyası içinde yer alan businessfacade anahtarının değeri okunmakta ve akabinde işletme katmanının bir parçası olan BusinessFacade implementasyon nesnesi oluşturulmaktadır.

Gösterim katmanının işletim katmanı ile olan sınırında FacadeFactory sınıfı yer almaktadır. Bu noktadan itibaren kontrol işletim katmanına geçer. Gerekli işlemler işletim katmanı tarafından yerine getirildikten sonra netice bir nesne olarak tekrar gösterim katmanına iletilir.

Tekrar FrontController sınıfına dönelim. Bu sınıfın servlet olarak Tomcat gibi bir uygulama sunucusu içinde çalışabilir hale gelmesi için uygulamanın web.xml dosyasına ekleme yapılması gerekmektedir. Java dilinde hazırlanmış her web projesinde bir web.xml bulunmak zorundadır. Bu dosya projenin web/WEB-INF dizininde yer almaktadır.

```
// Kod 9

<servlet>
  <servlet-name>FrontController</servlet-name>
  <servlet-class>
    com.pratikprogramci.designpatterns.otelrezervasyon.
        presentation.controller.FrontController
  </servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>FrontController</servlet-name>
  <url-pattern>/FrontController/*</url-pattern>
</servlet-mapping>
```

Web.xml dosyasında FrontController isminde bir servlet tanımlıyoruz. http://localhost/FrontController adresine gelen tüm istekler (request) FrontController sınıfına iletilecektir.

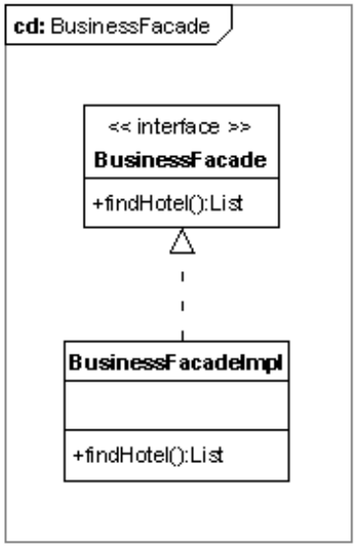
İşletme (Business) Katmanı

İşletme katmanı hazırlanan sistem içinde önemli bir rol oynamaktadır. Gösterim katmanının sadece veri gösterimi ve kullanıcı ile interaksiyonda kullanıldığını gördük. İşletim katmanında bulunan sınıflarda işletme mantığı (business logic) olarak isimlendirilen önemli metotlar yer alır. Programlamanın büyük bir bölümünü, işletme katmanı içinde yer lanan sınıflar oluşturur.

Bu katman içinde sunulan hizmetlere, katmanın bir parçası olan BusinessFacade üzerinden erişilir. Gösterim katmanında kullanıcılardan gelen istekleri (request) bir noktada toplayıp, değerlendirmek için FrontController tasarım şablonunu kullanmıştık. Facade tasarım şablonu da işletme

katmanının sunduğu hizmetlere erişimi bir noktada toplamak ve gösterim ve işletme katmanları arasındaki bağımlılığı azaltmak için kullanılır. Gösterim katmanı sadece BusinessFacade interface sınıfını kullanıp, işletme katmanı üzerinde işlem yapabildiği için facade tasarım şablonunu kullanarak, ilerde gösterim katmanının işletme katmanı üzerinde yapılacak değişikliklerden etkilenmesini engellemiş oluyoruz.

Gösterim katmanı, yapmak istediği işlemler için BusinessFacade sınıfının metotlarını kullanır.



