Kurumsal Java

Burak Bayramlı

Kurumsal Java

Editör: Burak Bayramlı
Hazırlayan: Burak Bayramlı
Derleyen: Burak Bayramlı

Birinci Basım: 2005 / İstanbul

ISBN: 975-00427-0-0

Vatandaşlarıma

Kural 1: Teknoloji seçerken her zaman en *kısa*, en *hızlı* işleyecek ve kod bakım külfeti *en az* olacak **dili** seçmeliyiz. Kısalık, yazılan kodun tuttuğu yer ile, bakım külfeti ise kodunuzda bir değişiklik yapıldığında *kaç başka yerin* değişmesi gerekeceği, kodun okunabilirliği ve anlaşılabilirliği ile alâkalıdır. Sektörümüzde, aynen temel bilimlerde olduğu gibi basitlik, *esastır*.

Kural 2: Bir proje için gereklilik kodlarken, aynı projede *teknologlar için teknoloji* üretmemeye dikkat etmeliyiz. Biz, kurumsal programcıların görevi, *işletmeler* için teknoloji üretmektir. Teknoloji amaçlı teknoloji yazmaktan kaçınıp, amaçlarımıza uyan bir teknoloji bulup onu kullanmalıyız.

Kural 3: Prensip edinirken, hatırlaması *en rahat* ve buna oranla etkisi *en fazla* olacak prensibi edinmeli ve prensibi disiplinle takip etmeliyiz.

Kural 4: İnsanlar (programcılar) unutabilirler ve mekanik hatalar yapabilirler. İnsanlardan her anda her işi hatasız yapan biyonik yaratıklar olmalarını beklemeliyiz. En verimli çalışanlar, biyonik adam/kadın olmanın imkansızlığı sebebiyle yapabilecekleri hataları öngörüp, ona göre *kendini koruyucu* alışkanlıkları ve prensipleri edinen insanlardır (bkz. Kural #3).

Kural 5: Ne zaman ki bir teknolojiyi ayakta tutmak için *kod üretmeye* başladık, Kural #1 ışığında o teknolojiyi atma zamanı gelmiştir. Eğer A dilini kullanıp B dilini üretiyor ve olduğu gibi kullanabiliyorsak, A dilinin daha kısa ve basit olduğu apaçık ortadadır.

Kural 6: Kurumsal kodlarımızda kullanabileceğimiz yeni teknolojiler, her zaman *metin bazlı* yeni *diller* olarak ortaya çıkarlar. Bu dillerin çoğunluğu için görsel IDE beklemek boşunadır. Teknolojileri metin bazlı olarak kullanmaya alışmalıyız.

Kural 7: Yazdığımız kodlarda, komut satırında işlettiğimiz komutlarda ve yaptığımız tıklamalarda, tekrarın ve tekerrürün her türlüsünü ortadan kaldırmalıyız. Kurumsal programcılıkta tekrar, hamallıktır. Hamal işlerini, onu en iyi yapacak şeye, bilgisayara bırakmalıyız.

İçindekiler

Önsöz

Bilgi işlem dünyası şahsıma oldukça zevkli bir kariyer sağladı. BASIC, C, ve COBOL programlarını kitaplardan anlamaya ve kodlamaya uğraştığım zamanlardan, şimdi yüksek ölçekli, Unix bazlı Java ve SQL teknolojilerini kullanan küme mimarilerini kurabildiğimiz zamanlar arasındaki projeler, ve bu projelerde kullanılan teknolojiler müthiş bir değişim yaşadılar. Neredeyse başladığımız yerden fersah fersah uzaktayız. Artık daha hızlı, daha az hatalı ve daha yüksek ölçekli programlar yazma şansımız var. Geliştirme araçlarımız daha iyileşti, ve işimizi büyük ölçüde rahatlattı.

Bu kadar teknoloji değişimine rağmen, ilginç olan bazı teknolojilerin hâla aynı kalmasıdır. Meselâ işletim sistemlerinden Unix hâla bizimledir, ve tahminim odur ki daha uzun süre bizle olmaya devam edecektir. Aynı şekilde ilişkisel veri tabanları kendilerini ispatladıkları 80'li yıllardan bu yana her şirkette bir demirbaş hâline geldiler, ve olmaya devam edecekler. Bu gruba en son katılan Java, kendine oldukça sağlam bir yer edindi ve çözüm yelpazesi gittikçe genişliyor. Zannediyoruz ki kurumsal yazılımda optimal bir teknoloji demetine doğru yaklaşıyoruz.

Elinizde tuttuğunuz kitap, bu optimal demeti biraraya getirme yolunda bir ilk denemedir. Bunun artık olabileceğini yakın bir zaman önce farkederek, tüm proje tecrübemizi birbirine uyacak bir şekilde sunmaya karar verdik. Son önemli parça olan kalıcılık (persistence) gelmesi en geciken parçaydı, fakat beklediğimize değdi sanıyorum. Hibernate açık yazılım projesi ile artık yüzümüz kızarmadan bir kurumsal sistem mimarisini *esnek* bir kalıcılık çözümüyle sunmamız mümkün olmuştur.

Dünyada ve Türkiye'de, hâlen, yazılmamış ve yazılması gereken bir kurumsal yazılım potansiyeli (application backlog) mevcuttur, ve bu sebeple sektörümüz için her seviyede ne kadar çok yeni teknik adam yetişirse o kadar iyi olacaktır. Bu amaçla elinizde tuttuğunuz kitap hem programcılar, hem de teknik liderlere yönelik yazıldı. Yazılım projesinin beyni, merkezi olan ve yönünü sağlayan teknik liderlere verebileceğim en iyi desteği bu kitapla vermeye çalıştım; Umarım başarılı olabilmişizdir.

Bu kitabı yazarken tecrübelerinden yararlandığım ve bana örnek olan insanlara teşekkür etmek istiyorum. Onlar olmasa, elinizdeki sentez hiçbir zaman gerektiği gibi olmayacaktı.

Öncelikle teknik lider ve arkadaşım Jim D'Augustine'e yazılım projelerinde sosyal ve teknik iletişimi hakkında bana iyi bir örnek olduğu için teşekkür ederim. Jim, altın kalbi, pür dikkati, bitmeyen enerjisi, ve veri tabanları hakkında engin bilgisi ile kurumsal bir projenin teknik ve teknik olmayan ihtiyaçları hakkında bana çok şey öğretmiştir. Ayrıca Jim'in yazılım gereklilik listesini oluşturma konusunda gösterdiği detaya dikkat, benim için özellikle öğretici olmuştu.

Jim Menard, Emacs maestrosu, dil koleksiyoncusu, deli mucit, kafa insan, bazen konuşmasının yetişemediği hızlı zekası ile herkese iyi bir teknolojist böyle olmalı dedirten kimlik olarak bana önce bol bol `.emacs` numaraları, daha sonra teknolojiye tamamen hakim olmamın ne demek olduğunu gösterdi. Bir de Perl'ü. Herkesin kodunda aynı şekilde yaptığı bir hatayı Jim üç satır Perl script ile değiştirince, bu beceriyi öğrenmem gerektiğini anlamıştım.

Programcılık dünyasındaki şahsiyetlerden alttakilere teşekkürü borç bilirim:

Kitaplarından öğrendiğim ve yazılımlarından faydalandığım kişiliklerden Richard Stallman'a şapka çıkartmak gerekiyor. Yazdığı Emacs adlı editörü 12 sene sonra hâla kullanmaktayım, ve bazı özelliklerini hâla yeni keşfediyorum. Dilden dile atlayan bir teknik lider için her dilde bir mod'u olan Emacs vazgeçilmez bir araçtır; Sn. Stallman'a bu güzel araç, ve daha da önemlisi serbest yazılım hareketini başlatmış olduğundan dolayı için teşekkürler ediyorum.

Bu kitabın dizilmesinde (typesetting) kullandığım LaTeX programının yazarı Donald Knuth önünde saygıyla eğilmekten başka çare yok. Eğer bu harika yazılımı olmasaydı, bu kitabın bu kadar hızlı yazılması ve düzgün format'ta olması mümkün olmazdı. Kitap ayrıca LaTeX üzerine kurulmuş `memoir` paketini kullanmaktadır, ve bu paketin yazarı Peter Wilson'a önce bu paketi yazdığı, ve e-mail üzerinden bazı sorularımı cevaplandırmakta gösterdiği sabır için teşekkür ediyorum.

Nesnesel dünyada, Bertrand Meyer'a nesnesel bir dilin ve tasarımın nasıl olması gerektiğini çok önceden bulduğu ve bizle paylaştığı için saygılar ve teşekkürler sunmak gerekiyor. Dr. Meyer'ın tasarladığı dil Eiffel ticari olarak hakettiği kadar yayılmamış olsa da, bu dili öğrenmiş olmak bizi diğer dillerde daha iyi bir programcı yapmıştır. Ayrıca, Dr. Meyer ile Boston'da bir konferans'ta tanıştığımda `comp.object`, `comp.lang.c++`, `comp.lang.eiffel` gruplarında yazdıklarımı hatırlayıp, imzalaması için ona verdiğim kitaba "Usenet'teki bizi düşünmeye sevkedecek yorumların için şimdiden teşekkürler" kelimelerini yazmış olması da benim için apayrı bir sevinç kaynağı olmuştu. Teşekkürler Dr. Meyer.

Linus Torvalds'a, bize Intel mimarisi üzerinde hızlı çalışan bir Unix olan Linux'u verdiği, ve böylece artık kurumsal yazılımlarda final mimari parçası olarak ucuz ve güvenilir bir Unix tavsiye edebilmemiz ve kullanabilmemize imkan verdiği için saygılar ve teşekkürler sunmak gerekiyor. Yüksek ölçekli

Linux bazlı Java uygulamaları uzun zamandır mevcutlar, bu da Linux'un mevcudiyetinin sektörde çeşitlilik, seçenek ve rekabet yaratmış olduğunun ciddi bir göstergesi. Ayrıca Linus Torvalds, yazılım disiplini hakkındaki görüşleri ve sektör analizleriyle biz kurumsal danışmanlar için tanıdık ve kardeş bir ruh'tur; Hata ayıklayıcı sevmemesine ve teknik proje idare yöntemine gönülden katılıyorum, ve başka birçok katılan olduğunu biliyorum!

Kurumsal Java kitabının tamamı açık yazılım kullanılarak hazırlanmıştır. Kitabın matbaa'dan direk basılmaya hazır PDF'ini üretmek için LaTeX, diyagramlar için Dia[1], resimleri bir format'tan diğerine aktarmak için ImageMagick[2], ve kitapta kullanılmış olan örnek kurumsal kodları test etmek için Uygulama Servisi olarak JBoss[3] ürünü kullanıldı. Şimdiye kadar çok faydalandığım açık yazılım kültürüne ben de JmxMonitor ve CmuCamJ projeleriyle bir şeyler geri vermeye uğraştım, umarım ileride daha fazlasını yapmam mümkün olur.

Kurumsal Java kitabının Türkiye'deki kurumsal yazılım sektörüne yapıcı bir etkisi olmasını umuyor, okurlara başarılar ve mutluluklar diliyorum.

Burak Bayramlı
İstanbul
Kasım 2005

[1] http://www.gnome.org/projects/dia
[2] http://www.imagemagick.org
[3] http://www.jboss.org

Bölüm 1

Giriş

Bu Bölümdekiler

- Kurumsal programcılık tanımı
- Niye bu kitap
- Bölümlerin içindekiler

Kurumsal sistemler (enterprise systems), vazgeçilmez bileşeni olarak bir veri tabanını içeren, kullanıcısına bir önyüz sağlayan ve aynı anda birden fazla kullanıcıyı destekleyen sistemlere verilen addır. Kurumsal yazılımlar, şirketlerin ya da organizasyonların veri alışverişini idare ederler, ve bu idareyle veriyi, stratejik bilgiye donüştürmeye çalışarak yazılımı kullanan şirkete rakiplerine karşı bir fark/üstünlük (differentiator) sağlamaya çalışırlar. Bilgi işlem, ya da IT, bu sektörün diğer adlarıdır.

Kurumsal yazılımları geliştirmekle uzmanlaşmış programcılara, kurumsal programcı adını veriyoruz. Kurumsal programcı, ya işi yazılım olmayan bir şirketin bilgi işlem bölümünde, ya da işi şirketlere yazılım danışmanlığı vermek olan danışman (consulting) şirketlerinde çalışır. Danışmanlık şirketinde kurumsal programcı asal çalışandır, dış dünyaya onun yaptığı iş pazarlanır. Diğer şirketlerde destekleyici konumdadır; Programcı, işi başka bir şey olan bir şirkete yazılım desteği vermektedir.

Bu kitabın *tamamının* odağı, kurumsal programcıdır. Fakat kurumsal bir yazılım üzerinde çalışmıyorsanız bile bu kitaptaki tavsiyelerin bir bölümü işinize yarayabilir; Meselâ veri tabanını sadece okuma amaçlı ve onu da içindeki çoğu veriyi hafızaya yükleyerek kullanan, çok kullanıcılı ama dış dünyaya hiçbir önyüz sağlamayan bir arka plan programı kurumsal program sayılamaz, ama o tür bir yazılıma faydalı olacak tekniklerin bazıları bu kitapta bulunabilir, meselâ bölüm 5, 8, 4. Kullandığınız programlama dili Java bile olmayabilir, bu durumda 5. bölümdeki bir Web uygulamasını dışarıdan test etmek için JMeter kullanımı ilginizi çekebilir. Web uygulaması bile yazmıyor olabilirsiniz, fakat masaüstü, zengin önyüz teknolojilerini kullanan bir uygulamanın bile nesnesel tasarım tekniklerine ihtiyacı olacaktır (8. bölüm).

İş tecrübemiz bağlamında şahsen uzun süre danışmanlık sektöründe çalıştığımız için, kitaptaki anektodların ve tecrübe ışığında paylaşılan tekniklerin çoğunlukla danışman şirketlere dönük olması normâldir. Fakat tecrübemiz gösteriyor ki danışmanlar, süreç kullanımı, tasarım metotları ve diğer birçok kurumsal programcılık metotlarında hep başı çekmiş ve yolgösterici olmuşlardır. Danışmanların böyle olmasının sebebi tamamen içinde yaşadıkları sektörün onların üzerinde mecbur bıraktığı şartlarla alâkalıdır; Müşteri her zaman değişir, sık sık gelen projeler hep yeni bir yeni başlangıç anlamına gelir. Danışmanın müşterisi için yeni proje, kâr potansiyelidir, ama bir yandan yeni teknoloji, risk anlamına gelir ve bu risklerin azaltılması için bazı yollar takip edilmelidir. Projeyi belli bir zamanda yapmaya talep olarak riski üzerine alan danışman şirketi, yazılımı kodlama (delivery) üzerinde en optimal yolu bulmak zorundadır, çünkü geç kalan ya da daha kötüsü bitmeyen bir proje, danışmanın şirketi için para ve ün kaybı anlamına gelir.

Kurumsal Java kitabını takip edebilmek için gereken bilgi seviyesi, temel Java bilgisidir. Bu kitapta miras alma (inheritance), çokyüzlülük (polymorphism) ya da diğer Java sözdizim teferruatları işlenmeyecektir; Eğer bu bilgilere ihtiyacınız var ise, temel Java hakkında güvenilir bir kaynağa başvurmanızı tavsiye ederiz.

1.1 Bölüm Rehberi

Kurumsal Java kitabını düz bir sırada okumak gerekli değildir, zaten bölümler sıralama önem taşıyacak şekilde hazırlanmamıştır. Eğer Web bazlı sistemleri öğrenmek istiyorsanız, 3. ve 2. bölümlere direk olarak gidebilirsiniz. Bu bölümlerde Struts/JSTL bazlı ve arka planda Hibernate teknolojisini kullanan bir Web sistemini kurmak için gerekli tüm teknikleri bulabilirsiniz.

Eğer uygulamamızdaki önyüz, Swing gibi bir zengin önyüz teknolojisi ise, arka planda uzaktan nesne erişim, ya da genel adıyla, dağıtık nesne teknolojisi kullanmak zorundayız, ve bu yaygın teknolojilerin üç tanesi, RMI, EJB Session Bean ve JMS, 4. bölümde detaylı olarak anlatılmıştır. Şahsen kariyerimizin ilk projesinde bir dağıtık nesne çözümü olan CORBA ile kurumsal kodlamaya başlamış olmamız, bu alanda bize oldukça tecrübe (hem iyi hem kötü) yaşattı, ve bu tecrübenin şekillendirdiği tavsiyeleri de bu bölümde bulacaksınız.

