Bölüm 8

Nesnesel Tasarım

Bu Bölümdekiler

- Nesne odaklı tasarım ve programcılık
- Tasarım düzenleri
- Mimariler

Nesne odaklı tasarım ve programcılık günümüzde o kadar geniş kabul görmüş haldedir ki, bu kavramlar sahneye çıkmadan önce nasıl program yazdığımızı (yaşı uygun olanlar) bile bazen hatırlayamaz oluyoruz. Fakat aslında kurumsal programcılığın en popüler dili Java'yı, onun temel aldığı C++, ve Smalltalk dillerini 1967 yılında çıkmış tek bir dile bağlamak mümkündür: Simula.

Simula yaratıcıları (Kristen Nygaard, Ole-Johan Dahl) dünyanın ilk nesnesel dilini simulasyon programları yazmak amacı ile yarattılar. Ayrısal olay simulasyonu (discrete event simulation), bir matematiksel sistemi cebirsel yöntemlerle (analitik modelleme -analytical modeling-) ile değil, sistemdeki aktörleri bir nesne olarak betimleyip, onların üzerine gelen olayları (event) gerçek dünya şartlarında olacağı gibi ama sanal bir ortamda tekrar yaratarak bir sistemi çözmeye verilen addır. Bazı problemlerin çözülebilir (tractable) bir analitik modeli kurulumadığından, ayrıksal olay simulasyonu popüler bir çözüm olarak kullanım bulmuştur.

Nesne odaklı tasarım ve programlama kavramlarının ilk kez simulasyon amaçlı yaratılmış bir dil içinde ortaya çıkmaları kesinlikle bir kaza değildir [3, sf. 1122]. Nesne kavramı, gerçek dünyadaki bir nesneyi çok rahat yansıttığı için, bu yönde kullanım bulması rahattı. Daha sonra Simula kullanıcılarının da fark ettiği üzere, simulasyon dışında Simula dilinin genel programcılık için yararlı olabileceği anlaşılmaya başlanmıştır. Bundan sonra nesne odaklı programlama kavramları genel programcı kitlesine de yayılmaya başladı.

Simula'nın özellikleri diğer diller tarafından hızla adapte edilmeye başlandı: Xerox PARC şirketinde Alan Kay, Simula'dan esinlendiği nesnesel odaklı kavramları, grafik ortamda programlama ortamı sağlayan yeni projesine dahil ederek Smalltalk dilini yarattı (1972). Eğer Simula akademik çevreleri etkilediyse, Smalltalk Byte dergisinin ünlü 1981 Ağustos sayısında *kitleleri* etkilemiştir: Bu sayıda görsel bir Smalltalk programlama ortamı ilk kez genel programcı seyircisine nesneleri görsel bir şekilde tanıtıyordu. Hakikaten de Smalltalk nesnesel diller içinde görsel geliştirme ortamını ilk destekleyen dil olarak günümüzdeki modern IDE'lerin çoğunun fikir babası sayılmaktadır.

Smalltalk'ın getirdiği ilginç bir özellik class (nesne tanımı) ile nesne kavramı arasındaki farkı ortadan kaldırmasıydı; Smalltalk dünyasında herşey bir nesneydi. Bu durum, günümüzde kullanılan "nesne odaklı programlama" teriminin bile Smalltalk'tan ne kadar etkilendiğini göstermektedir çünkü bu birleşim, ne yazık ki, her diğer "nesne odaklı" dil için doğru değildir: Java, Eiffel, C++ gibi güçlü tipleme takip eden dillerde class ve nesne çok net bir şekilde birbirinden ayrılır, yâni "nesne odaklı programlama" terimi, güçlü tipleme kullanan bir dil için "class odaklı programlama" olarak değiştirilmelidir (neyse!). Fakat Smalltalk diğer noktalarda, meselâ her şeyi nesne hâline getirmiş olması sayesinde debugger, nesne gezici (object browser) gibi koda erişmesi gereken ek araçların işini rahatlatabilmeyi başarmıştır.

Smalltalk'un dezavantajı, zayıf tipleme kullanan dinamik bir dil olmasıdır ve bu sebeple Smalltalk kullanan programlar ciddi performans problemleri

yaşamıştır. Sebebini şöyle açıklayabiliriz: Statik tipleme kullanan Java, Eiffel ve C++ derleyicilerinin bazı performans iyileştirmelerini programcılarına derleme anında sunmaları mümkündür: Meselâ, bu dillerin derleyicileri miras (inheritance) durumunda fonksiyonları bir dizin olarak önceden tutarak, dinamik bağlamayı (dynamic binding) anlık/sabit zamanda çözebilirler. Smalltalk, dinamik bir dil olması sebebiyle bu tür iyileştirmelerden faydalanamamıştır [3, sf. 1134].

Bu hatalar, AT&T Bell Labarotuvarlarında çalışan Bjarne Stroustrup tarafından dikkate alınarak, tekrarlanmamıştır. Simula'nın ana kavramlarını C diline taşıyarak C++ dilini oluşturan Stroustrup, özellikle C'den nesnesel kavramına geçiş yapmak isteyen programcılara 80 sonları ve 90 başlarında çok ideal bir seçenek sunmayı başardı. Bu zamanlarda C++, Kurumsal Yazılım Müdürleri (IT Manager) için her iki dünyanın en iyi birleşimiydi: Mevcut programcıları korkutmayacak kadar C, ama ileri kavramları öğrenmek isteyenlerin seveceği kadar "nesnesel". Fakat C++'ın nesnesel dil eklerinin gereğinden fazla çetrefil olması ve C++'ın bir çöp toplayıcıyı desteklememesi gibi sebepler yüzünden, 90'lı yılların ortalarında sektör kurumsal uygulamalarda Java diline doğru kaymıştır.

Java, C++'ın sözdizimini daha temizleştirerek çöp toplayıcı eklemiş, ayrıca eşzamanlı (concurrent) ve network üzerinde programlamaya paketten çıktığı hâliyle destek vermesi ile, nokta com patlaması (dot com boom) ile aynı anda anılır ve bilinir bir duruma gelmiştir. Bu yüzden "daha iyi bir C++" arayan kurumsal programcılar, ve yeni Internet ortamının gerektirdiği programlama için Java, gerekeni tam karşılayan bir dil hâline geliyordu. Günümüzde Java kurumsal programcılığın en yaygın kullanılan dili hâline gelmiştir.

8.1 Nesnesel Tasarım ve Teknoloji

Kurumsal programcılıkta kullanılan dilin temizliğine ek olarak, kullanılan *yan teknolojiler* ve onların gerektirdiği arayüzler (API) nesnesel tasarımımız üzerinde kesinlikle çok etkilidirler. Nesnesel tasarımın ve yöntemlerinin ilk yaygınlaşmaya başladığı 80'li ve 90'lı yıllarda yapılan önemli bir hata, dış teknolojilerin yok sayılması ve nesnesel tasarımın bir boşlukta (vacuum) içerisindeymiş gibi yapılmasının cesaretlendirilmeydi. Bu tavsiyenin projeler üzerinde etkisi öldürücü olmuştur, çünkü bir projede kullandığınız her teknoloji, *nesne yapılarınızı etkiler*.