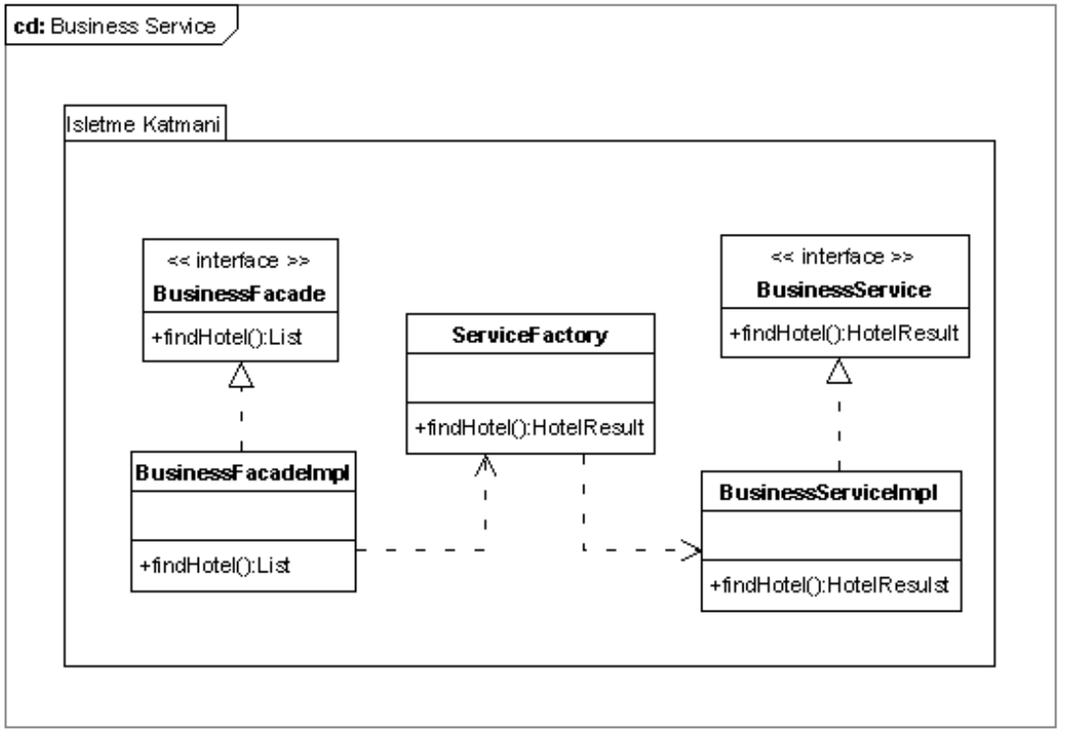
Resim 11

Gösterim katmanının bir parçası olan FindHotelCommand sınıfı FacadeFactory sınıfı aracılığıyla işletim katmanı ile bağlantı kurar.

```
// Kod 10

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            business.facade;

/**
 * Gosterim katmanı sadece BusinessFacade sınıfı uzerinden business katmanı
 * uzerinde işlem yapabilir. İki katman arasındaki işlemleri, facade tasarım
 * şablonu ile temiz bir şekilde birbirinden ayırmış oluyoruz.
 * Gosterim katmanı
 * sadece bu sınıfın metotlarını tanıdığı ve kullandığı için ilerde business
 * katmanı içinde yapılacak değişiklikler kesinlikle gosterim katmanını
 * etkilemiyecektir.
 *
 */
public interface BusinessFacade {
```

```
    /**
     * Otel bulmak için kullanılan metot.
     *
     * @param search
     *            arama kriterleri
     * @return List bulunan otel listesi
     * @throws BusinessException
     */
    public List<Hotel> findHotel(HotelSearch search)
            throws BusinessException;
}
```

BusinessFacade interface sınıfı gösterim katmanının kullanabileceği metodları ihtiva eder. Gösterim katmanı ve işletim katmanı arasındaki bağlantıyı daha esnek bir hal getirmek için, BusinessFacade sınıfını bir interface olarak tanımlıyoruz.