Bir kurumsal uygulamanın hızlı çalışması, en az doğru çalışması kadar önemlidir; Bu sebeple performans iyileştirmesi apayrı bir bölüm olarak sunulmuştur (5. bölüm). Bu bölümdeki bilgileri, ilgili olduğu her teknolojinin bölümlerine koyarak dağıtmak mümkündü, fakat performans konusunun kendine has terminolojisi ve hâtta metadolojisi olması sebebiyle, ayrı bir bölüm olarak verilmesi kararlaştırıldı. Ölçekleme ile ilgili konuları aynı bölümde bulabilirsiniz.

Uygulamamızın vazgeçilmez tabakası, tüm verilerin hem çıkış ve hem de sonuç noktası olan ilişkisel veri tabanlarını 9. bölümde anlatmaya çalıştık. İlişkisel tabanların temeli olan ilişkisel model (relational model), ve erişim yöntemi olan SQL burada işlendi. Her ne kadar Hibernate gibi kalıcılık teknolojileri ile artık basmakalıp SQL yazmaktan kurtulsak ta, yine de SQL'i iyi bilmemiz gerekmektedir. Sonuçta Hibernate'in sorgulama dili olan HQL bile, aslında SQL'in nesnesel bir versiyonudur. Ayrıca uygulama içinden kullanıma ek olarak proje sırasında uygulama dışından test amaçlı olarak veri tabanında ne olduğunu anlamak için SQL işletmek gerekmektedir. Bu yüzden SQL anlatımı bölüme dahil edilmiştir.

Nesnesel tasarım tekniklerini anlattığımız 8. bölümde nesnesel tasarımın diğer tasarım tekniklerinden farkını ve nesnesel tasarım yapmak için gereken temel bilgileri vereceğiz. Kurumsal yazılımlarda çok işe yarayacak bazı kritik nesnesel tasarım teknikleri de bu bölümde gösterilecektir. Daha sonra, tüm nesnesel modelleme numaralarının biraraya geldiği teknik mimarin ne olduğunu da bu bölümde târif edeceğiz, ve bu târifin teknik lider olmak isteyen herkes tarafından okumasını tavsiye ediyoruz.

Geliştirme ve kabul testleri, kitabımızın önemli bölümlerinden biri olacak (7. bölüm); Geliştirme sırasında kendi kendimizi kontrol mekanizması olan birim testlerini yazmayı bu bölümde işliyoruz. Ayrıca gereklilikleri dışarıdan (önyüzden) test etmek için gereken JMeter aracını kullanmayı da işleyeceğiz. Test etme işlemi, projemizin ayrılmaz bir parçası olarak kabul edilmelidir. Test yöntemlerinden özellikle birim testleri geliştirme sürecine ne kadar dahil edilebilirse, kendimizi o kadar iyi/başarılı hissetmeliyiz.

Kitabımızın teknik tarafı, işler hâlde ve test edilmiş örnek Java kodları ile destekleniyor; Bu örnek programları düzgün, temiz yapıda ve optimal çalışır hâle getirmek için çok uğraştık, ve en az kitabın metni kadar önemli olduğunu düşünüyoruz. Örnek kodların nereden indirileceği ve kullanılacağı A.1 bölümünde anlatıldı, kullanmak için gereken programlar JBoss Uygulama Servisi, Java derleyicisi, ve Ant derleme sisteminin nasıl kurulacâgı yine bölüm A içinde bulunabilir. Projelerimizde sıkça kullandığımız faydalı programların tümünün kuruluşu bölüm A içinde anlatılmaktadır, buna tek istisna MySQL, Oracle ve PostgreSQL'ın nasıl kurulacağını anlatan 9.5 bölümüdür.

Bir uygulamanın kodlaması bittikten sonra, işleyeceği nihai yer olan sonuç (production) ortamına nasıl gönderileceği, ve buraya gönderildikten sonra nasıl izlenip takip edileceği 6. bölümde işleniyor. Bir kurumsal uygulama tipik olarak birçok süreçten meydana gelir; Uygulama servisi, veri tabanı ve web server olarak düşünürsek en az üç tane Unix sürecimiz olacaktır. Bu süreçlerin ayakta kalması ve çökerse tekrar ayağa kaldırılmaları kurumsal sistemlerde gerekli teknik numaralardandır; Bunları yapabilmek için gerekli script'leri 6. bölümde bulacaksınız. Uygulamanızdan istatistik paylaşmak ve bu istatistiklerin üzerine alarm şartları koyarak sürekli, otomatik olarak takip edebilmek için önce Java JMX teknolojisi ile bunları nasıl dış dünya ile paylaşacağımızı, ve sonra tarafımızdan yazılmış bir açık yazılım projesi olan JmxMonitor ile bu istatistikleri nasıl takip edebileceğimizi göreceğiz.

1.2 Türkçe Kullanımı

Kitabımızdaki Türkçe kullanımında Sayilar ve Kuramlar blogunda takip ettiğimiz aynı dikkati vermeye özen gösterdik. Fakat örnek kodlarda ve karşılığı yeterince iyi olmayan kelimelerde o kelimelerin İngilizcesini kullanmaya karar verdik.

Örnek kodlar için bunu planlamamıştık. Fakat Türkiye'deki programcıların İngilizce kelimeler, ve değişken isimleri ile kodlama hususunda oldukça hiç bir problemi olmadığını gözlemledik. Ayrıca ilerideki bir tarihte elinizde tuttuğunuz kitabın İngilizce'ye çevirilmesi sözkonusu olabileceği için, örnek kodların İngilizce olması kararlaştırıldı.

Karşılığı bulunamayan (ya da beğenilmeyen) kelimelerde, meselâ bir Ant *task*'i, Ant *target*'i, ya da Unix *script*'i gibi, o kelimelerin olduğu gibi kullanılması kararlaştırıldı. Ant kelimelerindeki sorun, bu terimlerin herhangi bir karşılığının sektörde kimse tarafından ortaya atılmamış olmasıydı, yâni ortada bir konsensüs yoktu. Eğer bu kelimeler yerine "görev", ve "hedef" karşılıklarını kullanacak olsaydık, o zaman Ant İngilizce belgeleri ile ayrılığa düşecek, ve okuyucumuzun kafasını karıştırmış olacaktık. Ayrıca bu kelimeler pek yaygın olan ve başka amaçlar için kullanılan gerçek hedef ve görev kelimeleriyle aynıdırlar, bu yüzden hiç değilse *ayrı bir şey* olduğu daha belli olan task ve target kelimelerinin kullanılması bize daha mantıklı gelmiştir. Aynı durum, Türkçe karşılığı "işlem" olan transaction kelimesi için de geçerliydi.

Script Türkçe'ye *betik* olarak çevirilen, ve Türkçesi ne yazık ki sektörde tutmayan bir kelimedir. Bu kelimenin kullanımını uzun süredir takip ediyor ve şahsen de cümlelerimizin parçası yapıyorduk, fakat uyum bakımından yeterli olmadığına kanaat getirip, kelimenin orijinalini desteklemeye karar verdik. Betik kelimesi, ses yapısı ve uyumu açısından bir türlü bir script sözcüğünün gücüne erişememiştir. Bizim Türkçe kelime seçerken kıstasımız, bir kelimenin anlatım (cümle) içinde akıcı bir şekilde yer alabilmesi, ve o kelime üzerinden anlatılanın, anlaşılmasıdır. Ayrıca, bu kitap okunduktan sonra, burada edinilen bilgiler hakkında yeni bilgiler edinilmek istendiğinde, İngilizce dokümanların programcıya çok fazla yabancı gelmemesi diğer bir amacımızdı. Bu amaçla Türkçesini kullandığımız bir terimin yanına muhakkak İngiliz'cesini parantez içinde yazmaya dikkat ettik. Böylece, programcı isterse, o kelimeyi Google'da arayabilecek, ya da ilgili programın dokümantasyonunda tarama yaparak konuyla ilgili diğer teferruatları öğrenme şansına kavuşacaktır.

1.3 Ek Bilgiler

Kitaba yorumlarıyla yardımcı olan ve hataların düzeltilmesine katkıda bulunan Java Teknoloji Derneği'ndeki arkadaşlardan veri tabanı bölümünü okuyan Murat Acar'a, Hibernate bölümünü okuyan Melih Sakarya'ya ve Web bölümünü gözden geçiren Altuğ Altıntaş'a (ve yorum yapan diğer arkadaşlara) buradan teşekkür ediyorum. Geri kalan tüm hataların sorumluluğu bana aittir.

Hatalar bulundukça, `http://sayilarvekuramlar.blogspot.com/2008/¬ 11/kurumsal-java-hatalar-1-basm.html` adresinden (düzeltmeler ile birlikte) paylaşılacaktır. Kitap ile ilgili diğer bilgileri `http://sayilarvekuramlar.¬ blogspot.com` adresinden bulabilirsiniz.

İyi okumalar, ve başarılar.

Bölüm 2

Hibernate

Bu Bölümdekiler

- Hibernate projesini kuruluşu, dizin yapısı, Ant build.xml
- Öğe ve kolon eşlemesi
- Nesneler arası ilişkiler
- Sorgulama

Tartışmasız, veri tabanına erişim, yâni bir kaydı eklemek, silmek, güncellemek gibi işlemler, kurumsal uygulamaların en önemli bölümünü teşkil eder. Yaygın bilinen bir istatistiğe göre, bir kurumsal uygulamanın veriye erişiminde harcanan zaman, uygulamanın tamamında harcanan zamana oranla %80 kadar bir zaman teşkil etmektedir. Bu oran, çok büyük bir rakamdır ve kurumsal uygulamalarda veriye erişimin önemine işaret eder.

Java dünyasında veriye erişim için en temel yöntem, JDBC (Java Database Connectivity) adı verilen bir kütüphane üzerindendir. JDBC, Java ile veri tabanlarına erişim için piyasaya çıkan ilk çözümdür. Çok temel olması sebebiyle, JDBC'nin programcıya yapmasını izin verdiği işlemler veri tabanına yakın ve SQL ile direk alâkalıdır: Sorgu işletmek, sorgulardan gelen sonuçları listeleyebilmek, depolanmış işlemleri (stored procedure) çalıştırabilmek gibi. Veriye erişim teknolojileri klasmanında JDBC, DB ve uygulama arasında *çok ince bir tabakadır* denebilir. Veri, uygulamaya "tablolar kümesi" olarak gözükür ve bu tablolar kümesi üzerindeki işlemler ile veri eklenir, güncellenir ve ya silinir.

2.1 Faydalar

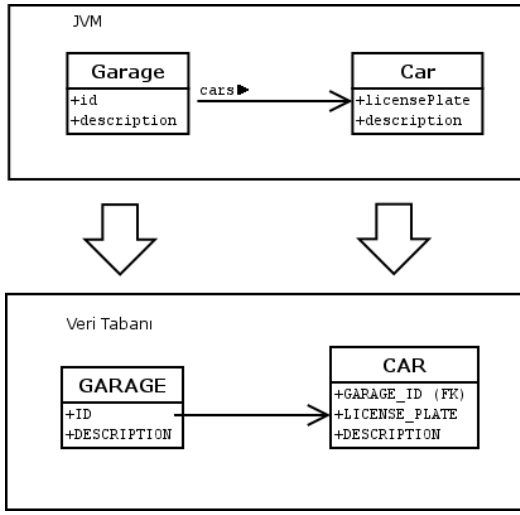
Kıyasla nesnesel programcılar, veriyi nesneler topluluğu olarak görmeye alışkındırlar. Nesnesel tasarımı takip eden bir programcı için veri, bir nesne içinde tutulabilen, işlenebilen, değiştirilebilen ve Java temel tipleri üzerinden geri verilebilen bir kavram olmalıdır. Nesnesel dillerin izole etme (encapsulation) gibi özellikleri, bu tür kodlama stilini ayrıca özendirmekte, idare edilir hâle getirmektedir.

Kod temizliği açısından da nesne odaklı veri idaresinin daha makbul olduğu açıktır. Uygulamamızın hem veri hem görsel tabakasında veriyi alışık olduğumuz kodlama ve taşıyıcı birimler üzerinden görmek isteriz. Bu birimler, içinde get ve set içeren basit Java nesneleri olacaktır. Bu basit Java nesne türüne yeni literatürde POJO adı da verilmektedir. POJO kelimesinin açılımı **P**lain **O**ld **J**ava **O**bject (Basit Java Nesneleri) olarak bilinir. Örnek bir POJO aşağıda gösteriliyor.

```
public class Car {

    public Car() { }

    String licensePlate;

    public String getLicensePlate() {
        return licensePlate;
    }

    public void setLicensePlate(String newLicensePlate) {
        this.licensePlate = newLicensePlate;
    }
```

```
    String description;

    public String getDescription() {
        return description;
    }

    public void setDescription(String newDescription) {
        this.description = newDescription;
    }
}
```

`Car` nesnesi bir arabayı temsil etmektedir, ve veri olarak içinde `licensePlate` (plaka) ve `description` (târif) öğelerini taşımaktadır. Araba nesnesi, pür bir Java nesnesi olduğu için, her türlü Java işlemine tâbi olabilir; Bir listeye eklenebilir, öğeleri dinamik olarak sorgulanabilir, hatta başka bir JVM'e bile gönderilebilir. Yapılacak işlemler Java programını yazan programcının hayali ile sınırlıdır.



Şekil 2.1: İki Nesne ve Tablo İçeren bir Eşleme

Hibernate gibi teknolojilerin amacı, kurumsal programcıların veriye erişim işlemlerini tamamen POJO'lar üzerinden yapmalarına imkan vermeleridir. Bir POJO'nun veri tabanına nasıl eşlenmesi gerektiğini eşleme (mapping) dosyası üzerinden anlayabilen Hibernate, bu noktadan sonra tüm gerekli SQL komutlarını dinamik olarak arka planda kendisi üretebilecektir. Yâni bir tablo üzerinde JDBC ile yaptığımız `SELECT, UPDATE, INSERT, DELETE` işlemlerinin tümü, tek bir eşleme dosyası ile Hibernate tarafından otomatik olarak yapılacaktır.

Aşağıda araba nesnemizi veri tabanı tablosuna eşleyen bir Hibernate dosyasını görüyoruz.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="org.mycompany.kitapdemo.pojo">
  <class name="Car" table="car">
    <id name="licensePlate" column="license_plate">
      <generator class="assigned"/>
    </id>
    <property name="description" column="description"/>
  </class>
</hibernate-mapping>
```

Eşleme dosyasındaki `Car` nesnesi, veri tabanındaki `car` tablosuna eşlenmektedir, ve tablonun her kolonunun hangi nesne öğesine eşlenmesi gerektiği teker teker belirtilmiştir. Bundan sonra bir `Car`'ı veri tabanına eklemek, şu Hibernate komutlarından ibarettir.