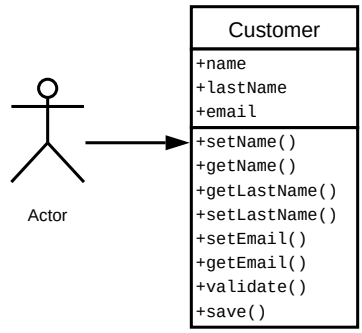
Diğer programcılık dallarında bu her zaman geçerli olmayabilir. Meselâ bilimsel formül hesabı (number crunching) yapması gereken türden programlar, dış sistemlerle fazla etkileşime girmezler. Girdi olarak bir dosya alırlar, ve çıktı olarak bir diğer dosya üretirler (ya da çizim (plot) basabilirler). Kıyasla bir kurumsal sistem, en azından bir veri tabanı ürünüyle, ek olarak ta uygulama servisi, JMS ile asenkron iletişim, e-mail, yardımcı kütüphaneler gibi birçok dış araç ile iletişim hâlinde olacaktır. Formül hesabı yapan programlarda dış sistemleri yok sayarak daha rahat nesnesel tasarım yapabilirsiniz, ama kurumsal

sistemlerde nesnesel tasarımı *nerede* kullanacağımızı çok iyi bilmemiz gerekecektir.

Nesnesel tasarım yapacağımız yer, programınızın işleyiş kontrolünün altyapı kodlarından sizin yazacağınız kodlara geldiği yerde başlar, ve tekrar dış teknolojiye geçtiğinde biter. Burada vurgulamak istediğimiz bir *öncelik mentalitesi* farkıdır: Yapılması yanlış olan, teknolojiden habersiz bir şekilde şirin ve güzel nesneler tasarlayıp onu sonra teknolojiye bağlamaktır. Bunun yerine yapılması gereken, önce teknolojiyle nasıl konuşmamız gerektiğini anlamamız ve sonra iletişim noktalarını iyice kavradıktan *sonra*, arada kalan boş bölgeleri nesnesel tasarım ile doldurmamızdır. Bunu bir örnek ile anlatmaya çalışalım:

Sisteme müşteri eklemesi gereken bir uygulama düşünelim. Bu müşteri veri yapısı hakkında tutmamız gereken özellikler (attributes) biliniyor olsun. Öğeler alındıktan sonra uygulamanın bazı doğruluk kontrolleri yapması gerekiyor, bunlar isim ve soyadının mecburi olması ve e-mail içinde `@` işaretinin bulunmasının kontrol edilmesi gibi işlemler olacaktır. Daha sonra, başka bir işlem ile müşteri veri tabanına yazılacaktır.

Bu gereklilik listesi üzerinden, bazı nesnesel programcılar tasarıma direk başlanabileceğini savunurlar. İdeal bir modeli de şöyle kurabilirler (Şekil 8.1).



Şekil 8.1: Yanlış Müşteri Nesnesi

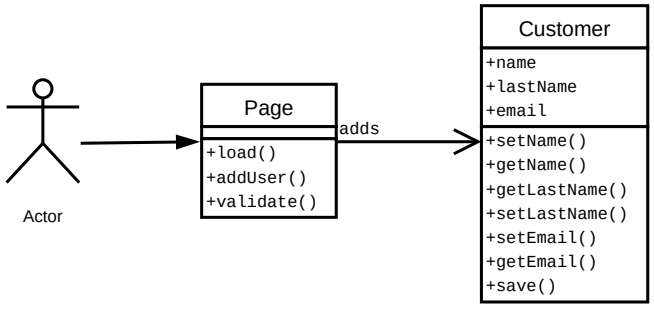
Bu modele göre önyüz, bir `Customer` (müşteri) nesnesini yaratacak, üzerinde `setName`, `setLastName`, vs. ile gereken verileri koyacak, daha sonra veri doğruluk kontrolü için `validate` çağrısını yapacak, ve en sonunda `save` ile müşteri nesnesini kaydedecektir.

Fakat, bu teorik model hiçbir *teknolojiyi* dikkate almamıştır, bu sebeple tasarımı etkileyebilecek bazı ek faktörler atlanmıştır. Bakalım teknolojiyi ekleyince modelimiz etkilenecek mi? Meselâ eğer önyüzde Struts altyapısını kul-

lanıyorsak, bağlanan (client) tarafında doğruluk kontrolü yapmanın bir Struts tekniği vardır. Özellikle örnekte gösterilen müşteri nesnesinde yaptığımız türden kontroller (isim, soyadı, e-mail) için, Javascript bazlı çalışan bir altyapı mevcuttur. Bu doğruluk kontrol altyapısı, Struts'ın ayar dosyasından `struts-¬ config.xml` ayarlanır, ve modelde görülen türden bir Java kodu yazılmasını gerektirmez. Bu yüzden, modelimize koyduğumuz `validate` metodu tamamen gereksizdir.

Aynı şekilde, eğer projemizde Hibernate kullanıyorsak, kalıcılık metodu olan `save`, müşteri nesnesi üzerinde değil, `org.hibernate.Session` nesnesi üzerindedir. Hibernate `save`'e parametre olarak müşteri nesnesini geçmek gerekir, `save` metodu müşteri üzerinde çağırılmaz. Yine teknolojiyi dikkate alınmadan modelin eksik olacağını kanıtlamış oluyoruz.

Fakat bitmedi. Şimdi önyüz ile `Customer` nesnesi arasında olabilecek çağrıları ele alalım. Meselâ, önyüz teknolojisi Swing olsun, ve bu sebeple birçok Swing önyüzünün ağ (network) üzerinden müşteriye erişmesi gereksin. Pür nesnesel modelleme yaptığımız için ve teknolojiyi dikkate (!) almadığımız için de, 8.2 üzerindeki yapıyı kuruyoruz.

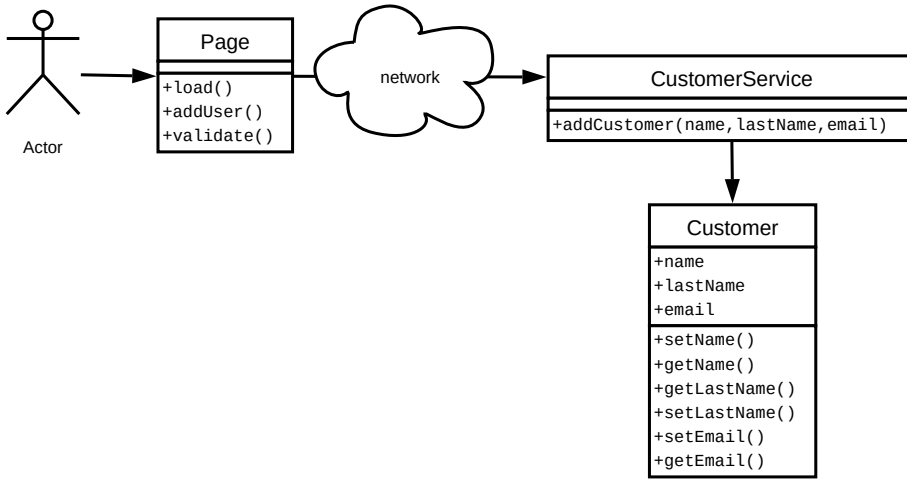


Şekil 8.2: Network Üzerinden Müşteri Eklemek

Bu model optimal çalışır mı? Eğer bir uygulamanın performansı en az doğruluğu kadar önemliyse, bu modelin hızlı çalısması önemli olmalıdır. Ne yazık ki bu soruya cevap, "hayır" olacaktır. Dikkat edersek önyüz, yâni `Page` (sayfa) nesnesi, `new` ile bir `Customer` nesnesi yarattıktan sonra, bu nesne üzerinde `set` metotlarını çağıracaktır. İşte problem burada ortaya çıkacaktır, çünkü network üzerinde erişim kurallarına göre, ufak ufak `set` çağrıları yapmak yerine, tüm gereken parametreleri birarada paketleyerek *tek bir defada* hepsini göndermek daha hızlıdır. Kurumsal programcılıkta ufak ufak ve çok çağrı yapan sistemlere geveze (chattery) sistemler denmektedir, ve bu tür mimarilerden kaçınılır.