Resim 12

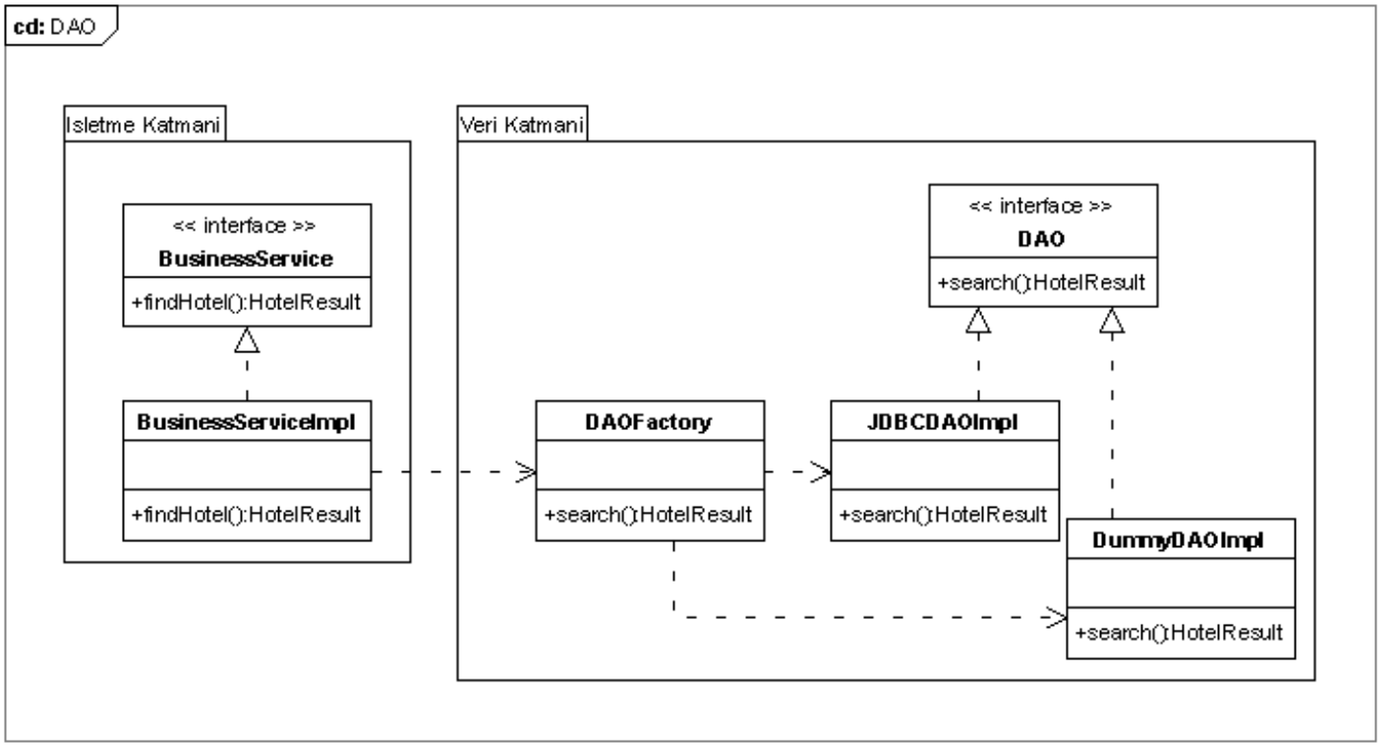
BusinessFacade sınıfını interface olarak tanımlanın beraberinde getirdiği bir avantaj vardır: Gösterim katmanının kullanabileceği tüm metotlar bir interface sınıfının arkasına gizlenmiş oluyor. BusinessFacade sınıfını bir interface olarak tanımlayarak, bu interface sınıfını kullanacak dış dünya (gösterim katmanı) ile bir nevi anlaşma yapmış oluyoruz. Bu anlaşmada BusinessFacade interface sınıfının açıklaması şöyledir: "Bunlar benim size sunduğum metot isimleridir, kullanabilirsiniz. Kendi bünyemde bu metotlar için değişik implementasyon sınıfları kullanabilirim.

Benden beklenen işlevi nasıl yerine getirdiğim dış dünya için önemli değildir. Dış dünya için önemli olan, sunduğum metot isimleri ve bu metot isimlerinin degişmemesidir. Metot gövdelerini kullandığım implementasyon sınıflarında istediğim şekilde değiştirebilirim ve kullanıcılarım bundan etkilenmez. Sadece metot isimleri değiştiğinde, kullanıcılar bu yeni metot isimlerini kullanmak zorundadırlar".

```
// Kod 11

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
    business.facade;

public class BusinessFacadeImpl implements BusinessFacade {

    /**
     * Arama işlemini yapan metot. ServiceFactory
     * uzerinden arama istegi işletme
     * katmaninda servis işlemlerini kontrol eden
     * BusinessService sınıfına
     * gonderilir.
     */
    @Override
    public List<Hotel> findHotel(final HotelSearch search)
            throws BusinessException {
        Logger.instance(this).debug("findHotel()");
        ArrayList<Hotel> list = null;
        try {
            final HotelResult result = ServiceFactory
                    .instance().findHotel(search);
            list = result.getList();
        } catch (final Exception e) {
            // Olusabilecek ServiceException BusinessException
            // olarak ust katmana gonderiliyor.
            throw new BusinessException(e);
        }
        return list;
    }
}
```

BusinessFacadeImpl sınıfında görüldüğü gibi BusinessFacade interface sınıfı tarafından tanımlanmış olan findHotel() metodu, BusinessFacadeImpl sınıfı tarafından implemente edilmiştir. BusinessFacadeImpl sınıfı ServiceFactory sınıfını kullanarak, gerçek işletme katmanı metotlarına ulaşır.

İşletmeni katmanının merkezinde BusinessService sınıfı yer almaktadır.



Resim 13

```
// Kod 12

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            business.service;

/**
 * Business katmanı tarafından sulunan metotlar BusinessService interface
 * sınıfında toplanır. ServiceFactory sınıfı aracılığı ile business katmanı
 * uzerinde işlem yapabilir.
 *
 */
public interface BusinessService {

    public HotelResult findHotel(HotelSearch search)
            throws ServiceException;

}
```

Bu interface sınıfı bünyesinde business metotları yer alır. BusinessFacade sınıfında olduğu gibi, BusinessService sınıfını interface olarak tanımlıyor ve BusinessServiceImpl sınıfında implemente ediyoruz. Buradaki amacımız, BusinessFacade ve BusinessService arasında esnek bir bağ

oluşturmak ve gerektiği zaman BusinessFacade etkilenmeden implementasyon sınıfını değiştirmektir.