```
Car car = new Car();
car.setLicensePlate(‘‘52 THU 34’’);
car.setDescription(‘‘araba tanım 1’’)
Session s = HibernateSession.openSession();
s.save(car);
```

Silmek için `s.delete(car)`, güncellemek (ya da eklemek) için `s.saveOrUpdate(car)` kullanabiliriz.

Gördüğümüz gibi, her tablo için dört SQL komutu yazmak yerine tek bir eşleme dosyası üzerinden Hibernate veri erişiminin faydaları açıktır. Kural #1 bağlamında Hibernate veri erişimi daha kısa bir dille gerçekleştirilmiştir, ve gene Kural #1 bağlamında bir Hibernate eşleme dosyası daha rahat bakılabilir bir koddur. Eğer, ileri bir zamanda `car` tablo isminin değişmesi gerekse, SQL şartlarında tüm `SELECT, UPDATE, INSERT, DELETE` komutlarındaki tablo isminin değiştirilmesi gerekecekti. Ya da, eğer bir kolon ismi değişse, o kolonu kullanan tüm SQL'lerin değişmesi gerekecekti; Fakat Hibernate kullanıyorsak, değişen kolonun sadece *nesne eşlemesini* değiştirerek, Hibernate üzerinden veriye erişen işlem kodlarda tek bir satır değiştirmemiz gerekmez.

Bu bölümün geri kalanında, kuruluş ve ana Hibernate kavramlarını işleyeceğiz. Eğer bu kavramları atlayıp, direk Hibernate'in nasıl kullanıldığını öğrenmek istiyorsanız, tavsiyemiz 2.2.4 ve 2.2.5 bölümlerine göz gezdirip, 2.4 bölümüne atlamanızdır.

2.2 Kurmak

Hibernate üzerinden erişmek istediğimiz her nesne (POJO) için bir eşleme dosyası yazılmalıdır. Fakat proje bazında, sadece bir kez yapılması gereken ayarlar şunlardır. İlk önce, `hibernate.cfg.xml` gerekecek. `hibernate.cfg.¬ xml` dosyası, Hibernate'e

- Veri tabanına nasıl erişeceğini
- Hangi önbellek paketini kullanacağını
- Önbellekleme yönteminin ne olacağını
- Hangi veri tabanı bağlantı havuzu (connection pool) kullanacağını

söyler. Ayar dosyasının `CLASSPATH`'teki bir dizin içinde, ya da `CLASSPATH`'teki bir `jar`ın en üst seviyesinde olması gereklidir. Buna uygun `jar` paketlemesi yapan örnek bir Ant `build.xml` script'ini, `SimpleHibernate` projesinde bulabilirsiniz.

Altta `Car` için hazırlanmış örnek bir `hibernate.cfg.xml` dosyasını gösteriyoruz.

Liste 2.1: hibernate.cfg.xml

```
1  <!DOCTYPE hibernate-configuration PUBLIC
2  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
3  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
4
5  <hibernate-configuration>
6
7    <session-factory name="foo">
8
9      <property name="hibernate.connection.driver_class">
10       org.gjt.mm.mysql.Driver
11     </property>
12     <property name="hibernate.connection.url">
13       jdbc:mysql://localhost:3306/cars
14     </property>
15     <property name="hibernate.connection.username">user</property>
16     <property name="hibernate.connection.password">pass</property>
17     <property name="hibernate.connection.provider_class">
18       org.hibernate.connection.C3P0ConnectionProvider
19     </property>
20     <property name="hibernate.c3p0.min_size">
21       5
22     </property>
23     <property name="hibernate.c3p0.max_size">
24       10
```

```
25      </property>
26      <property name="hibernate.c3p0.idle_test_period">
27        60
28      </property>
29      <property name="dialect">org.hibernate.dialect.MySQLDialect
30      </property>
31      <property name="hibernate.cache.provider_class">
32        org.hibernate.cache.OSCacheProvider
33      </property>
34
35      <mapping resource="org/mycompany/kitapdemo/pojo/Car.hbm.xml"/>
36      <class-cache
37          class="org.mycompany.kitapdemo.pojo.Car"
38          region="Simple"
39          usage="read-write"/>
40
41    </session-factory>
42
43  </hibernate-configuration>
```

Satır satır açıklama:

- **10, 13**: 10. satırda verilen değer, Hibernate tarafından hangi JDBC sürücüsünün kullanıldığını belirtiyor. Örnekte kullanılan değer, `org.¬ gjt.mm.mysql.Driver` adlı MySQL sürücüsüdür. Sürücü parametresi, direk JDBC'ye geçilen parametrelerden biridir (evet Hibernate kendi işleyişi için, arka planda JDBC kullanıyor). MySQL sürücüsünün veri tabanına erişimi için gerekli bilgiler 13. satırda verilmiş; Meselâ hangi makinaki hangi veri tabanı bilgisi.

- **15, 16**: Tabana bağlanmak için gerekli kullanıcı ismi ve şifre.

- **18**: Veri tabanı bağlantı havuzu (connection pool) paketi. Örnekte seçilen paket, C3P0 adlı pakettir. Havuzlar hakkında detayları 2.2.1 bölümünde bulabilirsiniz.

- **20 - 28**: Bağlantı havuzu için gerekli parametreler (havuzun olabilecek en az, ve en çok büyüklüğü).

- **29 - 32**: Hangi veri tabanı için, nasıl SQL üretilmesi gerektiği de burada belirtiliyor. Sürücü seçmek (MySQL, Oracle, vs), Hibernate'in nasıl SQL üretmesini belirlemek için yeterli değildir. `dialect` parametresi ile, üretilen SQL'in hangi taban için olduğu kesin olarak Hibernate'e bildirmelidir.

- **34**: Uygulamamızın kullanacağı POJO'ların listesi `mapping resource` ile Hibernate'e bildiriyoruz. Örneğimizde bir tane POJO tanımlıyoruz:

`Car`. Hibernate, `mapping resource`'da belirtilen `hbm.xml` dosyasını `CLASSPATH`'te bulacak, eşleme tanımlarını işleyecek ve POJO üzerinden SQL üretmeye hazır hâle getirecektir.

- **35 - 38**: Önbellek ayarları. (Bu konunun detayları 2.2.2 bölümünde gösterilecektir).

2.2.1 Bağlantı Havuzları (Connection Pools)

Bir bilgi işlem uygulamasının yapabileceği en pahalı işlemlerden biri veri tabanından bir bağlantı (connection) almaktır. Özellikle uygulamanız Oracle gibi ciddi bir taban üzerinde çalışıyor ve bu tabanın kontrolü güvenliğe çok önem veren bir admin'in de elinde ise, bağlantı alma sırasında birçok işlemden (hafıza hazırlama, vs. gibi) geçileceği için, bağlantı alma işlemi kurumsal uygulamalarda en pahalı ve yavaş işlemlerden biri hâline gelmektedir.

Bu yüzden, sistemin cevaplama zamanını (response time) hızlandırmayı amaçlayan kurumsal programcılar her işlem (transaction) başında sürekli yeni bir bağlantı açıp, işlem sonunda bu bağlantıyı kapatan kodlar yazmaktan kaçınırlar. Bunun yerine, bağlantılar uygulama başında bir kez açılır ve bir tür *bağlantı önbelleğinde* tutularak uygulamanın işleyişi boyunca açık tutulurlar. Bağlantı önbellekleri, bir nevi havuzdur; Bağlantılar açılıp kapanmazlar, sadece bu ortak havuzdan alınıp geri verilirler. Bu havuzun belli sayıda bağlantı içeren bir havuz olması özellikle önemlidir, çünkü her taban, aynı anda, sadece belli sayıda bağlantıya servis verebilir, ve optimal bağlantı sayısının üstünde servis vermenin veri tabanınızın performans açısından kötü sonuçları olacaktır.

Piyasada Java/JDBC odaklı birçok havuz ürünü mevcuttur. DBCP, Proxool, C3P0 adlı açık yazılım projeleri bu ürünlerden sadece birkaçıdır. Bahsedilen projelerin her biri, Hibernate tarafından kullanılabilen (`hibernate.connection.provider_class` parametresi ile) projelerdir.

2.2.2 Önbellek

Bir bilgi işlem uygulaması, eğer bir veri birimini tabandan *yazdığından* daha fazla *okuyorsa*, o veri birimi tabandan servislemek yerine hafızadan servislemek daha hızlı olacaktır. Önbellekleme (caching) tekniğinin altında yatan fikir özet olarak budur. Bir bilgi işlem uygulamasının %80 kadar zamanının veri tabanında geçtiğini düşünürsek, disk'e ne kadar az gidersek veri erişimini (ve bilahere uygulamanın bütününü) o kadar hızlandırmış oluruz.

Önbellek kullanımı, tek kullanıcılı ve çok kullanıcılı ortamlarda büyük performans iyileştirmeleri getirebilir. Tek kullanıcılı ortamda, aynı veriye birkaç kez erişen (aynı) kullanıcı, birinci kullanımdan sonra aynı veriye tekrar eriştiğinde o veriyi önbellekten çok hızlı bir şekilde alabilmiş olur. Çok kullanıcılı ortamda (aynı JVM'in birçok eşzamanlı kullanıcıya hizmet verdiği şartlarda) önbellek

kullanımı daha da avantajlıdır; Kullanıcı A'nın önbelleğe getirmesini tetiklediği bir nesneyi, kullanıcı B olduğu gibi hafızadan kullanabilecektir.

Hibernate, kontrolü altındaki tüm nesneleri programcıya hiçbir ek külfet getirmeden programcının istediği herhangi bir önbellek paketi üzerinde tutmasına izin vermektedir. Bu şimdiye kadar yalın JDBC teknikleri ile elimizde olmayan müthiş bir esnekliktir. Düşünün: Daha önce `HashMap` ya da elle yazılan bir yapı üzerinde olan önbellekleme, artık, Hibernate'e verilen tek bir parametre sayesinde otomatik olarak yapılıyor olacaktır.

Hibernate öncesi elimizle yapmamız gereken külfetli işlemler, pseudo kod olarak altta gösterilmiştir:

```
def araba_yükle(licensePlate : String)
  begin
     Car tablosunda licensePlate kimlikli veriye eriş.
     Bu veri, hafızadaki HashMap'te mevcut mu?
     begin
        Evet ise, hafızadakini döndür.
        Hayir ise,
         Tabandan yükle
         HashMap'e yaz
        Geri döndür
     end
  end
end

def araba_sil(licensePlate : String)
begin
   Veri HashMap'te mevcut mu?
   begin
       Evet, hafizadakini sil, sonra tabandan sil
       Hayir, tabandan sil
   end
end
```

Hibernate, kontrolü altında olan nesnelerin nasıl ve ne zaman önbellekleceğini önbellek paketlerine direk söyleyebilir, çünkü tabandan okuma, silme, yazma ve güncelleme işlemlerinin tümü zaten Hibernate üzerinden yapılan işlemlerdir. Hibernate, bu noktalara çengellenmiş olan önbellek kodları sayesinde bir önbellek paketini programcıya hiçbir ek külfet getirmeden idare edebilir. Yukarıda pseudo kod olarak gösterilen eylemlerin tamamı Hibernate için gereksiz hâle gelmiştir (Kural #2).

Ayrıca Hibernate, tek JVM'de işleyen türden önbellekleri kullanabildiği gibi, *birden fazla JVM* üzerinde çalışan ve birbirini ağ üzerinden güncelleyebilen *dağıtık önbellek* ürünlerini de kullanabilir. Dağıtık önbellekler, ağ üzerinden birbirini senkronize edebilen ürünlerdir. Önbellek kullanımı ve dağıtık önbellek konularını 5.4.5 bölümde daha detaylı olarak göreceğiz.

2.2.3 Örnek Proje Dizin Yapısı

Bir Hibernate projesinde gereken minimal dosyalar ve dizin yapısı altta belirtilmiştir.

```
+- SimpleHibernate
| +- build
| +- lib
| | +- activation.jar
| | +- ant-antlr-1.6.2.jar
| | +- antlr-2.7.4.jar
| | +- c3p0-0.8.4.5.jar
| | +- c3p0.license.txt
| | +- cglib-full-2.0.2.jar
| | +- commons-collections.jar
| | .. ....
| | +- jaxen-1.1-beta-4.jar
| | +- junit-3.8.1.jar
| | +- log4j-1.2.9.jar
| | +- mysql-connector-java-3.0.16-ga-bin.jar
| | +- oscache-2.1.jar
| | +- xalan.jar
| | +- xml-apis.jar
| +- resources
| | +- hibernate.cfg.xml
| | +- log4j.properties
| | +- oscache.properties
| +- src
| | +- java
| | | +- org
| | | | +- mycompany
| | | | | +- kitapdemo
| | | | | | +- dao
| | | | | | | +- SimpleCarTest.java
| | | | | | +- pojo
| | | | | | | +- Car.hbm.xml
| | | | | | | +- Car.java
| | | | | | +- service
| | | | | | | +- HibernateSession.java
| | | | | | +- util
| | | | | | | +- AllTest.java
| | | | | | | +- ClassPathFile.java
| | | | | | | +- TestUtil.java
| | +- sql
| | | +- sample_data.sql
| | | +- tables_mysql.sql
| +- build.properties
| +- build.xml
```

Bu dizinler ve içerikleri, `SimpleHibernate` projesinde bulunabilir. Şimdi, bu dizinde bulunan birimleri görelim.

2.2.4 Hibernate Session

Hibernate ile veriye erişmek için, öncelikle bir `org.hibernate.Session` yaratıp bu nesne üzerinde `openSession` metotunu çağırmak ve bir Hibernate oturumu başlatmak gerekir. Örnek kodlarda görülen `HibernateSession` class'ı, içinde bir Hibernate `SessionFactory` barındıran yardımcı bir nesnedir. Bu yardımcı nesne örnek proje için yazılmış ve hazır olan bir koddur ve projenize olduğu gibi eklenebilir.

Bir `org.hibernate.Session` almak ve açmak (openSession) için `HibernateSession` nesnesindeki `static` çağrıları kullanabilirsiniz. `HibernateSession` class'ının tüm metotları `static` metotlardır, bu yüzden `HibernateSession`'ın bir nevi Singleton nesnesi olduğu söylenebilir. Demek ki bir JVM içinde sadece bir aktif `HibernateSession` olabilir. `HibernateSession`'daki static blok, `HibernateSession` kullanılır kullanılmaz hemen çağrılacağı için `hibernate.cfg.xml` dosyaları okunmaya başlar, ve referans edilen tüm `hbm¬.xml` dosyaları takip edilerek gereken eşlemeler hafızaya alınır, işlenir, kontrol edilir ve önbellekte tutulur.

Static blok içinde yapılan diğer önemli bir işlem, bağlantı havuzu için 'en az gereken bağlantı (minimum connection count)' sayısı kadar bağlantının hemen açılmasıdır. Bağlantı havuzlarını anlattığımız 2.2.1 bölümünde bağlantı açmanın bir kurumsal uygulamada yapılabilecek en yavaş işlemlerden biri olduğunu açıklamıştık. Bu sebeple, `HibernateSession`'a erişen *ilk kod parçasının* kullanıcıya dönük bir kod parçası olmaması iyi olur. Niye? Çünkü, bir kullanıcının istediği basit bir sayfa/ekran için, 100 tane taban bağlantısının açılmasını beklediğini düşünün! Bu çok yavaş bir sayfa yüklemesi olacaktır! Tabii ki aynı sayfa ikinci kez istendiğinde, cevap çok çabuk dönecektir (gereken tüm bağlantılar artık açılmıştır), fakat o ilk yavaş sayfa yüklemesi ile kullanıcının ağzında kötü bir tat bırakmamak gerekir. Bu nedenle `HibernateSession`'a erişen ilk kod parçasının kullanıcıya dönük kodlardan ayrı, sadece *uygulama başlarken* çağrılacak *ayrı* bir kod parçasından olmasına özen gösterin.

Uygulamanın geri kalan kısmında, `HibernateSession` üzerinden aldığınız `Session` nesnesi ile istediğiniz veri tabanı işlemini gerçekleştirebilirsiniz. Tek nesne yükleme, yazma, silme, ve sorgulama metotlarının hepsi `Session` üzerinde bulunmaktadır.

İşimiz bittikten sonra ise, `HibernateSession.closeSession` kullanarak `Session`'ı kapatmamız gerekir, çünkü JDBC taban bağlantısını muhafaza eden nesne `Session` nesnesidir; `close` çağrısı ile veri taban bağlantısının bağlantı havuzuna dönmesini tetiklemiş oluruz. Eğer uygulama içinde hiç `close` çağrısı yapmamış olsaydık, uygulamamız bir süre sonra havuzdaki tüm bağlantıları bitirecek ve uygulama işleyişine devam edemez hâle gelecektir.

2.2.5 Hibernate Transaction

Veri tabanında okuma, yazma, silme işlemleri yapan bir kod parçası, işine muhakkak `beginTransaction()` ile başlamalı ve `commitTransaction()` ile işini bitmelidir. Özetle Hibernate ile her yaptığınız her işlemi alttaki kod bloğuyla sarmanız yararlı olur.

```
try {
    Session s = HibernateSession.openSession();
    HibernateSession.beginTransaction();

    // ...
    // Hibernate işlemleri
    // ...