O zaman, modelimize arayüzü daha geniş olan yeni bir class daha eklememiz

gerekiyor. Bu class, dış dünyaya gösterilen arayüz olacaktır, çünkü parametre listesini network üzerinden göndermeye daha elverişli durumdadır. Bu yeni model, Şekil 8.3 üzerinde görülebilir.



Şekil 8.3: Doğru Müşteri Modeli

Bu şekil üzerinde yeni bir class, `CustomerService` görüyoruz. Bu class hangi dağıtık nesne teknolojisini kullanıyorsak, o şekilde uzaktan erişime açılacak olan class'tır.

Özet olarak, ilk yola çıktığımız modelden oldukça uzaklaştık. Metot `validate`, hem Struts hem de Swing ortamında "bağlanan" tarafa alınması gereken bir metottu, bu yüzden `Customer`'dan çıkartıldı. Metot `save`, kalıcılık aracı üzerinde olacaktı, bu sebeple çıkartıldı. Ve son olarak uzaktan erişim için ilk modelde hesaba alınmayan `CustomerService` class'ı eklendi, böylece `Customer` nesnesinin uzaktan çağırılmasını engelledik.

8.2 Modelleme

Teknolojinin kontrolü bize bıraktığı ve bizim kontrolü geri verdiğimiz noktanın arasında (ve hâla teknolojiyi aklımızda tutarak) artık klasik anlamda modelleme yapabiliriz. Tabii en teorik, ve teknolojiden bağımsız olduğunu iddia eden modeller bile tek bir teknolojik önkabul yapmaktadırlar; Bir nesnenin

diğer bir nesneye yaptığı metot çağrısının milisaniye seviyesinde ve aynı JVM içinde işletileceği önkabulu. Bölümümüzün geri kalanında bu teknolojik önkabule dayanarak istediğimiz büyüklükte ve sayıda class'ı modelimize koymakta, ve herhangi bir metotu bir class'tan ötekine aktarmakta sakınca görmeyeceğiz.

8.2.1 Prensipler

Modellerimiz için takip etmemiz gereken basit prensipler şunlar olacaktır:

- Fonksiyonların yan etkisi olmamalıdır.
- Kurumsal uygulamalarda, model için class seçerken ilişkisel modelde (veri tabanı şemasında) tablo olarak planlanmış birimlerin class hâline gelmesinde bir sakınca yoktur. Bu ilişki her zaman birebir olmayabilir, ama şemamızdaki tabloların büyük bir çoğunluğunu modelimizde class olarak görülecektir.
- Bir class'ın metotları ve fonksiyonlarını bir "alışveriş listesinin kalemleri" gibi görmeliyiz. Bir class dış dünyaya birden fazla servis sunabilir, ve hâtta sunmalıdır. Bir class sadece tek bir iş için yazılma**ma**lıdır.
- Uygun olduğu yerde, tasarım düzenleri (design patterns) faydalı araçlardır. Erich Gamma ve arkadaşlarının paylaştığı düzenler içinden
  - Command
  - Template Method
  - Facade
  - Singleton

  kurumsal sistemlerde kullanılan düzenlerdir (bunların haricindeki Gamma düzenleri kullanım görmez).
- Hibernate ile eşlediğimiz POJO'lara `set` ve `get` haricinde metotlar eklemekte sakınca yoktur.

Şimdi bu prensipleri teker teker açıklayalım:

8.2.2 Fonksiyonların Yan Etkisi

Bir fonksiyon, geriye bir sonuç döndüren bir çağrıdır. Bu geri döndürülen değer, ya o çağrı içinde anlık yapılan bir hesabın sonucu, ya da nesnede tutulmakta olan bir nesneye ait referans değeri olacaktır. Her ne olursa olsun, bir fonksiyon bir değer döndürmekle yükümlüdür. Kıyasla metotlar geriye hiçbir değer döndürmezler. Bu yüzden metot dönüş tipi olarak `void` kullanmak gerekir.

Bir fonksiyonun görevi, raporlamaktır. Metotlar nesne içinde *bir şeyler değiştirmek* için kullanılır. Eğer bir örnek vermek gerekirse bir nesneyi, bir radyo gibi düşünebiliriz. Bu radyoda metotlar, kanal değiştirme, ses açma/kapama

gibi düğmelerdir. Fonksiyon ise, radyonun hangi kanalda olduğunu gösteren bir ibre olabilir.

Bu yüzden, fonksiyonlarımızı yazarken nesne içinde değişiklik yapmamaları için çok özen göstermeliyiz. Bu prensibe dikkat etmeyen ve çağırılınca nesnede değişikliğe sebep olan fonksiyonlara *yan etkisi olan* fonksiyonlar denmektedir, çünkü fonksiyonun ana amacı dışında bir yan etkiye sebebiyet vermiştir; Nesnenin iç verisinde değişikliğe sebep olunmuştur. Radyo örneğimize dönersek, düşünün ki hangi kanalda olduğumuza bakınca radyonun kanalı değişiyor! Böyle bir radyo muhakkak pek kullanışlı olmazdı, çünkü bir nesne hakkında bilgi almak istediğimizde, o bilgi alma işleminin, nesne üzerinde bir değişiklik yapmamasını bekleriz. Nesnesel odaklı tasarımımızda bir fonksiyon yazarken, aynı mantık ile düşünmeliyiz. Fonksiyonlarımız değişikliğe sebep olmuyorsa, içimiz rahat bir şekilde bu fonksiyonları istediğimiz kadar çağırabiliriz.

8.2.3 Veri Tabanı ve Nesnesel Model

Model için class seçmek, çoğu nesne odaklı tasarım literatüründe işlenen bir konudur, ve birçok kulağa küpe (rule of thumb) kural ortaya atılmıştır. Mesela dilbilgilsel bir yöntem söyledir: Gereklilikler (requirements) dokümanının cümlelerindeki isimler (noun) seçilir ve bunlardan bazıları class olarak seçilir. Bu yönteme göre, bir gereklilik dökümanındaki şu cümlede;

> "Asansör hareket etmeden önce kapılarını kapatmalıdır, ve asansör bir kattan öteki kata her hareket edişinde veri tabanında bir kayıt yazılmalıdır"

asansör, *kat* ve *kapı* birer class adayı olarak seçilmelidirler.