BusinessFacade ve BusinessService arasındaki bağlantıyı ServiceFactory sınıfı üzerinden gerçekleştiyoruz. Bu durumda BusinessFacade için BusinessService kullanımı tamamen transparen bir hale gelmektedir. ServiceFactory sınıfını kod 13 de görmekteyiz.

```
// Kod 13

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            business.service;

/**
 *
 * Kullanılmak istenen BusinessService implementasyon sınıfı
 * properties/service.properties dosyasinda tanımlanır.
 *
 */
public class ServiceFactory {

    /*
     * Factory değişkeni
     */
    public static ServiceFactory factory;

    /*
     * BusinessService değişkeni.
     */
    private BusinessService service = null;

    /**
     * Singleton tasarım şablonunu kullanarak, factory sınıfından sadece bir
     * nesne oluşturup, kullanıyoruz. Singleton olabilmesi
     * için konstruktorlarin
     * private olması gerekiyor. Böylece başka bir sınıf new ServiceFactory()
     * kullanarak, bu sınıftan bir nesne oluşturulmaz.
     *
     */
    private ServiceFactory() {
        initFactory();
    }

    /**
     * Bu sınıftan oluşturulan tek nesneye ulaşmak için kullanılır
     *
```

```
     * @return ServiceFactory factory
     */
    public static ServiceFactory instance() {
        if (factory == null) {
            factory = new ServiceFactory();
        }
        return factory;
    }

    /**
     * Kullanmak istediğimiz BusinessService implementasyon sınıfını
     * business.properties içinde tanımlıyoruz. Bu metot içinde
     * business.properties içinde tanımlanmış olan businessservice anahtarı
     * kullanılarak, istenilen BusinessService implementasyon nesnesi
     * oluşturulur.
     *
     */
    private void initFactory() {
        try {
            final String facade = ResourceBundle
                    .getBundle("otelrezervasyon/business")
                    .getString("businessservice");
            service = (BusinessService) Class
                    .forName(facade).newInstance();
        } catch (final Exception e) {
            e.printStackTrace();
            throw new RuntimeException();
        }
    }

    /**
     * ServiceFactory sınıfı dis dunyaya findHotel metodunu sunar.
     * Kendi içinde işlemi BusinessService sınıfının findHotel()
     * metoduna delege eder. Bu
     * sayede ilerde BusinessService interface sınıfı uzerinde değişiklik
     * yapılsa ve findHotel() metodu findHotel() olarak değiştirilse bile,
     * BusinessFacade ServiceFactory sınıfını kullandığı için
     * bu değişiklikten
     * etkilenmez. Sadece ServiceFactory.findHotel() metodunda değişiklik
     * yapılması yeterli olacaktır.
     *
     * @param hotel
     *            HotelSearch
     * @return HotelResult
     * @throws ServiceException
     *             Exception
     */
```

```
    public HotelResult findHotel(final HotelSearch hotel)
            throws ServiceException {
        return getBusinessService().findHotel(hotel);
    }

    /**
     * service değişkeni için get metodu.
     *
     * @return service
     */
    private BusinessService getBusinessService() {
        return service;
    }

}
```