    HibernateSession.commitTransaction();

} finally {
    HibernateSession.closeSession();
}
```

Kapanış `closeSession` işleminin `finally` içinde konması önemlidir, çünkü Hibernate operasyonunuz sırasında exception atılsa da, atılmasa da `closeSession` çağrısının yapılması böylece garanti olacaktır.

`Transaction` alâkalı çağrıların `HibernateSession` vasıtası ile yapıldığına dikkat ediniz. Diğer bazı Hibernate örneklerinde bir `Transaction` nesnesinin session'dan alınıp, üzerinde direk olarak `beginTransaction` ve `commit` çağrılarının yapıldığını görürüz. Biz, iki sebeple bunu yapmayarak, tüm `Session` ve `Transaction` alâkalı işlemleri `HibernateSession` üzerine aldık, çünkü

1. `Session` ve `Transaction` ile alakalı tüm işlemlerin tek bir yerde, tek bir class üzerinden olması (kod idaresi ve temizliği açısından).

2. `Session` ve `Transaction`'ın sadece bir thread'e bağlı olmasını sağlamak

`Session` ve `Transaction`'ı niye bir thread'e bağlamak istedik? Çünkü içinde Hibernate işlemleri olan metotların birbirini zincirleme çağırması gerektiğinde, her yeni çağrının yeni bir transaction ve session başlatmasını engellemek istiyoruz. Bunun sebebi, kurumsal uygulamaların %99'unda bir metot diğer metotu çağırınca, her iki metotun da aynı transaction altında olmasının çok yaygın olmasıdır. Örnek olarak, meselâ bir bankada iki hesabı arasında para transferi yapan bir kullanıcının `transferMoney` adlı bir çağrı yaptığını düşünün:

```
AccountService service = ...
service.transferMoney(100, '22993002','3399499'
```

Gereken class tanımları şöyle olsun:

```
public class AccountService {
    public void transferMoney(double howMuch,
                              String accountFrom,
                              String accountTo) {
       Account fromAccount = ...
       Account toAccount = ...
       fromAccount.subtract(howMuch);
       toAccount.add(howMuch);
    }
}

public class Account {
  public void subtract(double howMuch) { .. }
  public void add(double howmuch) { .. }
}
```

Bu kod'da görüldüğü gibi, bir hesaptan para eksiltmek ve yeni hesaba para eklemek *aynı transaction altında* olmalıdır. Aksi takdirde `subtract` (eksiltme) işleminden sonra servis makinasının fişi çekilmiş olsa, ekleme yapılmadan sistem çökmüş olacağı için para kaybolmuş olurdu! Bu yüzden `transferMoney` ile başlayan metot zincirinin tamamı aynı transaction altında olmalıdır. Böylece bu transaction altındaki tüm işlemlerin ya hepsi *başarısız* olur, ya da hepsi aynı anda *başarılı* olur.

İşte bu yüzden metot zinciri ile transaction sürekliliğini birbirine bağlamalıyız. Bunu yapmanın en basit yolu ise, o anda işlemde olan thread ile (`Thread.currentThread()`) transaction'ı birbirine bağlamaktır. Bunu `HibernateSession` içinde `ThreadLocal` kodlama kalıbını kullanarak [1, Sf. 301] yapabiliriz. Bu kalıp, o anki thread, Hibernate `Session` ve `Transaction` arasında bir bağlantı kurmaktadır. Böylece zincirleme birbirine çağrı yapan metotlar aynı `Thread` içinde olacağı için, aynı `Transaction` ve `Session` kullanılabilmiş oluruz.

2.3 Kimlik İdaresi

İlişkisel veri tabanlarında bir satırı tekil (unique) olarak belirleyebilmek için, anahtar görevini görecek "bir ve ya daha fazla" kolon tanımlaması gerekir. Bu kolon(lar)a, ilişkisel (relational) veri tabanı dünyasında asal anahtar (primary key) adı veriliyor. Tek bir satırı bulmak için SQL ile sorgu yaparken `WHERE` kısmında kullandığımız filtreleme kolonları, asal anahtar kolonlarıdır.

Hibernate, bir tablodaki tek bir satırı tek bir nesneye eşlemek ister. Tablodaki her kolon da, bir POJO üzerindeki öğe (attribute) ile eşlenecektir. Bu eşleme, eşleme dosyasında `property` kelimesi ile gerçekleştirilir. Fakat, asal anahtarı oluşturan kolon ve onun eşlendiği nesne öğesi, ayrı bir şekilde Hibernate'e bildirilmelidir. Çünkü Hibernate, bu öğeyi, diğerlerinden ayrı bir şekilde, nesneyi bulma, tanımlamak için kullanılacak, yâni nesnenin tekilliğini belirten

bir öğe olacak kullanacaktır. Bu öğe, komut eşleme dosyasında `<id>` ile belirtilir. `Car` örneğimizi ele alırsak:

```
<hibernate-mapping package="org.mycompany.kitapdemo.pojo">
  <class name="Car" table="car">
    <id name="licensePlate" column="license_plate">
      <generator class="assigned"/>
    </id>
    <property name="description" column="description"/>
  </class>
</hibernate-mapping>
```

Gösterilen `Car` nesnesinde tekil anahtar, `licensePlate` öğesidir, onun eşlendiği kolon ise `license_plate` kolonudur. `<id name="licensePlate"..>` komutunu kullanarak, Hibernate'e `Car` nesnesinin tekilliğini `licensePlate` öğesi olduğunu bildirmiş olduk. Bu öğe, `Car` nesnesini veri tabanından yüklerken `get` komutuna anahtar olarak verilecek öğe olacaktır.

```
Car car = (Car) s.get(Car.class, "34 TTD 2202");
```

2.3.1 Birden Fazla Kolon Anahtar İse

Bazen bir tablodaki asal anahtar tek kolon değil, birden fazla kolon olabilir. Bu durumda (SQL dünyasında) bir satırı tekil olarak bulabilmek için asal anahtarı oluşturan tüm anahtarları `WHERE` komutuna veririz.

Hibernate dünyasında, bir nesneyi bulmak için birden fazla öğe kimlik olarak gerektiğinde, eşleme dosyasının anahtar belirtilen bölümünde `<id>` yerine `<composite-id>` komutunu kullanmak gerekecektir. `Person` örnek class'ında bu tekniği görelim:

Liste 2.2: Person.hbm.xml

```
<class name="Person" table="person">
  <composite-id name="id" class="Person\$Id">
    <key-property name="firstName"/>
    <key-property name="lastName"/>
  </composite-id>
  ....
  </class>
```

Liste 2.3: Person.java

```
package org.mycompany.kitapdemo.pojo;

public class Person {

    public Person() { }
```

```
Id id;

public Id getId() {
    return id;
}

public void setId(Id newId) {
    this.id = newId;
}

/**
 * Inline Id class
 *
 */
public static class Id implements Serializable {

    public Id(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public Id() {}

    String firstName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String newFirstName) {
        this.firstName = newFirstName;
    }

    String lastName;

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String newLastName) {
        this.lastName = newLastName;
    }
}

// diğer öğeler ve onların get/set
// çağrıları buraya
// ...
```

```
}
```

Örneğimizde `Person` asal anahtarı, `firstName` (isim) ve `lastName` (soyisim) ikilisinden oluşmaktadır. Hibernate'e bu ikiliyi bir asal anahtar olarak vermemiz için, `firstName` ve `lastName`'i içeren `Id` isminde yeni bir **iç class** (inner class) yaratmamız gerekti. Bu class, daha sonra `composite-id` komutuna parametre olarak geçilmiştir.

> **Not**: Kod idaresi açısından, `Id` class'ını `Person` içine koyarak bir iç class hâline getirdik, ama `Id`, apayrı bir class olarak kendi `.java` dosyasında da olabilirdi. Kodun düzenli olması bakımından `Person` ile alâkalı her şeyin tek yerde olmasını istediğimiz için, bu kodlama şekilde daha faydalı bulduk.

Eğer bir `Person` nesnesini `get` ile yüklemek istiyorsak, bir iç class `Id` nesnesi yaratmalı ve nesneyi parametre olarak `get` komutuna geçmeliyiz.

```
Session session = HibernateSession.openSession();
Person.Id id = new Person.Id(‘‘ahmet’’, ‘‘yılmaz’’);
Person p = (Person) session.get(Person.class, id);
...
```

2.3.2 Anahtar Üretimi

Anahtarlar konusunda diğer önemli bir husus, anahtar "atama sorumluluğun" kimde olduğudur. Bazı uygulamalarda asal anahtar, verinin kendi içinden doğal olarak çıkar. Meselâ veri operatörleri `person` tablosuna veri eklerken, kişi isim ve soyadına gireceklerdir, ve bunu yaparken bir asal anahtarı da otomatik olarak eklemiş olacaklardır. Çünkü `person` için asal anahtar, `firstName` ve `lastName` kolonlarının bir kombinasyonudur.

Fakat bazı durumlarda asal anahtar verinin içinden çıkmayabilir. Bu şartlarda her satırı tekil olarak kimlikleyebilmek için bir asal anahtar *üretmek* gerekecektir.

Anahtar üretimi (ya da üretilmemesi gerektiğini) Hibernate'e `<id>` etiketi altında `<generator class=".."/>` komutu ile tanımlıyoruz. Eğer bir anahtar verinin içinden doğal olarak çıkıyorsa, hiçbir anahtar üretim eyleminin yapılmaması gerektiği Hibernate'e `<generator class="assigned"/>` ile bildirilir. Eğer ID'nin üretilmesi gerekiyorsa, en basit yöntem olarak `<generator class="increment"/>` kullanılır. Bu yöntem, `Integer` bazındaki bir ID öğesinin (ve kolonunun) her yeni değerini bir öncekinden bir fazla olarak hesaplar ve tabana otomatik olarak yazar. `Increment` ile tanımlanan nesne öğesinin `java.lang.Integer` olması gerektiğini bir daha vurgulayalım.

Eğer id üretimini veri tabanına bırakmak istiyorsanız, Hibernate de `<generator class="native"/>` kullanabilirsiniz. Bu seçim ile, meselâ MySQL üzerinde

auto-increment id üretim yöntemi, Oracle üzerinde ise sequence ile id üretimi yöntemi kullanılacaktır[1].

ID üretmek için diğer yöntemler de mevcuttur. Bu ek metotları Hibernate sitesi `www.hibernate.org` adresinden, ya da Hibernate yaratıcılarının kitabından [1] bulabilirsiniz.

2.4 Dört İşlem

Hibernate'i kurup, `HibernateSession` üzerinden veri tabanına erişmeyi halletiktikten sonra, artık dört işlem ile tek bir nesneyi eklemeye, okumaya, değiştirip yazmaya ve gerekirse silmeye hazırız. Bu işlemlere veri tabanı dünyasında CRUD (**CR**eate **U**pdate **D**elete) işlemleri ismi verilmektedir. Her ne kadar önceki bölümlerde Hibernate ile dört işlem kısmen anlatılmış olsa da, toplu bir özeti bu bölümde vermeye çalışacağız.

Elimizde bir kimlik değeri var ve bu kimlik ile bir nesneyi bulup hafızaya getirmek istiyorsak, bunu yapmak için `Session` üzerinden `get` komutunu kullanırız.

```
Car car = (Car) s.get(Car.class, "34 TTD 2202");
```

Bir nesneyi bulup yükledikten sonra, onun üzerinde veri değişikliklerini, nesnenin `set` metotlarını çağırarak gerçekleştirebiliriz.

```
...
car.setDescription(‘‘yeni araba’’);
```

Fakat bu değişiklikler hâla veri tabanına yansımış değildir. Bu son aşamayı da tamamlamak için, `s.saveOrUpdate` metotunu çağırmamız gerekiyor.

```
...
s.saveOrUpdate(car);
```

Değişik bir senaryo, bir nesneyi yüklemeden, yeni bir nesneyi sıfırdan yaratıp bu nesneyi veri tabanına ekleme senaryosudur. Bu senaryo için, `new Car()` kullanarak yeni nesneyi yaratıp, üzerinde aynı `saveOrUpdate` çağrısını yapmamız gerekir. `saveOrUpdate` metotu akıllı bir metottur; Kendisine verilen `car` nesnesinin içine (verisine) bakarak, bu nesnenin yeni mi, yoksa eskinin güncellenmiş hâli mi olup olmadığını anlama yeteneğine sahiptir. Böylece aynı `saveOrUpdate` çağrısı ile hem güncelleme hem ekleme işlemini yapabiliyoruz, ve bu çağrıyı yapan uygulama kodlarımızın nesnenin yeni mi eski mi olduğunu bilmesi gerekmiyor.

Bir nesneyi (ve o nesnenin eşlenmiş olduğu veri satırını) veri tabanından silmek için, `Session` üzerinde `delete` çağrısı yapılır. Bu çağrıdan önce, `get`

[1] Hibernate native tekniği kullanılınca akılda tutulması gereken önemli bir nokta, Hibernate 2.0 versiyonunda native kullanımıyla beraber unsaved-value="0" tanımının id etiketi içinde kullanılmasıdır. Hibernate 3.0 ile bu tanımın yapılması gerekli değildir; Zaten Kitabımız Hibernate 3.0 versiyonu baz alınarak yazılmıştır

ile nesnenin yüklenmiş olması gerekmektedir (tabii perde arkasında Hibernate gereken `SELECT` işlemini sonraya bırakarak ekstra `SELECT`'ten bizi kurtarır). Bu kullanım şöyledir:

```
Car car = (Car) s.get(Car.class, "34 TTD 2202");
s.delete(car);
```

Tabii bu işlemlerden önce ve sonra `HibernateSession` yardımcı nesnesi üzerinden session alıp, bir transaction başlatıp bitirmeyi hatırlamamız gerekiyor (bkz. 2.2.5). Ancak bir transaction commit edildikten sonra yaptığımız değişiklikler veri tabanına yansıyacaktır.

2.5 Nesneler Arası İlişkiler

Hibernate gibi bir Nesne-İlişkisel Eşleme (Object Relational Mapping - **ORM**) aracının belki de en önemli faydalarından biri, iki tablo arasındaki yabancı anahtarlar (foreign keys) üzerinden kurulmuş bir ilişkiyi, direk nesnesel bir ilişki şekline dönüştürebilmesidir. Bunu yapabilmek için, doğal olarak, eşleme dosyasında gerekli ayarları programcı verecektir. Fakat bu komutlar verildikten sonra Hibernate, arka planda bir tablodan ötekine, yâni bir nesneden ilişkili olduğu diğer nesneye atlamamızı sağlayacak SQL komutlarını otomatik olarak üretecektir! Programcıya, Java seviyesinde sadece `nesne.getOtherObjectList()` gibi bir çağrı yapmak kalacaktır.

İki nesne (ya da tablo) arasında, nicelik (cardinality) açısından değişik 3 tür ilişki olabilir. Tek nesne A'nın tek nesne B ile birebir, tek nesne A'nın birçok B nesnesi ile bireçok, ve birçok A nesnesinin birçok B nesnesi arasında çokaçok ilişkileri. Bu ilişki çeşitlerini ve Hibernate eşleme dosyasında nasıl temsil edileceklerini teker teker görelim.

2.5.1 Bire Bir (One to One) İlişki

Veri tabanında tek bir satırın diğer tek bir satıra işaret etmesi, bire bir türünden bir ilişkidir. Hibernate bu şekilde bir ilişkiyi, `<many-to-one>` etiketi ile destekler.

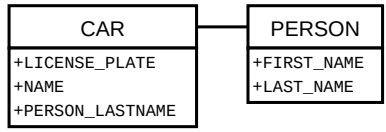
Bire bir ilişkisi, veri tabanı seviyesinde bir ya da daha fazla kolon üzerinden gerçekleştirilebilir. Eğer işaret 'edilen' tabloyu tekil olarak belirlemek için birden fazla kolon gerekiyorsa, işaret eden taraf asal anahtarı oluşturan bu kolonların hepsini kendi üzerinde taşımak zorundadır. Meselâ, asal anahtarı `name` ve `person_lastname`'den oluşan `person` tablosuna işaret eden herhangi bir POJO eşleme tanımı, şunları içermek zorundadır.

```
<many-to-one name="person" class="Person" not-null="true">
  <column name="name"/>
  <column name="person_lastname"/>
</many-to-one>
```

POJO class'ı içinde, `person` ilişkisine tekabül eden `get` ve `set` metotlarını eklemeliyiz.

```
..
Person person;

public Person getPerson() {
    return person;
}

public void setPerson(Person newPerson) {
    this.person = newPerson;
}
..
```

Bu ilişki örneğini `HibernateComposite` projesinde bulabilirsiniz.



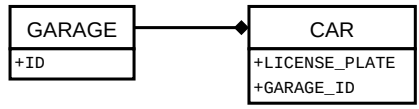
Şekil 2.2: Person Car Bire Bir İlişkisi

2.5.2 Bire Çok (One to Many) İlişki

Kurumsal uygumalarda en çok lâzım olan ilişki türü, bire çok türden ilişkidir. Örnek olarak gerçek hayattan iki nesneyi alalım: `Garage` (garaj) ve `Car` (araba). Diyelim ki bir garajda birçok araba durabilir. `Garage` ve `Car` arasındaki bu türden bir nicelik ilişkisine, bire çok türden ilişki diyoruz. Bu ilişkide, tek `Garage` nesnesi (`garage` tablosundaki tek bir satır) birçok `car` nesnesi ile (`car`'da birçok satır) arasında bir ilişki olacaktır. Veri tabanında bu ilişkiyi kurmanın yolu, bire çok ilişkinin **çok** tarafına, **bir** tarafındaki asal anahtarı alıp, yabancı anahtar olarak yerleştirmektir. Bu yapılınca, meselâ `garage.id = 1` filtresi ile, 1 no'lu `Garage` ile ilişkide olan tüm `car` kolonlarını bulmak mümkündür.