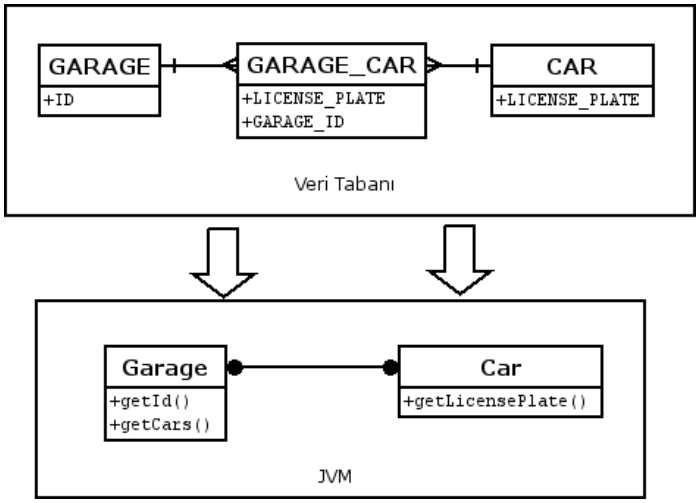
Fakat böyle basitçi bir yöntem ne yazık ki bizi fazla uzağa götürmez. İnsan dili değişik nüanslara çok açıktır ve çok daha katı kurallara göre yazılması gereken bir bilgisayar sistemi için bizi yanlış yöne sevkedebilir. Mesela üstteki örnekte aslında class olması gereken şey, *hareket*'tir. Ama bu kelime isim değil bir fiil olduğu için, yöntemimiz tarafından seçilmemiştir [3, sf. 723].

Kurumsal uygulamalarda dilbilgisel yöntem yerine veri tabanındaki tablolara bakarsak, kullanmamız gereken class'lar hakkında daha iyi bir fikir edinebiliriz. Meselâ üstteki örnekteki hareket, ilişkisel modelleme sırasında kesinlikle

Şekil 8.4: Java ArrayList Bir Makina Olsaydı

yakalanacak bir tablo olduğundan, bu tabloya bakarak modelimize aynı isimde bir class ekleyebiliriz.

Tabii şemamızdaki her tabloyu bir class hâline getirmemiz gerekmez. Meselâ daha önce kullandığımız `Garage` ve `Car` örneğine gelirsek, bu iki tablo arasında çoka çok eşleme yaptığımız durumda bir ara tablo (cross reference table) kullanmamız gerekecektir. Bu tablonun nesne modeline yansıtılması gerekmez. Şekil 8.5 üzerinde bu eşlemeyi görüyoruz.



Şekil 8.5: Şemadaki Her Tablonun Nesne Modeline Yansıtılması Gerekmez

Bu eşlemede `GARAGE_CAR` adlı tablo, nesne dünyasına yansıtılmamıştır (bu eşlemenin Hibernate üzerinden nasıl yapıldığını anlamak için 2.5.3 bölümüne bakabilirsiniz.

8.2.4 Metotlar ve Alışveriş Listesi

Class seçerken eşzamanlı olarak aklımızın bir köşesinde bir seçim daha yaparız; Hangi metotlar ve hangi fonksiyonların bir class üzerinde olacağı seçimi. Zihnimizde belli servisler ile veriler biraraya geldiğinde, bu birliktelikten bir class doğar. Peki bu birlikteliği bulmak için bir yöntem var mıdır? Bu iş için nasıl bir zihin durumu (mindframe) içinde bulunmalıyız?

Tek Amaçlı Class

Nesnesel tasarımda *tek bir* servis sağlayan class'lardan kaçınmamız gerekir, çünkü bu tür class'lar, hâla class modülünü bir servisler listesi olarak görmediğimizi gösterir. Bu tür nesneleri, onları belgeleyen dokümanlardan rahatça anlayabilirsiniz, genellikle şöyle târif edilirler: "Bu nesne xxx işini yapıyor". Bir xyz

işini "yapıyor" diyebildiğimiz nesne, demek ki tek o iş için yazılmıştır. Halbuki nesnesel yöntemin gücü, bir veri tanımı etrafında birçok servisin sağlanabilmesidir. Bu servislerin illâ ki birbirlerini çağırıyor ve birbirleri ile yakın alâkada olacak hâlde tasarlanmış olması gerekmez. Bu yüzden bir class'ın metot listesi, "bir alışveriş listesine benzer" denir. Değişik ihtiyaçlar için değişik kalemler vardır[1].

Meselâ `java.util.ArrayList` class'ını örnek alırsak, dış dünyaya sağladığı birçok servis vardır. `add` çağrısı ile nesneye (listeye) bir eleman eklenmesini sağlar, `remove` ile bir nesnenin silinmesi servisini sağlar. Çağrılar `add` ile `remove` arasında direk bir kod bağlantısı yoktur. Bu iki metot birbirlerini çağırmazlar. Ama aynı class üzerinde bulunurlar. Eğer bu iş için sadece "tek bir işlem için" yazılmış bir class kullansaydık, o zaman `ListAdder` ve `ListRemover` gibi iki class yazmamız gerekecekti. Bu class'lardaki metotları ise `doIt` gibi komik bir isimde olabilecek bir tek metot olacaktı. Tavsiyemiz, `doIt` metotlarından ve bu metotları içerecek class'lardan kaçınmanızdır.

Değişik servisler sağlayan bir nesne düşünmek tasarımcı için faydalıdır. Aynı şekilde, bir sistem içinde aynı class'tan gelen birçok nesnelenin aynı anda, farklı roller oynayabileceğini (yâni iki nesnenin değişik metotlarının çağırılabileceğini) düşünmek te tasarımcı için faydalıdır.

Nesnesel Modellerde Tepe Fonksiyonu Yoktur!

Prosedürel disiplinden gelen ve kurtulmamız gereken ikinci bir alışkanlık, yazmakta olduğumuz uygulama için sürekli bir "üst nokta" aramaktır. Üst noktadan kastımız, uygulamada her şeyin başladığı ve kontrol edildiği o tek işletici, çağırıcı, ana metot, başlangıç noktasıdır. Prosedürel günlerden kalma bu alışkanlığı, nesnesel sistemler kurarken terketmemiz gerekiyor, çünkü eğer sürekli üst nokta metotunu düşünürsek, tasarımız çok fazla "o kullanım için" olacak, ve kod düzeni açısından tüm fonksiyonlarımız o başlangıç koca metotunun uydusu hâline gelerek, en kötü durumda başlangıç metotunun olduğu class'ta bir takım ek metotlar haline gelecektir.

Nesnesel sistemlerde üst nokta yoktur. Nesnesel tasarımlarda "uygulama" denen şey, tüm class'lar kodlandıktan sonra *en son aşama olarak*, son anda gereken class'ların birleştirilerek (çağırılarak) biraraya getirilmesi gereken bir yapboz resmidir. Burada belki de en önemli fark, hangi eylemin hangisinden önce geleceğini vurgulayan bir öncelik farkıdır. Prosedürel yöntemler tasarımın yukarıdan-aşağı (top-down) yapılmasını zorlarken, nesnesel yöntemde tasarım alttan yukarı (bottom-up) doğru gider.

[1] 4.1 bölümünde bahsedilen Command mimarisinin bu kurala uymadığı düşünülebilir, fakat Command mimarisindeki durum, teknolojik bir gereklilikten (network iletişim hızının aynı JVM'de olan iletişimden daha düşün olması sebebiyle) ortaya çıkmıştır. Dediğimiz gibi, teknolojik sınırlar mimariyi etkilemiştir.

Örnek

Bu düşünce farkını herhalde en iyi şekilde bir örnek üzerinden aktarabiliriz. Uygulamamızın amacı şu olsun: Bir sayıyı bir sayı düzeninden diğerine taşımak. Meselâ, 2'lik düzende 110010010 sayısını, 10'luk düzendeki karşılığına gitmemiz gerekiyor. Ya da tam tersi yönde gidebilmemiz lazım. Her düzenden her düzene geçebilmeliyiz.