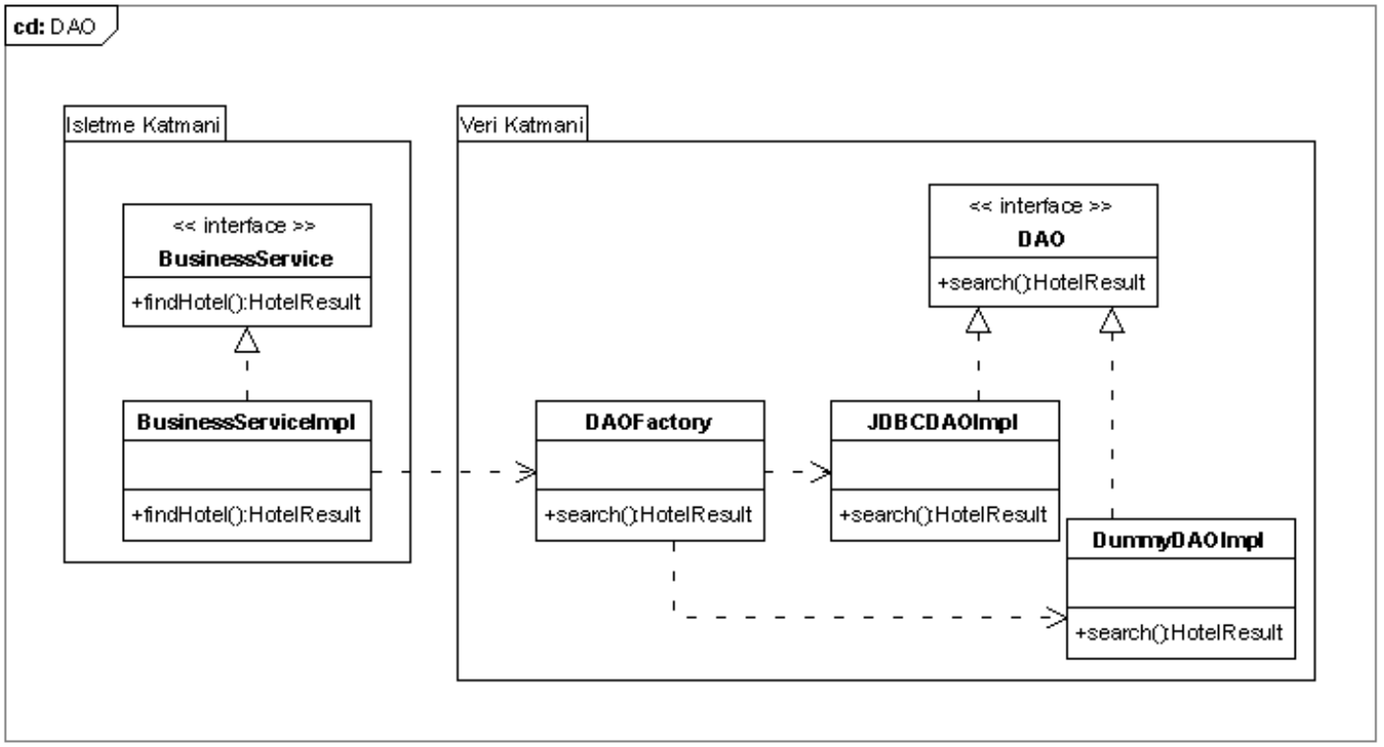
BusinessService bünyesinde bulunan metotlar BusinessServiceImpl sınıfında implemente edilir. Bu sınıf bünyesinde bulunan findHotel() metodu arama işlemini gerçekleştirmek üzere veri katmanında (persistence) bulunan DAOFactory sınıfını kullanır. DAOFactory işletim ve veri katmanı arasındaki bağlantıyı kuran sınıftır.

```
// Kod 14

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
        business.service;


/**
 * Business katmanı içindeki işlemler dis dunyaya interface olarak sunulur.
 * Sunulan bu işlemlerin implementasyonu BusinessSerbiceImpl sınıfında
 * gerçekleşir. Dis dunyayı etkilemeden BusinessService için değişik
 * implementasyonlar kullanılabilir.
 *
 */
public class BusinessServiceImpl
        implements BusinessService {

    /**
     * Arama yapmak için kullanılan metot.
     *
     */
    @Override
    public HotelResult findHotel(final HotelSearch hotel)
            throws ServiceException {
        Logger.instance(this).debug("findHotel()");
```

```
        HotelResult result = null;
        try {
            /**
             * Veri tabanına bağlanmak için DAOFactory kullanılır.
             */
            result = DAOFactory.instance().search(hotel);
        } catch (final Exception e) {
            e.printStackTrace();
            throw new ServiceException(e);
        }
        return result;
    }
}
```

Veri Depolama (Persistence) / Edinme Katmanı

Bu katmanın görevi, kullanılan veri tabanı ile bağlantı kurmak, istenilen verileri edinmek ve verileri veri tabanında depolamaktır. Bu işlem için DAO tasarım şablonu kullanılır. İşletme katmanı, DAO tasarım şablonu kullanıldığı için ne tür bir veri tabanı kullanıldığını bilmek zorunda değildir. Böylece işletim katmanı ile kullanılan veri tabanı arasındaki ilişki transparen hale gelmektedir. İşletim katmanını etkilemeden kullanılan veri tabanı her zaman değiştirebiliriz.