```
select * from car where garage_id = 1
```

ile ilişkideki tüm arabalar bulunabilir.



Şekil 2.3: Garage ve Car Bire Çok İlişkisi

Ayarlar

Hibernate'e bire çok türünden ilişkiyi yansıtmak için eşleme dosyasında **bir** tarafında `<set>` komutu kullanıyoruz. İçinde araba nesnelerini tutacak olan bir `java.util.Set` listesini de, `Garage` kodu üzerine yeni bir öğe olarak ekleyeceğiz. Örnek `Garage.hbm.xml` ve `Car.hbm.xml` dosyalarını altta gösteriyoruz.

Liste 2.4: Garage.hbm.xml

```
<hibernate-mapping package="org.mycompany.kitapdemo.pojo">

  <class name="Garage" table="garage">
    <id name="garageId" column="id">
      <generator class="increment"/>
    </id>
    ...
    <set name="cars"
        inverse="true"
        cascade="save-update, all-delete-orphan">
      <key>
        <column name="garage_id"/>
      </key>
      <one-to-many class="Car"/>
    </set>
  </class>
</hibernate-mapping>
```

Liste 2.5: Car.hbm.xml

```
<hibernate-mapping package="org.mycompany.kitapdemo.pojo">

  <class name="Car" table="car">
    <id name="licensePlate" column="license_plate">
      <generator class="assigned"/>
    </id>
    ...
    <many-to-one name="garage"
                column="garage_id"
     class="Garage"
     not-null="true"/>
  </class>
</hibernate-mapping>
```

Java kodları da şöyle olacaktır.

```
public class Garage {
    ...
    Set cars;
```

```
    public Set getCars() {
        return cars;
    }

    public void setCars(Set newCars) {
        this.cars = newCars;
    }
    ..
}

public class Car {
    ..
    Garage garage;

    public Garage getGarage() {
        return garage;
    }

    public void setGarage(Garage newGarage) {
        this.garage = newGarage;
    }
    ..
}
```

> **Not**: Hibernate eşleme ve Java seviyesindeki tip uyumları çok katı olarak kontrol edilir; Meselâ eşlemede `<set>` kullanıp POJO üzerinde `java.util.List` kullansaydınız (ki bu yanlış olurdu), Hibernate uygulamanız başladığında bu hatayı bulacaktır.

Artık bu eşleme üzerinden bir nesneden diğerine pür Java çağrıları kullanarak geçebiliriz.

```
Set cars = garage.getCars();
```

Cascade

`<set>` tanımı parçası olan `cascade="all-delete-orphan"`, bire çok ilişkiyi daha da güçlendirecek bir tanımdır. Cascade kelimesi İngilizce'de "olayların önce birinin, sonra ötekisi olacak şekilde, sırayla olması" anlamına gelir. `all-delete-orphan` işleminin cascade şeklinde olmasının istenmesi demek, ilişkinin **bir** tarafındaki nesne *silindiği* zaman (`delete`), bu silme işleminin `<set>` içindeki tüm nesnelere etki etmesi, yani ilişkide olan *diğer nesnelerin de silinmesi* anlamına gelir. Bu çok güçlü bir ilişkidir, ve gerçek dünya uygulamalarının veri modellerindeki her bire çok ilişki için uygun değildir. Sadece örnek amaçlı olarak `Garage` ile `Car` arasında böyle bir ilişki kurduk, yoksa, *normâl* fiziksel dünya şartlarında bir `garaj` silinince içindeki arabaların silinmesi yanlış olurdu.

`Cascade` için kullanılmış diğer seçenek `save-update`, `Garage`'da alınan listeye `garage.getCars().add(car)` ile yeni bir nesne eklenir eklenmez, bu eklemenin veri tabanına yansıtılmasını sağlar; Böylece yeni `Car` nesnesi üzerinde ayrıca bir `session.save` çağrısı yapılmasına gerek kalmaz.

Eklemek

İki nesne arasında bire çok ilişki kurulduğu zaman, eğer çok tarafındaki nesnelerden tabana bir tane daha eklemek istersek, bunu bir tarafındaki nesneden çok tarafı nesnesinin listesini alıp, o liste üzerinde ekleme yapmakla halletmemiz gerekiyor. Ayrıca, yeni eklenen nesne üzerinde, geriye, bir tarafındaki nesneye işaret eden referansa doğru referansı set etmeliyiz. Bir `Car` eklemek için alttaki örnek kodlar kullanılacaktır.

```
HibernateSession.beginTransaction();
Garage garage = (Garage) s.get(Garage.class, new Integer(1));

Car car = new Car();
car.setLicensePlate("falanfilan");
car.setDescription("falanfilan");

garage.getCars().add(car);
car.setGarage(garage);

HibernateSession.commitTransaction();
```

Fakat üzerine ekleme yaptığımız `java.util.Set`, temel Java tiplerinden biridir, o zaman üzerine ekleme yaptığımız anda Hibernate'in bu ekleme işleminden nasıl haberi olmaktadır? CGLIB sayesinde: Bir eşleme dosyası ile POJO'larımızı Hibernate'e tanıttığımızda, bu POJO'lar üstünde kullanılan `java.util.Set` kodları CGLIB tarafından değişime uğratılmaktadır, ve böylece değişmiş bu nesnelerde olan her değişimden Hibernate de haberdar olmaktadır. O zaman bu değiştirilmiş listeye yeni bir POJO eklenince, Hibernate, bu işlemi yeni bir veri satırı eklenmesi ve ilişki kolonlarının güncellenmesi isteği gibi algılayabilecektir.

2.5.3 Çoka Çok (Many to Many) İlişki

Bire çok örneğinde, garaj ve araba arasında çokluğu sadece bir tarafa gelecek şekilde koymuştuk. Bu veri tasarımına göre, bir araba *sadece bir* garaj altında olabiliyordu. Eğer bu kısıtlamayı kaldırıp, aynı arabanın fazla garaj altında olabilmesine izin vermek istiyorsak (her ne kadar fiziksel dünyaya pek uymasa da), o zaman çoka çok türünden bir ilişkiye gitmemiz gerekiyor.

Veri tasarımı açısından çoka çok ilişkilere üçüncü bir bağlantı tablosu (link table) gerekir. İlişkide olması istenilen iki tablodaki iki satırın kimlik no'ları bağlantı tablosuna yeni bir satır olarak **beraber** yazılınca, iki satır ilişkilendirilmiş olur. Bağlantı tablosu, sadece iki kimlik kolonu içeren çok basit bir tablodur.

Hibernate'e çoka çok ilişki içeren bir şemayı kabul ettirmek oldukça kolaydır, çünkü `<set>` ilişkinin kendisine de bir tablo ismi verilebilir. Bu tablo ismi, Hibernate tarafından herhangi bir nesneye değil, `<set>` ilişkisinin *kendisine ait* bir tablo olarak algılanacaktır. İstediğimiz bağlantı tablosu ismini burada verebiliriz.

```
<hibernate-mapping package="org.mycompany.kitapdemo.pojo">
  <class name="Garage" table="garage">
    ...
    <set name="cars"
         table="garage_car"
         lazy="true"
         cascade="save-update">
      <key>
        <column name="garage_id"/>
      </key>
      <many-to-many class="Car" column="license_plate"/>
    </set>
  </class>
</hibernate-mapping>
```

Tablo `garage_car` tablosu, ilişki tablosu olarak görev yapan tablodur.

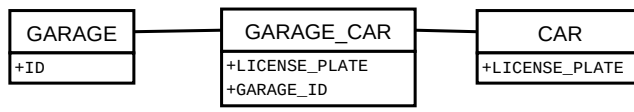
Kullanım olarak, iki nesne arasında ekstra bir ilişki tablosunun olmaması hiçbir şeyi değiştirmeyecektir (ve bu kod temizliği açısından çok iyidir). Çoka çok ilişki üzerinden **çok** tarafındaki listeyi almak, ya da çok tarafına yeni bir nesne eklemek, bireçok tekniğindeki ile aynıdır.

Çoka çok ilişki örnek kodları `HibernateManyToMany` kodları içinde bulunabilir.

2.5.4 Sıralanmış Set Almak

Eğer `Set` ile alınan araba listesini sıralı bir şekilde almak istersek, `<set>` komutunun içinde `order-by` ek komutunu kullanabiliriz. Meselâ `Garage`'dan aldığımız `Car` listesinin araba plaka kolonuna göre sıralanmasını istiyorsak, alttaki tanımı kullanabiliriz.

```
<hibernate-mapping package="org.mycompany.kitapdemo.pojo">
  <class name="Garage" table="garage">
    ...
    <set name="cars"
```



Şekil 2.4: Garage, Car ve Çoka Çok İlişkisi

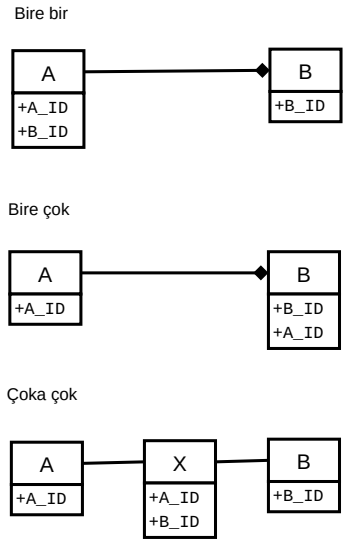
```
    ...
    order-by="license_plate asc">
    ...
   </set>
  </class>
</hibernate-mapping>
```

Liste sıralanırken, yukarıdan aşağıya doğru indikçe `license_plate`'in artmasını (a'dan z'ye doğru) istiyorsak `asc`, azalmasını istiyorsak `desc` komutunu kullanmalıyız.

2.6 Sorgular

Kurumsal uygulamamızın veriye erişim ihtiyaçları bazen tek bir nesneden başlayarak o nesnenin ilişkilerinin gezildiği türden, bazen de (istenilen sonucu almak için tüm parametrelerin elde olduğu durumda) tek sorgu ile işin bitirilebileceği türden olabilir.

Nesne gezmek yönteminde, `get` ile yüklenen tek nesneden diğer nesnelere atlanarak istenen veriye erişilmesi sağlanır. Bu tür veriye erişimi görsel arayüzü olan uygulamalarda oldukça sık olarak kullanılır, çünkü kullanıcı, her sayfada sadece belli ölçüde veriye bakabilir, ve yeni sayfalar istendikçe sonraki sayfalarda o önceki verilerle ilintili diğer verilere geçilmesi gerekir. Bu yöntemde, Hibernate üzerinden nesne dünyasına `get` yapılan bir girişten sonra bire bir, bire çok,



Şekil 2.5: İlişki Türleri ve Veri Tabanı

çoka çok ve ilişkiler gezilerek gereken verilerin alınması (kullanıcının ihtiyacına göre) halledilebilir.

Parametre bazlı bilgi toplama tekniğinde ise, istenilen veriye ulaşmak için gereken tüm parametreler elde vardır, ve bu parametreler üzerinden uygulanan bir sorgu ile sonucun direk alınabilmesi beklenir.

SQL dünyasında, parametreli direk erişimi yapmanın yolu, lazım olan tüm tabloları `JOIN` komutu ile birleştirmek, ve gereken satırları `WHERE` içindeki filtreleme tanımları ile çekip çıkartmaktır. Bu tür kullanımlarda tek bir Hibernate `get` komutu ile çetrefil filtreleme gerektiren parametreleri kullanamayacağımıza göre, Hibernate'in sorgulama tekniklerini tanımamız gerekiyor.

Hibernate, SQL'e çok benzeyen bir çeşit sorgulama dilini destekler. Ayrıca Hibernate, sürüm 3.0'dan itibaren direk SQL işletip sonuçları POJO listesi olarak döndürebilme özelliğine de sahiptir, fakat birazdan göreceğiniz gibi HQL dili Hibernate'in zaten elinde olan eşleme bilgilerini kullanarak SQL'dan çok daha esnek sorgular yapmamızı sağlamaktadır (SQL ile sorgulayıp cevapları POJO'lar olarak almayı daha sonra 2.6.5 bölümünde göreceğiz).

2.6.1 Basit Bir Sorgu

Tabandaki tüm `garage` satırlarını almamızı sağlayacak bir HQL sorgusu yazalım. Kod idaresi açısından, her sorguyu bir DAO nesnesine koymamız uygun olur. Her POJO için bir DAO yazabiliriz. `Garage` POJO'su için böyle bir class altta gösteriliyor.

Liste 2.6: GarageDAO.java

```
package org.mycompany.kitapdemo.dao;

import java.sql.*;
import org.mycompany.kitapdemo.util.*;
import org.mycompany.kitapdemo.pojo.*;
import org.mycompany.kitapdemo.service.*;
import org.hibernate.*;
import org.hibernate.cfg.*;
import java.util.*;

public class GarageDAO {

    public GarageDAO() {
        HibernateSession.beginTransaction();
    }

    public List getAll() {
        Query query = null;
        Session s = HibernateSession.openSession();
        query = s.createQuery("select garage from Garage garage");
        return query.list();
```

```
    }

}
```

Örnek sorgu olarak kullanılan `from Garage` HQL sorgusundan gelen sonuç, bir `List` nesnesi olarak `getAll`'dan geri dönülmüştür. Eğer sorgunuzdan tek bir satır sonuç geleceğinden eminseniz, `Query` nesnesinden `list()` almak yerine `uniqueResult()` çağrısını yapabilirsiniz. Bu çağrı, bir liste yerine, tek bir nesne almanızı sağlayacaktır; Böylece sonuca erişmek için `list.get(0)` gibi ek bir çağrı yapmanıza gerek kalmaz.

2.6.2 Join İçeren Sorgular

Diyelim ki, sorgudan istediğimiz daha karışık bir istek şöyle olsun; Kolon `description` üzerinde "örnek description 1" metnini içeren arabaların *garajlarının* bir listesini isteyelim. Bu sorgu için SQL dünyasında bir `JOIN` ile `garage` ve `car` tablosu birleştirilir, kartezyen birleşimi üzerinden `garage_id`'si uyanlar ve `car` tablosu üzerindeki `description` kolonunda "örnek description 1" metnini taşıyanlar filtrelenirdi.

Ama HQL dilinde kartezyen birleşimini yapmak için teker teker kolon belirtmeye gerek yoktur. Zaten elimizde `Garage` nesnesinden `Car` nesnesine gitmemizi sağlayacak bir Hibernate ilişki tanımı vardır. Bu ilişki tanımı Hibernate tarafından `garage` ve `car`'ı birleştirmek için zaten kullanılmaktaydı, ve HQL, aynı ilişki ismini hem tek *nesneden gezmek* hem de *bir sorguda birleştirim* için kullanabilir. O zaman `cars` ilişkisi üzerinden bir sorgu şöyle olur:

```
public List getGaragesForCars(String description) {
    Session s = HibernateSession.openSession();
    Query query = s.createQuery(
                              "select garage from Garage garage " +
                              "left join garage.cars car " +
                              "where car.description = :description "
                              );
    query.setString("description", description);
    return query.list();
}
```

Sorguda `join garage.cars` adlı Hibernate `<set>` ilişkisinin ismini kullanarak, Hibernate'in eşleme dosyalarından daha önce de tanıdığı iki tabloyu birleştirmiş olduk. Kural #1'e uygun olarak SQL ile kullanacağızdan daha kısa bir dil kullanıyoruz: Bu teknik iki tablo arasındaki bağlantı kuran anahtarlar birden fazla olunca daha da yararlıdır. Böyle bir durumda SQL dili tüm ilişki kolonlarının kullanılmasını gerektirecekti ve dil kullanımı iyice uzayacaktı. Kıyasla HQL için yeni kolonların hiçbir etkisi olmayacaktır, hâlâ aynı *tek* kelime ile (ilişki ismi) tabloları birleştirebiliyor oluruz.

Parametre bazlı filtrelemeyi gerçekleştirmek için, `garage.cars` ilişkisinin yeni ismi olan `car` üzerinden `description`'a erişiriz ve filtreleme şartımızı belirtiriz. En son olarak sonuçları `query.list()` ile Hibernate'den alırız.

2.6.3 Sorgular ve İsimli Parametreler

HQL dilinin JDBC ile SQL yapmaya bir üstün tarafı daha vardır, o da, sorgulara geçilen parametrelerin *isimli parametreler* (named parameter) olabilmesidir. Meselâ üstteki örnekte `description` adlı öğenin değerini ismiyle vermiş olduk, altta gördüğümüz gibi:

```
query.setString("description", "ornek description 1");
```

Kıyasla, JDBC kullanılırken bir `PreparedStatement` hazırlanır ve bu nesneye geçilen sorgu içinde parametrelerin yeri boş bırakılarak '?' karakteri ile gerçek parametre değerlerinin sonradan geleceği belirtilir. Bu parametrelerin değerleri '?' işaretinin **sırasına** göre bir numarayla belirtilir.