Prosedürel (functional) bir geçmişten gelen programcı, hemen bu noktada "ne yapılması gerektiğine" odaklanır, ve o üst fonksiyonu düşünmeye başlardı. Onun bulması gerekenler, verileri alan, işleyen fonksiyonlar olacaktı, ve veriler bir işlemden diğerine aktarılırken yolda değişe değişe istenen sonuca ulaşılacaktı. Tasarım şöyle olabilir.

```
public static void main() {
   double fromNumber;
   double fromBase;
   double toBase;

   fromNumber = ...;
   fromBase = ...;
   toBase = ...;

   ...
}
```

İyi bir prosedürel programcı olarak hemen ana metotu koyduk. Uygulama için gereken girdileri burada zaten alıyoruz. O zaman bu girdileri ne yapacağımızı tasarlamamız gerekiyor. Hemen bir takım alt metotları kodlamaya başlıyoruz. Bir sayıyı bir düzenden diğerine çevirirken ara seviye olarak onluk düzene gitmemiz gerekiyor, çünkü onluk düzenden diğerlerine nasıl geçeceğimizi biliyoruz. O zaman ilk önce, onluk düzene geçen metotu çağırmamız ve kodlamamız gerekiyor.

```
public static void main() {
   String fromNumber;
   int fromBase;
   int toBase;

   fromNumber = ...;
   fromBase = ...;
   toBase = ...;

   int fromNumBase10 = convertToBaseTen(fromNumber);
}
public double convertToBaseTen(double fromNumber) {
   ...
}
```

Sonra bu ele geçen sayıyı, yeni düzene çevirecek fonksiyona göndereceğiz.

```
public static void main() {
   String fromNumber;
   int fromBase;
   int toBase;

   fromNumber = ...;
   fromBase = ...;
   toBase = ...;

   int fromNumBase10 = convertToBaseTen(fromNumber);
   String toNumber = convertToBase(fromNumBaseTen, toBase);
}
public int convertToBaseTen(double fromNumber) {
   ..
   return numBaseTen;
}
public String convertToBase(int fromNumBaseTen, int toBase){
   ...
   return toNum;
}
```

Bu yöntem oldukça kalabalık bir koda sebebiyet verdi. Özellikle ana metot daha programın başında neredeyse yapılabilecek her çağrıyı yapıyor, ve tüm gereken değerleri o hatırlıyor. Bundan daha iyi bir tasarım yapamaz mıydık?

Nesnesel tasarımı deneyelim. Nesnesel yöntemdeki prensipleri hatırlayalım. Yukarıdan aşağı değil, aşağıdan yukarı gidiyoruz. Bunu yaparken tasarladığımız metotları, bir class'ın alışveriş listesi gibi görüyoruz. Bir class, bir uygulama içinde birden fazla rol oynayabilir. O zaman uygulamada bu tanıma uyan class nedir?

Bir sayı! Evet, bize gereken *kendini* onluk düzene, ya da *kendini* onluk düzenden başka bir düzene çevirebilecek olan, ve hangi sayı düzeninde olduğunu *kendi bilen* bir sayı class'ıdır. Bir sayı class'ı, uygulama sırasında birçok değişik rol oynayabilir; Çevirilen rolü oynayan bir nesne, kendini onluk düzenden gösterebilecek, hedef rolünü oynayan nesne ise, hangi düzende olması gerektiğini bilen, ve çevirim için kendini önce onluk düzene, sonra gereken hedef düzene çevirmeyi bilecek bir nesne olacaktır. Bu açıdan bakılınca sayı class'ı üzerindeki metotlar bir alışveriş listesidir. Üst nokta düşünmeden bu metotları koyduk, yâni uygulamamız artık tepe noktadan kontrol edilen bir çağrı zinciri değil, *iki nesnenin rol aldığı bir simulasyon* hâline geldi. Kodu yazalım:

```
public class Number {

  String value;
  int base;

  public Number(String value, int base) {
  }
```

```
  public Number(int base) {
  }

  public int toBaseTen() {
    ..
    return numInTen;
  }

  public String getValue() {
    return value;
  }

  public void convert(Number from) {
    //
    // Çevirilecek sayı: from.toBaseTen()
    // Hedef: this.base
    //
  }
}
```

Gördüğümüz gibi kodlama açısından neredeyse aynı olan metotlar doğru class üzerinde gelince isimleri daha temiz hâle geldi. Hangi düzende olduğunu bilen bir nesneye `convert` çağrısı gelince ve parametre olarak bir diğer `Number` nesnesi verilince bunun anlamı çok nettir; Parametre olarak gelen `Number`'daki sayı düzeninden kendi içimizde tuttuğumuz sayı düzenine geçmek istiyoruz.

Ayrıca bu yeni tasarımda, çevirilecek ve hedef sayılar hakkındaki bilgileri ana metotta duran değişkenlerde tutmamız gerekmiyor. Her class, kendisi hakkında bilgileri kendi tutacaktır. Bunlar, sayı değeri ve hangi sayı düzeninde olunduğudur.

Bu yeni tasarımı kodladıktan sonra, *en son olarak* ana metotu kodlayabiliriz. Ana metotun ne kadar daha temiz olduğunu göreceğiz.

```
public class App {

  public static void main(..) {

   Number from = Number(..,..);
   Number to = Number(..);

   to.convert(from);

   System.out.println(to.getValue());

  }

}
```

Görüldüğü gibi çevirilecek sayı, hangi düzende olduğu bilgisiyle beraber, `from` referansıyla erişilen `Number` nesnesi içinde tutulmaktadır. Hedef sayı düzeni, `to` referansı ile erişilen `Number` içinde tutulmaktadır. Sayı değeri ve düzeni, beraber, aynı modül içinde durmaktadırlar, ve bu durum ana modülü rahatlatmış, ve genelde kod bakımı açısından bir ilerleme sağlamıştır.

Diğer bir ilerleme, `convert` adlı metota hedef sayının bir `Number` parametresi olarak gelmesidir. Eskiden `String` ve `int` tipinde değerler geliyordu, ve bu basit tiplere bakarak neyin ne olduğu tam anlaşılamıyordu. Yeni yöntem sayesinde daha üst seviyede olan `Number` tipleri metot'tan metota gönderilmektedir, ve bu da kodun anlaşılırlığı açısından iyidir.

8.3 Tasarım Düzenleri (Design Patterns)

Nesnesel dillerin programcıya sağladığı belli silahlar/yetenekler (capabilities) vardır. Bu yetenekler nesnesel yöntemin tabiatından gelirler, meselâ çokyüzlülük (polymorphism) ve miras alma (inheritance) yeteneklerinin standart örneği, üst seviye bir class'tan miras alan alt seviye gerçek (concrete) class'ların, üst seviye referansı üzerinden erişilmesi örneğidir. Bu kullanımda, meselâ, aynı `List` üzerinde aynı üst seviye tipinde ama gerçek tipi değişik alt tiplerde olan nesnelerin örneği gösterilebilir. Tüm bu alt tipte nesneler üzerinde, üst seviyedeki bir metot alta çokyüzlülük ile intikal etmiştir, ve bu metotun gerçekleştirimi her alt tipte değişik olması sebebiyle, üst seviye tip üzerinde yapılan çağrılar, değişik davranışta bulunurlar. Bu kullanım, ufak çapta bir tasarım düzenidir.

Bu tasarım düzenini kullanan bir uygulama örneği şöyle olabilir: Ekranda üçgen, çember ve poligon çizmeye izin veren bir çizim programını düşünün. Mouse ile tıkladığımız noktada, hangi figür mod'undaysak (`Circle`, `Triangle`, `Polygon`), o figür, o noktada, o nesnenin `draw` metotu üzerinden çizilecektir.