Resim 14

```
// Kod 15

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            persistence.dao;

/**
 * DAO interface sınıfı.
 *
 * Burada sadece search() metodunu tanımlıyoruz. Bu metodun gövdesi, bu
 * interface sınıfını implemente eden sınıf tarafından olusturulacaktir. Basit
 * bir implementasyon ornegi için DummyDAOImpl sınıfına bakınız.
 *
 */
public interface DAO {
    /**
     * Bir otel aramak için kullanılan metot.
     *
     * @param HotelSearch
     * @return HotelResult
     * @throws DAOException
     */
    public HotelResult search(HotelSearch hotel)
            throws DAOException;
}
```

DAO tasarım şablonunu kullanabilmek için DAO isminde bir interface sınıf tanımlıyoruz. Bu sınıf bünyesinde işletme katmanının kullanabileceği metotlar tanımlanır.

Kullanılan veri tabanı ve veri tabanı erişim teknolojisine göre (JDBC, Hibernate, Ibatis vs..) değişik tipte implementasyon sınıflar oluşturabiliriz. JDBCDAOImpl sınıfında JDBC teknolojisini kullanarak, DAO interface sınıfını implemente ediyoruz:

```
// Kod 16

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            persistence.dao;



/**
 * DAO interface sınıfını implemente eder ve JDBC kullanarak veri tabanına
 * bağlantı kurar.
 *
 */
public class JDBCDAOImpl implements DAO {
```

```
    /**
     * Bu metot içinde JDBC ile arama işlemi yapılır.
     */
    @Override
    public HotelResult search(final HotelSearch search)
            throws DAOException {
        Logger.instance(this).debug("search()");
        final HotelResult result = new HotelResult();
        Connection con = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;
        try {
            final ArrayList<Hotel> list = new ArrayList<Hotel>();

            con = getConnection();
            pstmt = con
                    .prepareStatement("select * from hotel "
                            + "where country=? and city=?");
            pstmt.setString(1, search.getCountry());
            pstmt.setString(2, search.getCity());
            rs = pstmt.executeQuery();

            while (rs.next()) {
                final Hotel hotel = new Hotel();
                hotel.setHotelName(rs.getString(1));
                hotel.setCountry(rs.getString(2));
                hotel.setCity(rs.getString(3));
                hotel.setStar(rs.getInt(4));
                hotel.setReservable(rs.getBoolean(5));
                list.add(hotel);
            }
            rs.close();
            pstmt.close();
        } catch (final Exception e) {
            throw new DAOException(e);
        } finally {
            try {
                con.close();
            } catch (final SQLException e) {
                e.printStackTrace();
            }
        }
        return result;
    }

    /**
     * Veri tabanı bağlantısı oluşturur.
```

```
     *
     * @return Connection Connection
     */
    private Connection getConnection() {
        // Veri tabanına bağlantı kurar ve
        // bir connection nesnesi oluşturur.
        // ornek olduğu için bu metot null
        // geri vermektedir.
        return null;
    }
}
```

Burada DataMapper tasarım şablonundan faydalanarak, Hotel nesnelerini oluşturabilirdik.

Program ilk yazılmaya başlandığı zaman belki veri tabanı henüz hazır olmayabilir. Bir veri tabanı olmamasına rağmen yazılıma devam edebilmek için DummyDAOImpl şeklinde bir implementasyon düşünülebilir:

```
// Kod 17

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            persistence.dao;


/**
 * DAO interface sınıfını implemente eder.
 *
 */
public class DummyDAOImpl implements DAO {

    /**
     * Bu metot içinde cok basit bir arama operasyonu simule edilir. Gerçek
     * implementasyon sınıflarında veri tabanÄ± ile bağlantı kurulur ve gerekli
     * veriler okunur.
     */
    @Override
    public HotelResult search(final HotelSearch search)
            throws DAOException {
        Logger.instance(this).debug("search()");
        final HotelResult result = new HotelResult();
        try {
            final ArrayList<Hotel> list = new ArrayList<Hotel>();

            // birinci dummy otel
            final Hotel hotel = new Hotel();
            hotel.setHotelName("Saray Oteli");
```

```
            hotel.setCountry(search.getCountry());
            hotel.setCity(search.getCity());
            hotel.setStar(5);
            hotel.setReservable(true);
            result.add(hotel);

            // ikinci dummy otel
            final Hotel hotel2 = new Hotel();
            hotel2.setHotelName("Belek Otel");
            hotel2.setCountry(search.getCountry());
            hotel2.setCity(search.getCity());
            hotel2.setStar(4);
            hotel2.setReservable(true);
            result.add(hotel2);
        } catch (final Exception e) {
            throw new DAOException(e);
        }
        return result;
    }
}
```

DummyDAOImpl sınıfında görüldüğü gibi bir veri tabanı mevcut olmasa bile, gösterim ve işletim katmanı görevlerini yerine getirebilecek şekilde veri katmanı ile beraber çalışabilmektedirler. DummyDAOImpl gerekli HotelResult nesnesini kendisi oluşturarak, içine Saray ve Belek otellerinin yer aldığı Hotel nesnelerini yerleştirmektedir. Böyle bir implementasyon ile veri tabanı olmadan bile yazılım süreci devam ettirilebilir.