```
statement.setString(1,''değer'');
statement.setInt(2,200);
```

JDBC teknolojisinin bu tekniğinin kod idaresi bakımından kötü bir yaklaşım olduğu açıktır, çünkü bir süre geçtikten sonra koda bakan programcı için 1,2,3 gibi numaralar bir anlam ifade etmez (Kural #4). Bu numaraların hangi parametreye ait olduğunu anlamak için sorguya tekrar bakmak ve soru işaretlerini teker teker *saymak* gerekecektir. Daha kötü bir senaryoda, yeni programcı, sorgudaki parametrelerin sırasını bir şekilde değiştirir ve `set` komutlarını güncellemeyi unutursa, uygulamamıza birdenbire yeni bir hata (bug) eklenmiş olacaktır. Kural #1: Her zaman bakım külfeti en az olacak metotu seçiniz. Bu açıdan HQL'in parametre geçiş yöntemi tercih edilir olmaktadır.

2.6.4 Guruplama Teknikleri

Kurumsal uygulamalarda kullanılan sorgulama dillerinin olmazsa olmaz özelliği guruplama özelliğidir (SQL dilinde sıkça kullandığımız `SUM`, `AVG`, `MAX` ve `GROUP BY` gibi komutlar, bu tür guruplama fonksiyonlarıdır). HQL, aynen SQL gibi, guruplamayı desteklemektedir. Meselâ alttaki gibi bir `Person` eşlemesi olduğunu farz edelim, ve bu eşleme üzerinden bazı guruplama tekniklerini görelim.

```
<hibernate-mapping package="org.mycompany.kitapdemo.pojo">
  <import class="org.mycompany.kitapdemo.dao.Summary"/>
  <class name="Person" table="person">
    <id name="lastName" column="lastname">
      <generator class="assigned"/>
    </id>
    <property name="name" column="name"/>
    <property name="age" column="age"/>
  </class>
```

```
</hibernate-mapping>
```

Eğer tüm `PERSON.AGE` toplamlarını almak istersek

```
Session s = HibernateSession.openSession();
String hql = ‘‘select sum(person.age) from Person person’’;
Integer ageSum = (Integer)(s.createQuery(hql)).uniqueResult();
```

sorgusunu kullanabiliriz. En üst değer (maximum) bulmak için, `max` kullanılır.

```
Session s = HibernateSession.openSession();
String hql = ‘‘select max(person.age) from Person person’’;
Integer ageSum = (Integer)(s.createQuery(hql)).uniqueResult();
```

2.6.5 SQL ile Sorgulamak

Hibernate ile sorgulamak için, HQL yerine direk SQL de kullanabilirsiniz. Bizim tavsiyemiz, çoğu zaman daha güçlü ve kısa olan HQL dilini kullanmanız yönündedir. Sadece ve sadece bazı durumlarda üretilen SQL'in yerine elle SQL yazmak isterseniz, Hibernate'in `createSQLQuery` ve `addEntity` komutları ile direk SQL işletebilirsiniz.

Hibernate üzerinden SQL işletince geriye dönecek değerler JDBC `Resultset` yerine `List` içinde POJO'lar olarak alabiliriz. Bunu yapmak için SQL'e tek eklemeniz gereken, dondürülecek tablo ismi etrafında `{ }` işaretleri ve `SQLQuery` nesnesi üzerinde `addEntity` çağrısıdır. Meselâ,

```
select garage.* from Garage garage
where garage.description like ’%garage%’
```

gibi bir SQL, şuna dönüşecektir:

```
Query query = s.createSQLQuery(
                          "select {garage.*} from Garage garage " +
                          "where garage.description like ’%garage%’ "
                          )
    .addEntity("garage", Garage.class);
```

Görüldüğü gibi `garage` tablosunun tüm kolonlarını döndürmek istiyoruz, bunun için `garage.*` ibaresini kullandık. Bu normal SQL'dir. Buna ek olarak, `garage.*` etrafına `{ }` ekleyerek, geri dönen değerlerin Hibernate POJO'ları olduğunu bildirmiş oluyoruz. Ayrıca `createSQLQuery` çağrısından geri gelen `SQLQuery` nesnesi üzerinde hangi POJO'yu geri almak istediğimizi belirtmemiz gerekiyor. Bu belirtildikten sonra Hibernate, JDBC'den gelen `Resultset` üzerindeki kolonları zaten elinde olan eşleme dosyaları üzerinden dönüştürerek, biz nesnesel programcıların sevdiği bir nesne listesi olarak sonucu almamızı sağlayacaktır.

2.6.6 Değişik Nesnelerden Tek Sonuç Listesi

Şimdiye kadar gördüğümüz sorgulama tekniklerinde, sonucu sadece Hibernate tarafından eşlenmiş bir nesne ya da nesneler için almayı gördük. Fakat bazen

uygulamamızda farklı nesnelerden gelen öğeleri Hibernate'te eşlenmemiş bir birimin listesi olarak sunmamız gerekebilir. SQL ile bunu

```
SELECT t1.kolon1, t2.kolon1, t3.kolon1, ... FROM TABLO1 t1, TABLO2 t2, ...
```

şeklinde bir kullanım ile yapıyoruz. Sonuç listesinin içindeki kolonlar oradan, buradan toplanıyor. Acaba HQL, aynen SQL gibi farklı nesnelerden (tablolardan) toplanmış öğeleri (kolonları) tek bir sonuç listesi olarak sunamaz mıydı?

Evet bunu yapmak mümkündür. SQL'e benzer bir şekilde,

```
select garage.description, car.description, person.age
from Garage garage
left join garage.cars car
left join garage.cars.person person
where ...
```

gibi bir kullanım, bize bir `java.util.List` içinde `Object[]` nesneleri döndürecektir. `Object[]` içindeki her eleman, HQL `select` listesinde belirtilen öğe isimlerinden bir tanesi olacaktır.

Daha İyisi

Fakat `Object[]` üzerinden indis kullanarak erişim, hatırlaması ve bakımı zor bir yöntemdir. Bu sebeple daha idare edilir bir yöntem olarak bir "ara nesne" kullanma yöntemi tercih edilir. Geçici ara nesne kullanımında, kurucu metotu sadece bizim beklediğimiz öğeleri alan bir "ara class" yazılır. Örneğimiz için bu class ismi `Summary` olsun.

Liste 2.7: Summary.java

```
package org.mycompany.kitapdemo.dao;

public class Summary {

    public Summary(String garageDesc, String carDesc, Integer age) {
        this.garageDesc = garageDesc;
        this.carDesc = carDesc;
        this.age = age;
    }

    String garageDesc;

    public String getGarageDesc() {
        return garageDesc;
    }

    public void setGarageDesc(String newGarageDesc) {
        this.garageDesc = newGarageDesc;
    }
```

```
    String carDesc;

    public String getCarDesc() {
        return carDesc;
    }

    public void setCarDesc(String newCarDesc) {
        this.carDesc = newCarDesc;
    }

    Integer age;

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer newAge) {
        this.age = newAge;
    }
}
```

Bu ara nesneyi kullanan HQL sorgumuz şu şekilde değişecektir.

```
select new Summary(garage.description, car.description, person.age)
from Garage garage
left join garage.cars car
left join garage.cars.person person
...
```

Böylece sorgudan geri gelen `java.util.List` içinde `Object[]` yerine, `java.util.List` içinde `Summary` nesneleri alınabilecektir.

Import

`Summary` Hibernate tarafından eşlenen bir class olmadığı için, HQL'in bu class'ı *görmesi* için eşleme dosyasına *dahil edilmesi* gerekecek. Dahil etme (import) işlemini, SELECT listenizde referans edilen herhangi bir class'ın `hbm.xml` dosyasında

```
<hibernate-mapping ...
  <import class="org.mycompany.kitapdemo.dao.Summary"/>
  <class name=....>
```

gibi bir kullanım ile yapabilirsiniz. Örnek kod için `HibernateQueries` projesine bakınız.

2.7 Otomatik Şema Üretimi

Eğer geliştirme amaçlı olarak, uygulama her başladığında uygulama tarafından Hibernate eşleme dosyaları baz alınarak veri tabanında bir şema üretilsin istiyorsak, o zaman `hibernate.cfg.xml` dosyasında

```
<property name="hibernate.hbm2ddl.auto">update</property>
```

ibaresini kullanmak yeterlidir. Hibernate'in eşleme dosyalarından şema üretebilmesi şaşırtıcı gelebilir; Fakat biraz düşünülürse, Hibernate'in bu işi yapmak için elinde tüm bilgilerin bulunduğu anlaşılacaktır. Geliştirme amaçlı olarak bu üretimin uygulama her başlatıldığında yapılması bazı uygulamalar için kullanışlıdır. Fakat sonuç ortamındaki (production) tablolarını her başlangıçta kaybetmek istemeyen (ya da hangi tablo ekleme komutlarının üretildiğini kontrolden geçirmek isteyen) çoğu kurumsal uygulama için bu özellik pek yararlı olmayacaktır.

Eğer veri şemasının direk veri tabanına verilmesi yerine bir metin dosyasına yazılması daha uygun ise, Hibernate (`DROP TABLE`, `CREATE TABLE` komutlarını içeren bir script üretme yeteneğine de sahiptir. Bunun için komut satırından `SchemaExport` adlı Hibernate class'ını kullanabiliriz (class'ın kodları `hibernate3.jar` içindedir). Kullanım için;

```
java -Dhibernate.dialect=net.sf.hibernate.dialect.OracleDialect -cp \
/lib/cglib-full-2.0.2.jar;HIBERNATE_LIB/lib/odmg-3.0.jar;\
HIBERNATE_LIB/eg;HIBERNATE_LIB/lib/jboss-common.jar;\
HIBERNATE_LIB/lib/commons-collections-2.1.1.jar;\
HIBERNATE_LIB/lib/xerces-2.4.0.jar;HIBERNATE_LIB/lib/xml-apis.jar;\
c:/archive/kod/kitapkod/Struts/StrutsHibAdv/build/WEB-INF/classes;\
HIBERNATE_LIB/lib/dom4j-1.4.jar;HIBERNATE_LIB/hibernate3.jar;\
HIBERNATE_LIB/lib/log4j-1.2.8.jar;\
HIBERNATE_LIB/lib/commons-logging-1.0.4.jar \
org.hibernate.tool.hbm2ddl.SchemaExport --output=schema.sql --text \
HIBERNATE_LIB/eg/org/hibernate/auction/AuctionItem.hbm.xml \
HIBERNATE_LIB/eg/org/hibernate/auction/Bid.hbm.xml \
HIBERNATE_LIB/eg/org/hibernate/auction/User.hbm.xml
```

- Hangi dialect'i kullanıldığı `-D` parametresi ile verilmelidir.
- Classpath, `-cp` seçeneği ile tüm gereken Hibernate jar'ları içermelidir.
- Şema üretimi için baz alınacak tüm `hbm.xml` dosyaları, "en sonda" ve aralarında boşluk olacak sekilde sıralanmalıdır. Örnek kullanımda Hibernate dağıtım dizini altında yer alan örnek kodların eşleme dosyaları kullanılmıştır. Eşleme dosyalarının olduğu dizinler izafi (relative) olarak değil, kesin (full) olarak verilmelidir.
- Şema DDL çıktı dosyasını `--output` seçeneği ile belirtebiliriz.

- `--text` secenegi ile sadece dosyaya yazılmasını istediğimizi belirtebiliyoruz.

Eğer `SchemaExport` yerine `SchemaUpdate` kullanırsak, bu komut bir önceki şema ve "mevcut eşleme dosyaları üzerinden üretilecek olan şema" arasındaki farkı bularak, sadece o farklar için gerekli `ALTER TABLE` komutlarını bir metin dosyasında üretecektir. Bu dosya teknik lider tarafından kontrol edilip belki bir test şema üzerinde denendikten sonra sonuç ortamına gönderilebilir, çünkü ekleme amaçlı `ALTER` komutları mevcut diğer kolonların içindeki veriye zarar vermeyecektir.

2.8 Middlegen

Bir projenin başında, eğer projenin kullanabileceği bir veri taban şeması mevcut değil ise, bu şemayı tanımlamak projenin görevidir. Fakat bazen, bir şema zaten *mevcuttur* ve uygulamamızın mevcut şemayı kullanması gerekiyordur. Hiç sorun değil: Mevcut tablolar için Hibernate eşleme dosyalarını, ve POJO kodlarını elle yazarız ve Hibernate üzerinden şemaya erişmeye başlarız.

Fakat eğer mevcut şemada bulunan tablo sayısı çok fazla ise, her tablo için bir POJO'yu elle yazmak külfetli bir işlem olabilir. Programcıya bu konuda rahatlık sağlamak için, Middlegen adı verilen bir araç sağlanmıştır. Middlegen ile, şemanıza bağlanıp POJO kodlarını ve eşleme `hbm.xml` dosyalarını otomatik olarak ürettirebilirsiniz. Middlegen (üretim amacıyla) bir şema içinde aşağıdaki kalemlere bakar.

- Kolon, tablo isimleri
- Kolon tipleri
- Tablolar arasındaki yabancı anahtar ilişkileri (foreign key constraints)

Yâni Middlegen kod üretmek için şema "meta verisini" kullanır. Öğe tipleri için, kolon tipine en yakın Java tiplerini kullanılmaya çalışılır: Meselâ `VARCHAR` kolon için `java.lang.String` kullanmak gibi. Ayrıca Middlegen, yabancı anahtar ilişkilerini, POJO'lar arası `<many-to-one>`, `<set>` gibi ilişkilere çevirmeye uğraşacaktır.

Bu üretim yapıldıktan sonra, üretilen kodları projenize kopyalayarak, üstüne gereken eklemeleri yapıp devam edebilirsiniz.

2.8.1 Felsefi Tavsiye

Kod üretim tekniğini, eğer gerekiyorsa, sadece bir kez ve "bir başlangıç noktası oluşturması" amacı ile kullanmalıyız. Hibernate eşleme dili, herhangi bir üretim teknolojisinin üretebileceğinden çok daha güçlüdür. Hibernate eşleme yapısında

envai türden ilişkilendirme yöntemi olduğu gibi, performans amaçlı optimizasyonlar, kısım kısım yükleme (batch fetching), sonradan yükleme (lazy loading) gibi birçok ayarlar vardır. Tüm bu ayarları sadece veri tabanına bakarak üretmek neredeyse imkansızdır.

Ayrıca Middlegen ile kod üretim tavsiyemiz, Kural #5 ihlâli olarak algılanmamalıdır. POJO ve eşleme dosya üretimini sadece "bir şema mevcut olduğu zamanlar için" tavsiye ediyoruz. Ayrıca üretimin verdiği eşleme dosyaları final değildir, sadece bir *tahminden* ibarettir. Üretilen kod projeniz için uygun olmayabilir. Sadece bir başlangıç noktası oluşturması açısından önemlidir, ve en azından düz, "bir kolon bir öğe" şeklindeki eşlemeleri yapması açısından faydalı olacaktır.

2.8.2 Kullanmak

Middlegen için en son sürüm `Middlegen-Hibernate-r5` sürümüdür. Bu sürüm tarafımızdan Hibernate 3.0'a uyumlu hâle getirilerek yeni bir sürüm oluşturulmuştur. Kitap kodları (Ek A içinde anlatıldığı gibi) bulabileceğiniz bu proje, `kitap-tools-middlegen-x.x.zip` dosyasındaki `Middlegen-Hib-r5-mycompany` projesidir. Internet'te bulunan versiyon yerine, bu versiyonu kullanınız.

`Middlegen-Hib-r5-mycompany` projesi, `localhost`'ta çalışan ve bir MySQL üzerinde tutulan `cars` adlı veri tabanını kullanmak üzere hazırlanmıştır. Eğer değişik bir taban kullanmak istiyorsanız, öncelikle `build.xml` içindeki taban ayar dosyası ismini değiştirin.