Buna ek olarak çizim programı, "ekranı yenile" komutuna hazır olmak için ekrandaki figürlerin bir listesini tutmalıdır. Böylece ekrana "tekrar çiz" (`refresh`) komutu verildiğinde, herşey silinip tüm figürler üzerinde tekrar `draw` metotu çağırılabilecektir. `List` nesnesini taşıyan `Screen` (ekran) nesnesi, o listeyi tekrar gezdiğinde, listedeki elemanları `Figure` tipine dönüştürmesi (cast) yeterlidir. Çünkü `Figure` üzerinde `draw` interface metotu tanımlıdır, ve çokyüzlülük kanunlarına göre `Figure` tipi üzerinden baktığımız bir nesnenin `draw` metotunu çağırmak için o nesnenin gerçek tipine inmemiz gerekmez. Nesnesel yöntem, `draw` çağrısını gerekli koda otomatik olarak götürecektir. Böylece alttaki kod parçası

```
for (int i = 0; i < figureList.size(); i++) {
  Figure figure = (Figure)figureList.get(i);
  figure.draw();
}
```

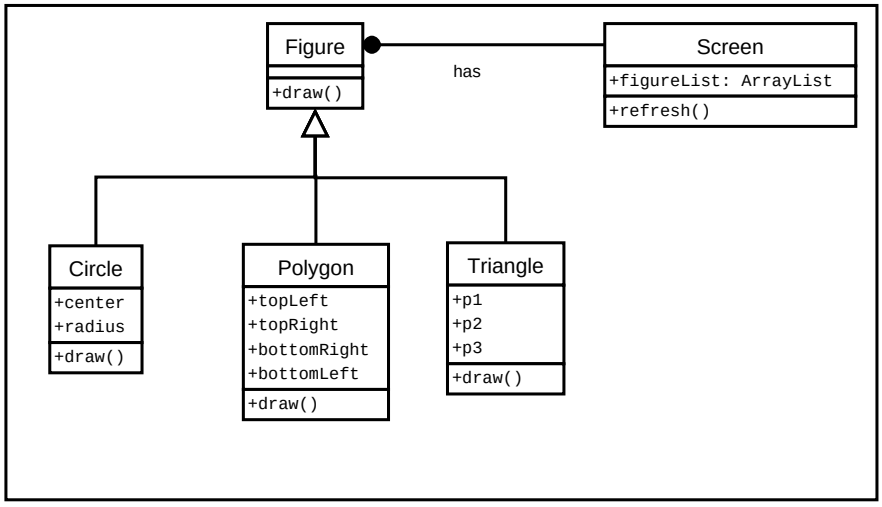
ile tüm figürleri tekrar çizmemiz mümkün olacaktır.

Demek ki tasarım düzenleri, baz seviyedeki nesnesel teknikleri kullanarak, değişik şekillerde birleştirerek oluşturulmuş üst seviye tekniklerdir. Nesnesel tekniklerin ve dillerin anlatıldığı her kitapta aslında tasarım düzenleri de bir yandan anlatılmaktadır, sadece en direk ve dil özelliğini en basitçe vurgulayacak olan çeşitleri ön planda olmaktadır.

8.3.1 Kullanılan Düzenler

Gamma ve arkadaşlarının çıkardığı Tasarım Düzenleri [2] adlı kitap, yazarlarının projelerinde üst üste kullandığı ve temel kullanımlardan daha değişik kodlama ve tasarım düzenlerini dünyaya tanıtmıştır. Bu kitaptaki teknikler bir yana, bir tasarım düzeninin nasıl bulunup ortaya konulacağını ortaya koyması açısından kitap daha da faydalı olmuştur. Kurumsal programcılar için yapmamız gereken tek uyarı, bu kitaptaki paylaşılan tasarım numaralarının pür dil seviyesinde olmasıdır (ve kurumsal uygulamalarda dış teknolojinin önemini artık biliyoruz). Tasarım Düzenleri kitabındaki çözümlerin neredeyse tamamı "aynı JVM, yerel metot çağrısı" öngörüsüyle yazılmış tekniklerdir. Kurumsal yazılımlarda işe yarayan Gamma TD teknikleri altta görülebilir:

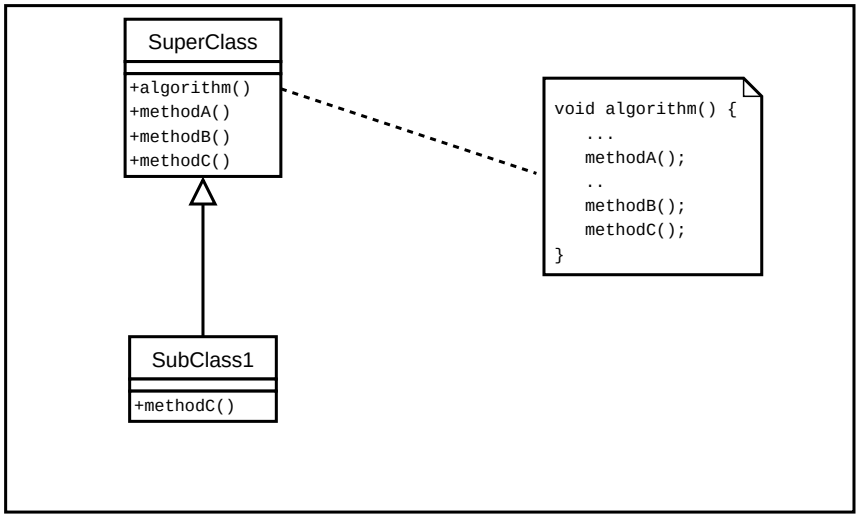
- Command
- Template Method
- Facade
- Singleton



Şekil 8.6: Şekiller Nesne Diyagramı

Bu düzenleri kısaca açıklayalım (Command düzeni haricinde, çünkü bu düzen 4.1 bölümünde ayrıntısıyla anlatılmıştır).

Template Method



Şekil 8.7: Template Method Nesne Tasarımı

Arasında miras ilişkisi olan üst ve alt class tiplerini kodlarken, ileride yeni eklenebilecek alt class tiplerine yarayacak olan metotların üst class'a çekilmesi gerekir. Bu metotlar, böylece her yeni alt class tarafından paylaşılabilmiş olacaktır.

Ortak metotları üst class'a koyduğumuzda, bazen, kodun içinde "genel olmayan" ve "her class için değişik olması gereken" bir bölüm gözümüze çarpabilir. Eğer böyle bir bölüm mevcut ise, tüm metotu tekrar aşağı, alt class'a doğru itmeden, Template Method düzenini kullanabiliriz. Bu düzene göre, her alt class'ta değişik olabilecek ufak kod parçası ayrı bir metot içine konarak, üst tipte soyut (`abstract`) olarak tanımlanır. Böylece alt class'lar bu metotu tanımlamaya mecbur kalırlar. Ve alt class'taki özel bölüm tanımlama/kodlaması yapılır yapılmaz üst class'taki metotlar genel bölümlerini işletip, özel bölüm için alt sınıftaki metota işleyişi devredebilirler. Bundan kendilerinin haberleri bile olmaz, çünkü onlar kendi seviyelerindeki `abstract` metotu çağırmaktadırlar.

Template Method, üst seviyede bir metot iskeleti tanımlayıp alt class'lara sadece ufak değişiklikler için izin verilmesi gerektiği durumlarda kullanışlıdır.