Son olarak DAOFactory sınıfını inceliyelim. Değişik tipte DAO implementasyonları olabileceği için, kullanılacak implementasyon nesneni oluşturma işlemini DAOFactory sınıfı üzerinden yapmak istiyoruz. Böylece işletim katmanı, DAO implementasyon nesnesinin nasıl oluşturulduğunu bilmek zorunda kalmadan, DAOFactory sınıfını kullanarak, istediği işlemleri gerçekleştirebilecektir.

```
// Kod 18

package com.pratikprogramci.designpatterns.bolum9.otelrezervasyon.
            persistence.dao;

/**
 * DAOFactory sınıfı ile persistence.properties dosyasinda tanımlanmış olan DAO
 * implementasyon sınıfı kullanılır. Yeni DAO implementasyon sınıfları
 * oluşturulabilir ve persistence.properties dosyasina eklenerek, sistemin diğer
 * bölümlerini etkilemeden yeni implementasyon sınıfı kullanılır. Tasarım
 * şablonlarının kullanılmasının asıl amacı budur: sistemi oluşturan metotların
 * bağımsız olarak design edilmesi ve yapılan değişikliklerin sistemin diğer
```

```
 * bölümlerini etkilememesi.
 *
 * @author Ozcan Acar
 *
 */
public class DAOFactory {
    /*
     * static factory değişkeni
     */
    public static DAOFactory factory;

    /*
     * Kullanılacak dao sınıfı (Burada sadece interface sınıfı kullanılmış.
     * İstediğimiz bir dao implementasyon nesnesini bu değişkene
     * esitliyebiliriz)
     */
    private DAO dao = null;

    /**
     * Singleton tasarım şablonunu kullanarak, factory sınıfından sadece bir
     * nesne oluşturup, kullanıyoruz. Singleton olabilmesi için konstuktorlerin
     * private olması gerekiyor. Böylece başka bir sınıf new DAOFactory()
     * kullanarak, bu sınıftan bir nesne oluşturmaz.
     *
     */
    private DAOFactory() {
        initFactory();
    }

    /**
     * Bu sınıftan oluşturulan tek nesneye ulaşmak için kullanılır
     *
     * @return DAOFactory factory
     */
    public static DAOFactory instance() {
        if (factory == null) {
            factory = new DAOFactory();
        }
        return factory;
    }

    /**
     * Kullanmak istediğimiz DAO implementasyon sınıfını persistence.properties
     * içinde tanımlıyoruz. Bu metot içinde, persistence.properties içinde
     * tanımlanmış olan dao anahtarı kullanılarak, istenilen DAO implementasyon
     * sınıfı oluşturulur.
     *
```

```
     */
    private void initFactory() {
        try {
            final String key = ResourceBundle
                    .getBundle(
                            "otelrezervasyon/persistence")
                    .getString("dao");
            dao = (DAO) Class.forName(key).newInstance();
        } catch (final Exception e) {
            e.printStackTrace();
            throw new RuntimeException();
        }
    }

    /**
     * dao değişkeni için get metodu.
     *
     * @return dao
     */
    private DAO getDao() {
        return dao;
    }

    /**
     * DAOFactory sınıfı dis dunyaya seach metodunu sunar. Kendi içinde, işlemi
     * DAO sınıfının search metoduna delege eder. Bu sayede ilerde DAO interface
     * sınıfı uzerinde değişiklik yapılsa ve search() metodu search2() olarak
     * değiştirilse bile, gosterim katmanı DAOFactory sınıfını kullandığı için
     * bu değişiklikten etkilenmez. Sadece DAOFactory.search() metodunda
     * değişiklik yapılması yeterli olacaktır.
     *
     * @param hotel
     * @return HotelResult
     * @throws DAOException
     */
    public HotelResult search(final HotelSearch hotel)
            throws DAOException {
        return getDao().search(hotel);
    }

}
```

Kullanmak istediğimiz DAO implementasyonunu properties/otelrezervasyon/persistence.properties dosyasına ekliyoruz:

```
dao = com.pratikprogramci.designpatterns.otelrezervasyon.
```

```
                persistence.dao.DummyDAOImpl
```

DAOFactory sınıfı persistence.properties içinde bulunan dao anahtarı üzerinden kullanmak istediğimiz DAO implementasyon sınıfını tespit ederek, gerekli implementasyon nesnesini oluşturur. Property dosyalarının kullanılması, programların bakımı ve dinamik olarak genişletilebilmeleri adına büyük kolaylık sağlamaktadır.