```
<!DOCTYPE project [
<!ENTITY database SYSTEM "file:./config/database/mysql.xml">
]>
```

Eğer `mysql.xml` yerine `oracle.xml` koyarsanız, `config/database/oracle.xml` ayar dosyası kullanılacaktır. Tüm mevcut taban ayar dosyaları için `config/database` altına bakabilirsiniz.

Ayar dosyalarının "içinde" hangi taban, hangi kullanıcı ve şifre kullanıldığı gibi ayarlar vardır. Meselâ MySQL için taban ismini değiştirmek isterseniz, `config/database/mysql.xml` içinde bu değişiklikleri yapabilirsiniz. Alttaki ayarlar, `localhost` üzerindeki `cars` veri tabanına kullanıcı `root` ve boş şifre ile bağlanmak için hazırlanmıştır.

```
<property name="database.script.file"
          value="\${src.dir}/sql/\${name}-mysql.sql"/>
<property name="database.driver.file"
          value="\${lib.dir}/mysql-connector-java-3.0.9-stable-bin.jar"/>
<property name="database.driver.classpath"
          value="\${database.driver.file}"/>
<property name="database.driver"
          value="com.mysql.jdbc.Driver"/>
<property name="database.url"
          value="jdbc:mysql://localhost/cars"/>
```

```
<property name="database.userid"
         value="root"/>
<property name="database.password"
         value=""/>
<property name="database.schema"
         value=""/>
<property name="database.catalog"
         value=""/>
<property name="jboss.datasource.mapping"
         value="mySQL"/>
```

Üretilecek kodların gideceği dizini ve POJO'ların paket ismini belirlemek için, `build.properties` içindeki `pojo.package` değişkenine paket ismi veriniz. Örneklerimizde bu isim, `org.mycompany.kitapdemo.pojo` ismi olmuştur.

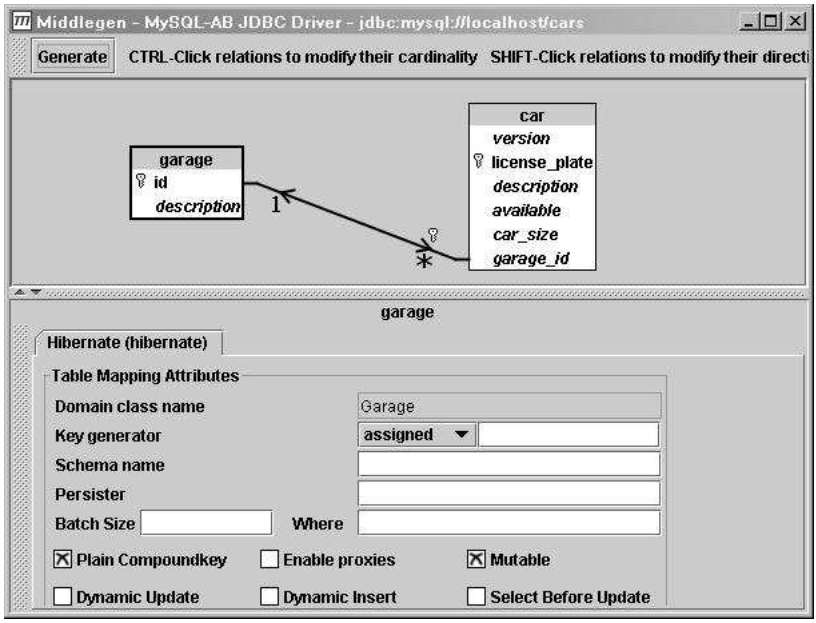
2.8.3 Test Şema

Şimdi `StrutsHibAdv` projesine girerek, `src/sql/tables_mysql.sql` dosyasını MySQL tabanınız üzerinde işletin. Bu şema içinde, yabancı anahtarlar ve kolon ilişkileri detaylı bir şekilde hazırlanmıştır, ve Middlegen bu ilişkileri takip ederek kod üretimini yapabilecektir. Bu şema, altta gösterilmiştir.

```
DROP TABLE IF EXISTS garage;
CREATE TABLE garage (
  id numeric(30) not null,
  description varchar(30) ,
  PRIMARY KEY(id),
  INDEX (id)
) TYPE = InnoDB;

DROP TABLE IF EXISTS car;
CREATE TABLE car (
  version int(10) not null,
  license_plate varchar(30) ,
  description varchar(30) ,
  available int(1) ,
  car_size varchar(1) ,
  garage_id numeric(30),
  PRIMARY KEY(license_plate),
  INDEX (license_plate),
  INDEX (garage_id),
  FOREIGN KEY (garage_id) REFERENCES garage(id)
) TYPE = InnoDB;
```

Niye `TYPE = MyISAM` yerine `TYPE = InnoDB` kullanılmıştır? Çünkü MySQL tabanlarında, kolonlararası ilişki (foreign key constraint) kurmak `MyISAM` depolama tipinde mümkün değildir. Bu konu detayları için 9.1.1 bölümüne başvurabilirsiniz.

Şema hazırlandıktan ve Middlegen ayarları tamamlandıktan sonra komut satırında (`Middlegen-Hib-r5-mycompany` dizininde) `ant` komutunu işletebilirsiniz. Bu komut sonucunda, 2.6 şeklindeki gibi bir ekran ortaya çıkacaktır.



Şekil 2.6: Middlegen Ekranı

Bu ekranda `Car` ve `Garage` nesnelerinin ve nesneler arasındaki ilişkilerin tabandaki kolonlararası ilişkiden hareketle algılanarak gösterilmiş olacaktır.

Ekrandaki her nesne, ve o nesnedeki her öğe "tıklanabilir" durumdadır. Bir tıklama ile, o nesne ya da öğe hakkındaki detayları ekranın altında görebilirsiniz. Eğer bu detaylarda beğenmediğiniz bir durum varsa, değişikliği görsel araç üzerinden yapmanız mümkündür. İki POJO arasındaki *ilişki yönünü bile* bu görsel araçtan değiştirmeniz mümkündür: `SHIFT` ve mouse'un sol düğmesine ile ilişki üzerine tıklarsanız, ilişki yön değiştirecektir.

İstediğiniz tüm değişiklikler tamamlandıktan (ya da, herşey uygun gözüküyorsa, hiçbir değişiklik yapmadan) sonra, ekranın sol üst köşesindeki "Generate" düğmesine basarak final kodu üretebilirsiniz. Üretilen kod nereye gider? Eğer `pojo.package` değişkeni için `org.mycompany.kitapdemo.pojo` tanımlandı ise, eşleme dosyaları, `build/gen-src/org/mycompany/kitapdemo/pojo` altına atılacaktır.

Son olarak, POJO Java kodlarını üretmek için

```
ant hbm2java
```

komutunu kullanın. Bu işlem, aynen eşleme dosyası üretiminde olduğu gibi `build/gen-src/org/mycompany/kitapdemo/pojo` altında java dosyalarını yaratır.

Üretim bittikten sonra bu dizin altında `Car.java` ve `Garage.java` adındaki Java dosyalarını bulabilirsiniz.

2.9 Özet

Bu bölümde, veri nesneleri POJO'lar ile veri tabanındaki tablolaları birbirine bağlayan, bize SQL'e benzer bir dil ile sorgu yapmamızı sağlayan, ve sonuçları POJO listeleri olarak dönebilen Hibernate teknolojisini işledik. Hibernate, her tablo için dört işlem olarak bilinen `INSERT`, `DELETE`, `UPDATE`, `SELECT` komutlarını üreterek bizi büyük kod yazma külfetinden kurtarmaktadır. Ayrıca, hiç ek kod yazmadan ticari ya da açık yazılım önbellek paketlerine entegre olabilmesi sayesinde, Hibernate, önbelleklemeyi programcı tarafından elle yapılan bir işlem olmaktan çıkarmaktadır. Tablolar arası her türlü ilişkinin Hibernate dünyasında nesnesel karşılıkları vardır; Veri tabanında bire bir, bire çok, çoka çok türünden tüm ilişkiler, nesne dünyasında bir nesneden diğerine sanki herşey hafızada oluyormuş gibi takip edilebilmektedir. Nesnesel programcılık dünyasında bu işleme 'nesne haritasını gezmek (traversing the object graph)' denir.

Her nesne için yapılan öğesi/kolon eşlemesi haricindeki tüm Hibernate ayarları, `hibernate.cfg.xml` adlı dosyadan verilir: Önbellek, veri tabanı bağlantı havuzunu eşleme dosyalarının yerleri programcı tarafından burada tanımlanmalıdır.

Hibernate, SQL benzeri, ama daha esnek ve güçlü olan HQL adlı bir sorgu dilini destekler. Bu sorguların sonuçları, nesne listeleri olarak Hibernate tarafından anında çevirilerek programcıya sunulur.

Middlegen, mevcut şemalardan Hibernate eşleme dosyası ve POJO kodları üretmek için kullanılır.

Son olarak, Hibernate'in işleyişini hızlandırmak, ölçeklemek için, 5.4.4, 5.4.5, 5.4.6, ve 5.4.7 bölümlerine, Hibernate kodlarımızı JUnit ile birim testlerinden geçirmek için ise, 7.1.3 bölümüne bakabiliriz.

Bölüm 3

Web Uygulamaları

Bu Bölümdekiler

- Bir Web projesinin geliştirme ve hedef ortamını kurmak için gerekenler
- Türkçe karakter desteği ve ulusallaştırma
- Struts ve JSTL Etiketleri
- Genel Web ihtiyaçlari ve çözümleri

Aynı anda birçok kullanıcıya hizmet verebilen servis tarafı Web teknolojileri, ve dinamik içerik göstermek için kullanılan Web sayfası etiket dilleri şu anki hâline gelmek için uzun bir evrimden geçti. Dinamik içerik, ilk nesilde, Perl, sh, bash, C gibi dillerle yazılmış, Apache Web Server tarafından işler kod olarak çağırılan CGI programları tarafından üretiliyordu.

Java dünyası, CGI yerine pür Java Uygulama Servisleri tarafından çağırılabilen Java Servlet'leri yerleştirdi, ve içerik göstermek için de JSP adlı etiket dilini bize sundu. Bu ilk nesilde, görsel *olmayan* işlemleri dahi Servlet yerine görsel amaçlı yazılmış JSP'lerden yapmak mümkün oluyordu, ki bu tür kullanıma Model I ismi de verilmiştir. Fakat Model I, aksiyon JSP'leri ile içerik JSP'lerini birbirinden ayırmayarak bakımı zor mimarilere sebebiyet verdiği için, bir süre sonra Model II'ye geçilmiştir.

Model II, aksiyonlar (veri tabanından bilgi almak, hata kontrolü gibi işlemler) için ayrı JSP'ler yazılmasını öngörüyordu. Fakat bu mimariden de hâlen eksik olan faktör, meselâ bir HTML form'undan bilgi almak, kullanıcının tarayıcısını başarı ya da hata sonrası bir sayfaya gönderebilmek gibi işlemler için sağlanması gereken altyapı hizmetleri idi, ve bu işlemler JSP (ya da Servlet) içinden Java kodu yazılarak programcıya zahmet getiren bir şekilde yapılması gerekiyordu.

3.1 MVC

Bunun üzerine daha önce masaüstü görsel programlar için başarıyla kullanılmış MVC mimari şekli Web dünyasına uyarlanmaya başlandı. MVC mimarileri ve ürünleri, görsel bir programı üç ana bölüme ayırıp her bölüme ayrı altyapı kodları ve hizmetleri sağlayarak bakımı kolay kodlar yazmamıza yardım eden mimarilerdir. MVC programının ana bölümleri şunlardır:

- Model
- View
- Controller

Model, hakkında program yazdığımız iş alanını hakkındaki kuralları, hesapları yaptırdığımız alan modelini temsil eden nesnelerdir. Meselâ kurumsal programınızı yazdığınız şirket (müşteriniz) araba kiralama hizmeti veren bir şirket olsaydı, bu program içinde `Car` nesnesi, *modelin* parçası olurdu.

View, görsel işlevleri yerine getiren kodlardır. Ekrana bilgi göstermek için yapılan tüm çağrıların toplamı MVC'nin V'sini (View) oluşturacaktır.

Controller, yâni kontrolcu, görsel programınızın akışını, hangi sayfadan sonra hangi sayfaya, hatalardan sonra hangi görsel birime yönlendirileceğini kontrol eden kısımdır. Bu birime yönlendirici, ya da trafik polisi ismi de verilebilirdi.

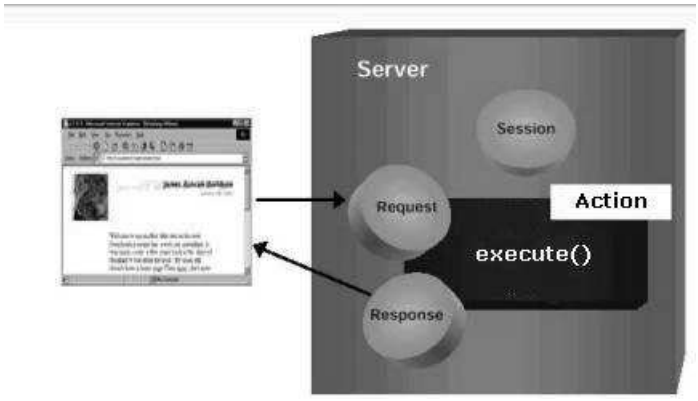
Java dünyasında MVC mimarisini destekleyen birçok ürün mecuttur. Bu ürünlerden en tanınmışı ve stabil hâlde olanı, Apache projesi tarafından sunulan Struts projesidir.

3.2 Ana Kavramlar

Görsel Web mimarilerinde çok sık lazım olan kalıplar, Apache Struts tarafından MVC altyapısı çerçevesine koyularak, Jakarta Struts projesi oluşturuldu. Modern Web uygulamalarında çok ihtiyaç duyulan kodlama gereklilikleri şunlardı:

- Giriş yapılan Web sayfalarından, yani Form'lardan, veri alıp vermeyi rahatlaştırmak.
- JSP'de bir düğme (button) basış hareketine tekabül eden bir 'işlemi', tekrar kullanılır (reusable) bir nesne içinde muhafaza etmek, ve bir Web uygulamasını artık bu işlemlerin (action) biraraya konulduğu bir bütün olarak görmek.
- Sayfalar arası akış tanımlarını (flow) Servlet'lerin, yani Java kodunun, içinden kurtarıp bir ayar dosyası içinde tutmak, böylece akışı değiştirmeyi kolaylaştırmak. Ayrıca bu sayede akış içindeki işlemleri de daha rahat tekrar kullanılır hale getirmek.

3.2.1 Form ve Action



Şekil 3.1: Request, Response ve Struts Action

Bir Struts projesi `struts-config.xml` denen ayar dosyası etrafında döner. Bu dosya, şu şekilde tanımlar içerir.

```
<form-beans>
  <form-bean
      name="AddCarForm"
      type="org.apache.struts.action.DynaActionForm">
    <form-property
        name="licensePlate"
        type="java.lang.String"
        initial=""/>
    <form-property
        name="description"
        type="java.lang.String"
        initial=""/>
  </form-bean>
</form-beans>

<action-mappings>
  <action
      path="/main"
      scope="session"
      type="org.mycompany.kitapdemo.actions.GetCarsAction">
    <forward name="fail" path="/pages/main.jsp"/>
    <forward name="success" path="/pages/main.jsp"/>
  </action>
  <action
      path="/add-car"
      scope="session"
      type="org.mycompany.kitapdemo.actions.AddCarAction"
      name="AddCarForm"
      validate="false">
    <forward name="success" path="/main.do"/>
    <forward name="fail" path="/main.do"/>
  </action>

</action-mappings>
```

İki tanım herhalde hemen gözünüze çarpmıştır. Biri `AddCarForm`, öteki `AddCarAction`. Struts kodu yazarken, ya bir JSP, ya bir Form, ya da bir Action yazıyoruzdur. Form, tahmin edileceği gibi veri girilen bir Form sayfasının üzerindeki değerleri tutan bir Java Bean nesnesidir. Üzerinde sadece get, set, reset, ya da validate gibi metotlar vardır. Struts'ın sağladığı önemli bir rahatlık, özel etiketler üzerinden sayfanızdaki alanları bu Bean'e otomatik olarak bağlamak olacaktır. Sayfadaki alan değişince bu bean'deki eşlenmiş olan değer de değişecektir. Böylece Java sunucu tarafındaki kodlarınız get/set kullanarak sayfa verilerine erişebilmiş olur. Hattâ en son Struts versiyonlarında, üzerinde get/set metotları olan ayrı bir bean'e bile ihtiyacımız kalmamıştır; Artık bu bean Struts `DynaActionForm` kullanımı sayesinde dinamik olarak üretilebilmektedir.

Action (işlem) ise, bir düğmeye bastığımızda yapılması gereken işlemi kodladığımız yerdir. O da bir Java nesnesidir, ve JSP sayfası içinde (`struts-config.xml`'de tanımlanan ismi üzerinden) cağırılması gerekir.

"Çağırılması" kelimesi tabii Struts için tam uygun bir kelime değildir. Sayfadaki düğmeler ile Action'lar, `struts-config.xml` içinde "eşlenir" demek daha uygun olur. Çünkü çâğırma işleminin kendisini Struts mimarisi kapalı kapılar arkasında yapmaktadır.

JSP sayfasında bir düğme ile Action'ı eşlemek şöyle olur.