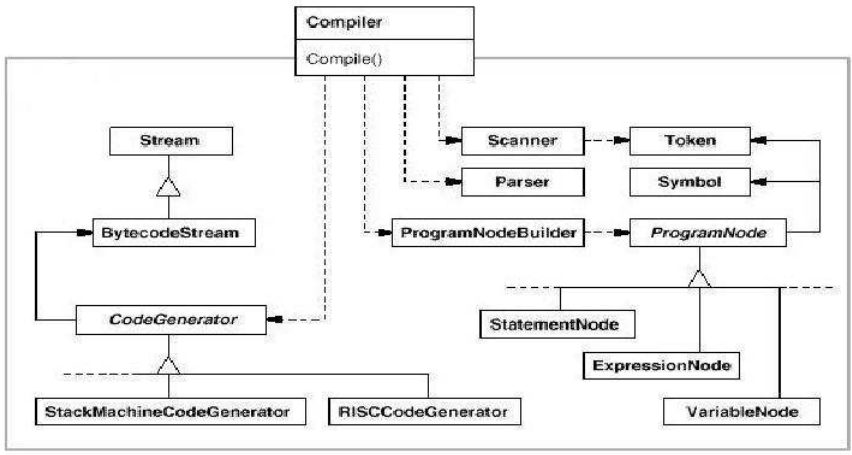
Facade

Bir özelliği işletmek için birçok nesneyi ardı ardına çağırmak gerekiyorsa, özellik kullanımını tek bir giriş class'ına alarak arka plan çağrıları giriş class'ına yaptırmak faydalı olabilir. Tasarım Düzenleri kitabında bu giriş class'ına Facade adı veriliyor. Facade'ın faydası, karmaşıklığı azaltarak bir sistemin dış dünyaya gösterdiği arayüzü basitleştirmeyi amaçlamasıdır. Meselâ Şekil 8.8 üzerinde gösterilen nesne modeli `javac` gibi bir derleyici (compiler) sistemin tasarım modelidir. Bu modelde görüldüğü gibi birçok iş yapan class'lar mevcuttur. Fakat dışarıdan bağlanan için bu alt seviye nesneleri yaratıp teker teker çağırmak şart değildir, dışarısı için tek giriş nesnesi olan `Compiler`'ı kullanmak hem kullanım, hem de kod bakımı açısından daha rahat olacaktır.

Singleton

Uygulamamızda bir class'tan sadece bir nesne olsun istiyorsak ve bunu mimari olarak kodu kullanan her programcı üzerinde zorlamak istiyorsak, o zaman Singleton düzenini kullanabiliriz. Bu düzene göre Singleton olmasını istediğimiz class'ın ilk önce kurucu metotunu (constructor) Java `private` komutu ile dışarıdan saklarız. Böylece kurucu metot sadece class'ın kendisi tarafından kullanılır olur. Bunu yapmazsak herkes class'ı istediği gibi alıp birden fazla nesnesini kullanabilirdi.

Kurucu metotu sakladığımız için, bir yaratıcı metotu bir şekilde sağlamak zorundayız. Singleton class'larında bu metot tipik olarak `instance` adında bir



Şekil 8.8: Facade

yaratıcı metot olur. Bu metot, `static` olmalıdır çünkü Singleton class'ından hâlen tek bir nesne bile mevcut değildir ve ilk çağrılabilecek metot bu yüzden `static` olmalıdır. Bu metot, gerçek nesneyi `static` olan diğer bir `private` değişken üzerinde arar/tutar, buna `uniqueInstance` (tekil nesne) adı verilebilir. Eğer `uniqueInstance` üzerinde bir nesne var ise, o döndürülür, yok ise, bir tane yaratılıp döndürülür. Önce mevcudiyet kontrolü yapıldığı için nesnenin bir kez yaratılması yeterli olmaktadır, ilk `new` kullanımından sonra Singleton'dan geriye gelen nesne hep aynı olacaktır.

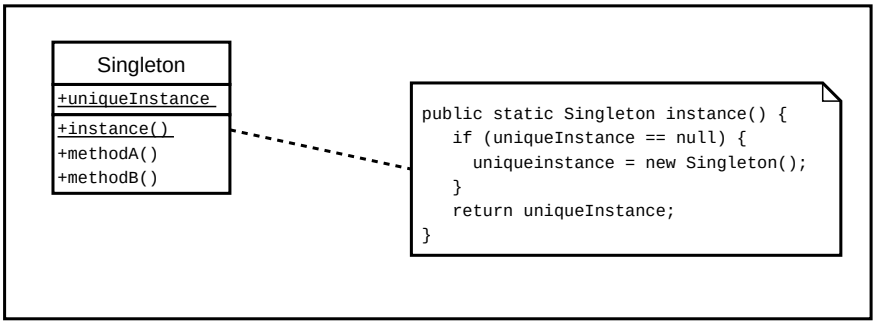
8.3.2 POJO'lar ve İşlem Mantığı

Kalıcılık aracı Hibernate, ya da diğer POJO bazlı teknolojileri kullanırken, bir modelleme tavsiyesini aklımızda tutmalıyız; POJO'lar diğer class'lar gibi bir class'tırlar, üzerlerine `get` ve `set` haricinde çetrefil, işlem mantığı metotları ve fonksiyonları konulmasında bir zarar yoktur.

Hâtta daha ileri giderek bunu yapılmasını şiddetle tavsiye edeceğiz. Son zamanda popüler olan veri, uzaktan nesne çağrısı yapma teknolojilerinin POJO bazlı olmaya başlaması ile oluşan bir izlenim, POJO'ların aptal bir şekilde bırakılıp sadece veri transferi için kullanılmaya başlanmasıdır. Buna hiç gerek yoktur çünkü bir eğer bir POJO, uygulamamızın verisini tutan *yer* ise ve bir class, (nesnesel modelleme açısından) veriler ve işlemleri birarada tutan bir birim ise, o zaman işlemlerimizi POJO'lardan ayırıp bambaşka bir "işlem class'ı" içine gömmemize gerek kalmayacaktır. İşlemimiz bir POJO içindeki veriyi kullanıyor ve modelleme açısında bu temiz bir sonuç veriyor ise, işlemin POJO class'ı içine koyulmasında hiçbir sakınca yoktur.

8.3.3 Diğer Düzenler

Gamma ve arkadaşlarının Tasarım Düzenleri kitabındaki düzenlerin pek azının kurumsal uygulamalar için faydalı olmasına rağmen tasarım düzenleri kavramı,



Şekil 8.9: Singleton

bir prensip ve düşünce sistemi olarak yazılım dünyası için faydalıdır. Hatta söylenebilir ki matematiksel bazı olmayan yazılım mühendisliğine disiplinli ve metodik bir şekilde yaklaşmak isteyenler için tasarım düzenleri ve işleyen yöntemler (best practices), neredeyse izlenebilecek "en formele yakın" yöntemlerdir.

Bu bağlamda, okuduğunuz bu kitap bize göre faydalı tasarım düzenleri ve işleyen yöntemlerin toplamıdır. Fakat bu kitapta Gamma kitabına kıyasla daha dış teknolojiye yakın tasarım düzenleri sunulmaktadır. Bu sebeple buradaki tasarım düzenleri mimari(architecturel) çözümler kategorisine girebilecek tavsiyelerdir. Bunun sebepleri bizim şahsi proje tecrübemize dayanıyor; Tasarım Düzenleri kitabının neredeyse her kelimesini dikkatle izlediğimiz ve yine de birçok badire atlattığımız bir projemizin sonunda teknik liderimiz ve arkadaşım Jim D'Augustine yakınarak şöyle demişti: "Bize tasarım düzenleri değil, mimari düzenler lâzım!"