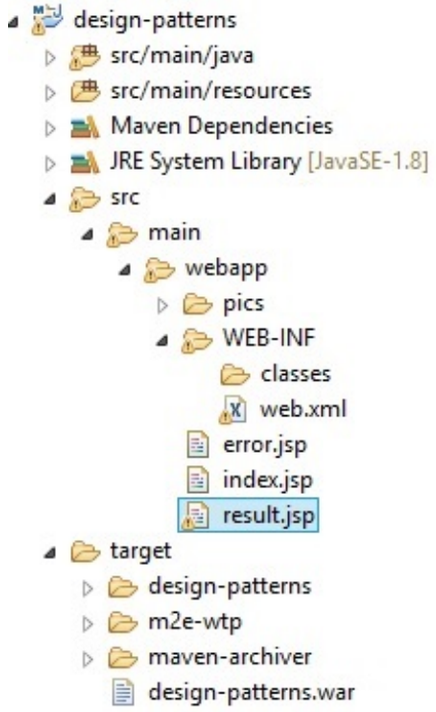
Bu Uygulamayı Nasıl Çalışır Hale Getirebilirsiniz

Uygulamayı bir Maven projesi olarak hazırladım. Uygulama aşağıdaki komut ile derlenmektedir:

```
c:/> mvn clean install
```

Uygulamanın derlenebilmesi için internet erişiminizin olması gerekiyor. Sadece bu durumda Maven projenin pom.xml dosyasında yer alan bağımlılıkları (dependency) merkezi Maven deposundan indirebilir.

Derleme işlemi tamamlandıktan sonra target dizininde design-patterns.war isminde bir dosya bulacaksınız. Bu dosyası Apache Tomcat 8 gibi bir uygulama sunucusunda çalışır hale getirebilir ve uygulamanın çalışma tarzını inceleyebilirsiniz.



Resim 15

Daha verimli sonuç alabilmek için Eclipse gibi bir çalışma ortamını kullanmanızı tavsiye ediyorum.

Son Söz

Bir yıllık bir çalışmanın ardından elinizde tuttuğunuz bu kitap oluştu. Çalışma hayatıma paralel olarak hazırladığım bu kitapta, edindiğim bilgi ve tecrübeleri sizinle paylaşmak istedim. Umarım yer yer çok karmaşık olabilen bu konuyu sade ve anlaşılır bir dille size aktarabilmişimdir.

Tasarım şablonları yazılım sektöründe kanımca büyük bir rol oynamaktadır. Profesyonelce yazılım yapılabilmesi için tasarım şablonlarının iyi uygulanması gerekmektedir. Alfabenin 29 harfini nasıl ezbere biliyor ve kelime oluşturmak için düşünmeden bu harfleri kullanabiliyorsak, bu kitapta adı geçen tasarım şablonlarını programcı olarak çok iyi bilerek, yazdığımız programlarda kullanabilmeliyiz. Bu tecrübe gerektiren bir süreçtir. Tekrar tekrar, zaman buldukça bu kitabı ya da başka kaynakları kullanarak, tasarım şablonları hakkında kendinizi geliştirmeniz, kısa bir zaman sonra ezberinizdeki harfler misali onların kullanımını kolaylaştıracaktır.

Tasarım şablonları her yerde kullanılmak zorundadır diye bir kural yoktur! Tasarım şablonları ile yeni tanışan bir programcı, mümkün mertebe bildiği tüm şablonları programın her yerinde uygulama eğilimi gösterir. Programcı tasarım şablonlarını gereğinden fazla kullanarak, hayatını güçleştirmemelidir. Amaç her zaman bakımı ve geliştirilmesi kolay program yazmak olmalıdır. Bu açıdan bakıldığında tasarım şablonları, günlük işlerimizde kullanabileceğimiz araçlardan farksızdırlar. Onlar yeterinde su ve güneş ışığı alabilen çiçek gibidirler. Uygun bir ortamda doğru şekilde kullanıldıkları taktirde güzel bir çiçek gibi açarak, hayatımıza ve yaptıklarımıza anlam katarlar.

Zaman bulup, bu kitabı okuduğunuz için teşekkür ederim. Umarım sizde edindiğiniz bilgileri başkalarıyla paylaşma firsatı bulursunuz, çünkü bu kelimelerle ifade edilmesi zor bir hazdır...

Aydınlık yarınlar umuduyla.....

EOF (End Of Fun)
Özcan Acar