Son olarak, Karşı Düzen (Anti Pattern) akımından bahsedelim. Karşı Düzenler, normâl tasarım düzenleri aksine ne yapılması gerektiğini değil, *ne yapılmaması gerektiğini* tavsiye ederler. Örneklerden bir tanesi DTO (Data Transfer Object) karşı düzenidir; Bilindiği gibi DTO tekniğini kullanlar, servis tarafından yapılacak her transfer için sadece get/set metotlarından oluşan bir veri class'ı (Hibernate buna POJO diyor) yazmayı salık verir. DTO karşı düzeni DTO class'larının gereksiz olduğunu söylemektedir. Buna biz de katılıyoruz, çünkü artık Hibernate gibi modern yaklaşımlar sayesinde veri alışverişinin tamamen POJO'lar üzerinden olduğu için, ek veri transfer class'larına gerek kalmamıştır. Servis tarafı ve veri tabanı arasında kullanılan nesneler, büyük bir rahatlıkla başka yerlere veri transfer etmek için de kullanılabilirler. (Tarihi olarak insanlar DTO kullanımına herhalde Entity Bean'lerin hantal yapısı yüzünden mecbur olmuştu, fakat Entity Bean'ler artık teknik olarak emekli edildiğine göre, DTO'ya olan ihtiyaç ta ortadan kalkmıştır).

8.4 Mimari

Nesnesel tasarımda birkaç class'ı içeren bir tasarım, nokta vuruş çözümü yâni ufak çapta bir tasarımdır. Bu teknikler uygulamanın ufak bir bölümününün çözümü için kullanılan nesnesel *numaralardır*. Fakat uygulamadaki tüm özelliklerin kodlanması için bazı genel kurallar koyan bir *alt tabaka* olması şarttır. Bu tabaka bazı kodlama kalıplarının içeren ve programcıların miras alması gereken bir üst class, ya da çağırılması proje teknik lideri tarafından herkese duyurulmuş olan yardımcı class'lar toplamı olabilir. Literatürde bu şekilde genel kurallar koyan class'ların ve kod parçalarının toplamına **mimari** diyoruz. Eğer `StrutsHibAdv` örnek projesini düşünürsek, bu projenin mimarisini şöyle tarif edebiliriz:

> "Tüm Hibernate işlemlerinden önce `Session` açılması, ve transaction başlatılması `HibernateSession` yardımcı class'ı üzerinden

> yapılacaktır. Tüm Struts Action'ları `org.mycompany.kitapdemo¬ .actions` altında olacak, ve her sayfa `header.inc` dosyasını kendi kodları içine en tepe noktada dahil edecektir. Basit veri erisimi haricinde sorgu içeren tüm veri erişim işlemleri, bir `DAO` kullanmaya mecburdur; Meselâ `Car` ağırlıklı sorgular için `CarDAO` kullanılması gibi. Her `DAO`, kurucu metotu içinde bir Hibernate transaction başlatmalıdır, ama commit `DAO` commit yapmayacaktır. Struts Action'lar da commit yapamazlar. Commit yapmak, bir Servlet filtresi olan `HibernateCloseSessionFilter`'nin görevidir, ve bu filtre her `.do` soneki için yine herkesin kullandığı `web.xml` içinde aktif edilmiştir. Aynı filtre, `Session` kapatmak, ve hata (exception) var ise, o anki transaction'ı rollback etmek ile de yükümlüdür".

Görüldüğü gibi bu kurallar projedeki her programcının bilmesi gereken kurallardır. Projeye ortasında katılan bir programcı, hemen bir Struts Action yazmaya başlayıp içine alânen bir takım JDBC kodları yazarak veri tabanına erişmeye çalışmayacaktır. Bu projenin kurallarına, *mimarisine* göre, Struts Action'da `HibernateSession` üzerinde açılan `Session` ile, Hibernate yöntemleri üzerinden veri tabanına erişilecektir. Yine aynı kurallara göre yeni programcı `commit`, ve `close` çağrılarını elle yapmaktan men edilmiştir. Projenin mimarisi, bu çağrıları merkezileştirmiş, ve kodun geneline bu şekilde bir temizlik sağlamıştır. Yeni gelen programcının bu kuralı takip etmesi beklenecektir.

Bu şekilde tarif edilen mimarilerin, kod temizliği açısından olduğu gibi, proje idaresi yönünden de etkileri olduğunu anlamamız gerekiyor. Bir mimari bağlamında bazı kuralların konulması ve bazı kullanımların merkezileştirilmesi demek, uygulamamızda önce bitmesi gereken parçanın "mimari kısmı" olduğu sonucunu getirir. Proje idaresi açısında mimari, kodlama açısından seri üretime geçmeden önce bitmesi gereken şeydir. Eğer seri üretimden çıkan her ürünü bir özellik olarak düşünürsek, mimari de fabrika olacaktır. Tabii bu analojiyi dikkatli anlamak gerekiyor, sonuçta işler hâldeki bir program da bir fabrika gibi görülebilir; Bizim burada bahsettiğimiz programın işleyişi değil, o programın kodlama aşamasında programcıların kodlama eforudur.

Proje idaresi bakımından mimarinin önce bitmesine karşı bir argüman, mimarinin özellikler kodlanırken bir yan ürün olarak kendi kendine çıkması beklentisidir; Fakat eğer mimari kod temizliği, hata azaltımı gibi getirmesi açısından önemli ise, önceden mevcut olması gereken bir kavram olduğu ortadadır. Bir mimariyi projenin ortasında kodlarımıza sonradan empoze etmeye karar vermişsek, bu durum mevcut olan kodlar üzerinde yapılması gereken büyük bir değişiklik anlamına gelebilir, ve bu değişiklik için harcanacak efor, o teknik ilerlemenin getireceği herhangi bir avantajı silip yokedebilir. Bu sebeple mimarinin projenin başında hazır olması önemlidir.

Mimariyi tasarlamak, her projede teknik liderin görevidir. Ayrıca mimari ortaya çıkartıldıktan sonra kuralların takip edildiğinin kontrolü de teknik lider üzerinde olacaktır. Öyle ki, proje bittiğinde tüm kod bazı sanki tek bir kişi

yazmış gibi gözükmelidir. İyi bir mimari kodda tekrarı azaltacak, kullanım kalıplarını ortaya koyarak yeni katılan programcılara yön gösterecek ve hata yapma ihtimallerini azaltacaktır. Mimari önceki projelerde alınan dersleri de yansıtan bir kurallar toplamıdır.

Bunun haricinde her özelliği (functionality) kodlayan programcı kendi istediği gibi kodlamakta serbesttir.

Tasarım düzenleri ile mimari arasındaki ilişki şöyledir: Mimarimizin bir kısmı içinde bir tasarım düzeni bir kural olarak ortaya çıkabilir. Örnek olarak 4.1 bölümünde tarif edilen mimari, `Command` tasarım düzeninin her uzaktan nesne erişim gerektiren durum için kullanılmasını mecbur kılmış, böylece bir mimari seçim hâline gelmiştir. Fakat bir tasarım düzeni hiçbir mimarinin parçası olmadan, tek bir özellik için kendi başına da kullanılabilir. Kısacası mimari genelde birçok kişiyi etkileyen, ve genel olan bir kavramdır.