```
<html:form action="/add-car.do">
 ...
 <td>
   <html:submit>Araba Ekle</html:submit>
 </td>
</html:form>
```

`add-car.do` bağlantısı ise, sonunda `.do` kelimesi içerdiği için bir Struts Action bağlantısıdır. Struts projenizi kurduğunuzda Web Sunucunuza (Tomcat, Weblogic, JBoss) verdiğiniz `web.xml` içerisinde, `.do` adlı soneki, Struts ana işlemci Servlet'ine bağlanması gerektiğini Web Sunucusuna belirtmemiz gerekmektedir. Proje bazında tüm Action'lar için bir kere yapılması yeterli olan bu tanımı merak edenler, örnek uygulamamızda `web.xml` içine bakabilirler; Aşağıdaki türden bir ibare göreceklerdir.

```
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    ...
  </servlet>
```

3.2.2 JSP ve Form Bağlantıları

Struts altyapısı, Form'lara otomatik değer aktarabilmek (ve daha başka görsel amaçlar için de) özel Struts etiketleri sağlar. Yâni bir Struts projesinde JSP yazarken bildiğimiz `jsp:bean` türünden etiketler değil, Struts `html:text` gibi etiketler kullanacağız. Meselâ, sayfamıza bir metin (text) alanı koyup, bu alanı, Form bean'i üzerindeki bir alana bağlamak istesek,

```
<html:text property=''licensePlate''/>
```

gibi bir ifade kullanmamız gerekir. `licensePlate` (plaka) adı ile belirtilen bu metin giriş alanı ekteki örnek kodlarda `AddCarForm` nesnesi üzerindeki `getLicensePlate`, `setLicensePlate` metotlarına eşlenecektir. Yani, Struts altyapısı

tarafında çağırılacaktır. (Tabii `DynaActionForm` kullanımı var ise, get/set içeren bean'in varlığına bile gerek yoktur, dinamik `AddCarForm` nesnesinin get/set metotları dinamik olarak çağırılacaktır).

Dikkat edelim, get ve set'in "ikisinin birden" Struts tarafından çağrılacağından / kullanılacağından bahsettim. Struts, Form bean'ine sadece tek bir zamanda değer aktarmaz. Değer aktarımı ters yön içinde geçerlidir, yani, arka planda Java üzerinden Form'da değişiklik yapıp bir sayfaya tekrar döndügünüzde, sayfadaki değerin de değiştiğini göreceksiniz.

ComboBox, Radio Button gibi görsel birimlerin ayrı ayrı `html:` kullanımı vardır.

3.2.3 Action'lardan Sonra Yönlendirme

Struts'ın üstlendiği önemli diğer bir görevde, uygulamanız için trafik polisliği görevidir. Normal Java web uygulamalarında bir Servlet'ten öteki Servlet'e aktarım `redirect` çağrısı ile yapıyorduk. Fakat bu yöntem sonraki sayfa ismini direk JSP içine gömmesi sebebiyle kod muhafazası ve bakımı açısından iyi bir yöntem değildir.

Struts mimarisi, yönlendirme tanımlarını Java kodu içerisinden çıkartarak bir ayar dosyasına koyarak değiştirilmesini rahatlatmış, ve bu şekilde Java kodlarının daha temiz ve modüler olmasını sağlamıştır.

Mesela örnek uygulamamızda `AddCarAction` işlendikten sonra diğer bir Action olan `get-cars.do`'ya gitmek istesek, bunun için Java kodu yazmamıza gerek yoktur. `struts-config.xml` içinde, `AddCarAction` tanımı altında, bu yönü belirtmemiz yeterlidir. Struts Action class'ı sadece ve sadece bir durum kodu döndürmekle yükümlüdür. Bu durum koduna bağlı olarak nereye gidileceğinin kararını `struts-config.xml` dosyası verecektir. Örnek Action tanımımızdaki

```
<action
    path="/add-car"
  ..
  <forward name="success" path="/main.do"/>
  <forward name="fail" path="/main.do"/>
```

kullanımı, `success` durum kodu döndürüldüğünde `/main.do`, hata döndürüldüğünde de yine `/main.do`'ya gidilmesini belirtmiştir (durum kodu `fail` için gidilecek yer değişik bir yer de olabilirdi. Bu tanım programcıya ait bir seçimdir. Genellikle ekleme ekranlarındaki hataları aynı sayfada gösterek iyi olduğu için bu yöntem seçilmiştir). `main.do`'nun kendisi de nihai olarak bir JSP sayfasını gösterecektir, böylece Controller (Action) aşamasından sonra View aşamasına geldiğimizi görüyoruz.

```
<action
    path="/main"
    scope="session"
```

```
    ..
  <forward name="fail" path="/pages/main.jsp"/>
  <forward name="success" path="/pages/main.jsp"/>
</action>
```

3.2.4 Action Zincirleme

Struts teknolojisinin Action ve yönlendirme yeteneği sayesinde kullanıcının tıklaması ve bir View içeren (sayfa) gelmesi arasında birkaç tane Action'ın zincirleme bir şekilde işletilmesi bile mümkündür. Modüler tasarım açısından bunu yapmak faydalı da olabilir. Struts projelerinizde filanca aksiyona ihtiyaç duyup ta "ah böyle bir Action zâten başka bir tarafından yazılmış, onu kullanayım" diyerek o mevcut Action'ı zincire kattığımız çok olmuştur. Bu, Struts'ın MVC modüler altyapısı sayesinde tekrar kullanılabilir Action'ların yazılabilmesi sayesinde olmuştur. Yeni bir Action'ı zincire katmak, Java kodlaması gerektirmediği için, `struts-config.xml` seviyesinde yapılan değişiklikler yeterli olacaktır.

Meselâ, elimizde `GetGaragesList` adında tüm garajların listesini alıp web oturumu üzerinde `garageList` değişkeniyle depolayan bir Action olsun. Bu action, normâlde `garage-list.do` adında garajları listeleyen bir sayfa `garages¬ .jsp` için kullanılmaktadır. `struts-config.xml` şöyledir.

```
<action
    path="/garage-list"
    scope="session"
    type="org.mycompany.kitapdemo.actions.GetGaragesAction"
    validate="false">
  <forward name="success" path="/pages/garages.jsp"/>
  <forward name="fail" path="/pages/garages.jsp"/>
</action>
```

Fakat diyelim ki, yeni bir sayfa yazıyoruz. Bu `newPage.jsp` adında yeni sayfa için gösterilen bir takım diğer şeylere ek olarak, bir de garaj listesini bir seçim listesi (listbox) içinde göstermek gerekiyor. Garaj listesini nereden alacağız? Yeni Action içinde bir daha garaj sorgulaması yapmak bir çözüm olabilir, fakat buna gerek yok. Daha önce yazdığımız `GetGaragesAction`'ı kullanabiliriz ve yeni Action'ımıza zincirleyebiliriz.

```
<action
    path="/new-action"
    scope="session"
    type="org.mycompany.kitapdemo.actions.NewActionX">
  <forward name="success" path="garage-list-for-x"/>
  <forward name="fail" path="/pages/error.jsp"/>
</action>

<action
    path="/garage-list-for-x"
```

```
    scope="session"
    type="org.mycompany.kitapdemo.actions.GetGaragesAction"
    validate="false">
  <forward name="success" path="/pages/newPage.jsp"/>
  <forward name="fail" path="/pages/newPage.jsp"/>
</action>
```

3.2.5 Struts ve JSTL

Apache Struts, mimari açıdan MVC'nin hem Controller, hem View tarafını gerçekleştiren bir pakettir. Controller, çünkü Struts'ın önemli bir hizmeti, aksiyon birimlerine ve yönlendirme işlemlerine bir altyapı sağlamasıdır. View, çünkü Form nesneleri ve JSP arasında kurulmuş ilintiler sayesinde View tarafı için de verilen hizmetler vardır.

Tek eksik, Struts'ın etiketlerinin çok çetrefilli sunum stilleri bazen yetersiz kalmasıdır. Mesela, bir URL'i `html:link` ile dinamik olarak oluştururken, *birden fazla* URL parametresi kullanamamaktayız.

Bu ve diğer bazı etkenler yüzünden, pür prezentasyon ihtiyaçları için kitabımızda Struts'ın Controller ve Form hizmetlerinin üstüne, pür prezentasyon amaçlı olarak JSTL adlı etiket dilini kullanmayı seçtik. JSTL, JSP Standart Etiket Kütüphanesi (JSP Standart Tag Library) kelimesinin kısaltılmışıdır. JSTL, eski JSP dilinden çok daha güçlü bir dil olarak yeni nesil etiket dillerine standart getirmeyi amaçlamıştır. Java dünyasından standartların her zaman başarılı olduğu söylenemez, fakat JSTL için ayrıca bir referans gerçekleştirimi (reference implementation) sağlandığı için ortada bir işler kod vardır, ve etiket dilinin temizliği tüm bunlara eklenince ortaya Java Web dünyası için tercih edilir bir seçenek çıkmıştır.

Bu sebeple, kitabımızdaki tüm örneklerde, ve tavsiye ettiğimiz mimari ve ürünsel yelpazede JSTL'i bulacaksınız.

3.3 Geliştirme Ortamı

Kitabımızdaki örnek Web kodlarını işletmek için JBoss Uygulama Servisi'ni kullanacağız. JBoss'ta bir web uygulamasını işleme koymak için, `JBOSS/default/deploy` dizini altına sonuç kodlarını göndermek gerekir. Fakat bu, `.class` dosyalarını olduğu gibi `deploy` dizini altına kopyalayabiliriz demek değildir. J2EE standartının mecbur kıldığı özel bir paketleme sistemini takip etmemiz gerekiyor. Bu paketleme çeşitleri şunlardır: EAR, SAR ve WAR.

En üst seviye paket EAR paketidir. EAR'in içine SAR, SAR'ın içine de WAR konur. Eğer uygulamanızda EJB bileşeni yok ise, sadece EAR ile uğraşmadan direk SAR kullanabilirsiniz. Biz, Struts odaklı tüm örnek uygulamalarımızda SAR paketlemesini kullanacağız.

Önemli bir husus, açık paket vs. kapalı paketleme konusudur. SAR, EAR ve WAR, aynen JAR yönteminde olduğu gibi sıkıştırılmış bir zip dosyası olabilirler. Biz bu yöntem yerine, sıkıştırılacak dosyaların açık bir şekilde tutulmasını seçtik. JBoss Uygulama Servisi, 3. versiyonundan itibaren, `deploy` altında gördüğü `proje.sar` dosyası ile `proje.sar` *dizinini* eşdeğer tutmaktadır. Yâni, bir SAR içine zipleyerek koyacağınız dosyaları bir SAR dizini altına kopyalarsanız, JBoss servisi uygulamanızı fark gözetmeden işleme koyacaktır.

Niye açık dizin yapısını seçtik? Çünkü sonuç (production) makinasına kod gönderirken bazı ayar dosyalarının o makina için değişmesi gerekiyor. Eğer kapalı bir SAR, EAR, WAR dosyası gönderiyor olsaydık, bu dosyaların *içinde* kalmış olan ayar dosyalarını sonradan değiştirmemiz mümkün olmazdı. Tabii ki her makina için *özel* EAR, SAR, WAR *derlemek* mümkündür, fakat sonuç ortamı için değişik derleme yapmak kesinlikle idare edilebilir bir yöntem değildir. Öncelikle her makina için değişik olacak ayar dosyalarının derleme sistemine bildirilmesi gerekmektedir. Bu derleme sistemini daha karıştıracaktır.

Bu sebeple her makina için gerekli ayar dosyalarını derleme ve kod gönderimden *sonra* EAR, SAR, WAR *içinde* değiştirmek (eskisinin üzerine yazmak), daha idare edilebilir bir yöntem olmuştur. En önemlisi, bu yöntemi takip ederek sistem admin'lerine ayar dosyalarını *elle bile* değiştirme seçeneğini sağlamış oluyoruz.

Şimdi, veri tabanına erişen kurumsal bir Struts uygulamasını geliştirmek için gereken **geliştirme** ve **hedef** dizin yapılarını görelim. Hedef Uygulama Servisimiz JBoss 4.0.1, ve hedef dizinleri JBoss'un kuruluş dizinin altında alttaki gibi olacaktır.

3.3.1 Geliştirme Dizinleri

```
+- StrutsHibSimple
| +- build.properties
| +- build.xml
| | +- dd
| | | +- jboss-service.xml
| | | +- struts-config.xml
| | | +- tiles-defs.xml
| | | +- validation.xml
| | | +- validator-rules.xml
| | | +- web.xml
| | | +- META-INF
| | | | +- MANIFEST.MF
| | | +- tags
| | | | +- c-rt.tld
| | | | +- c.tld
| | | | +- fmt-rt.tld
| | | | +- fmt.tld
| | | | +- sql-rt.tld
```

```
| | | | +- sql.tld
| | | | +- struts-bean.tld
| | | | +- struts-html.tld
| | | | +- struts-logic.tld
| | | | +- struts-nested.tld
| | | | +- struts-tiles.tld
| | | | +- x-rt.tld
| | | | +- x.tld
| | +- lib
| | | +- c3p0-0.8.4.5.jar
| | | ...
| | | ...
| | | +- cglib-full-2.0.2.jar
| | | +- commons-beanutils.jar
| | | +- hibernate3.jar
| | | +- xml-apis.jar
| | +- resources
| | | +- application.properties
| | | +- hibernate.cfg.xml
| | | +- log4j.properties
| | | +- log4j.xml
| | | +- oscache.properties
| | +- src
| | | +- java
| | | |+- org
| | | | +- mycompany
| | | | | +- kitapdemo
| | | | | | +- actions
| | | | | | | +- AddCarAction.java
| | | | | | | +- GetCarsAction.java
| | | | | +- dao
| | | | | | +- Dao.java
| | | | | +- pojo
| | | | | | +- Car.hbm.xml
| | | | | | +- Car.java
| | | | | +- service
| | | | | | +- AppStartup.java
| | | | | | +- AppStartupMBean.java
| | | | | | +- HibernateSession.java
| | | | | +- util
| | | | | | +- AllTest.java
| | | | | | +- ClassPathFile.java
| | | | | | +- RequestCharacterEncodingFilter.java
| | | | | | +- TestUtil.java
| | | +- pages
| | | | +- detail.jsp
| | | | +- main.jsp
```

```
| | | +- sql
| | | | +- tables_mysql.sql
```

Bu geliştirme dizin yapısında aranan dosyayı bulmak oldukça rahattır. En üst seviyede `src/`, `resources/`, `dd/` ve `lib/` dizinleri konulmuştur. Eğer yeni bir JBoss tanım XML'i konmak istense, bunun `dd/` altına gideceği bellidir. Aynı şekilde uygulamanın kendi içinde kullandığı ayarlar için properties ayar dosyaları ve diğer XML bazlı dosyalar `resources/` altında gene en üst seviyeden erişilir hâldedir. Kaynak dosya kategorisine giren her şey, `src/` altındadır: JSP sayfaları, Java kodları ve SQL DDL komutları gibi kalemler bu dizin altında alt dizinler olarak bulunacaktır.

3.3.2 Hedef Dizinleri

Geliştirme dizininize gidip, komut satırından `ant` komutunu işlettiğinizde, `JBOSS¬ /server/default/deploy` altında aşağıdaki dizin yapısının oluştuğunu göreceksiniz.

```
+- kitapdemo.sar
| +- META-INF
| | | +- jboss-service.xml
| | +- conf
| | | +- log4j.xml
| | +- kitapdemo.war
| | | +- pages
| | | | +- detail.jsp
| | | | +- main.jsp
| | | +- META-INF
| | | | +- MANIFEST.MF
| | | +- WEB-INF
| | | | +- jboss-service.xml
| | | | +- struts-config.xml
| | | | +- tiles-defs.xml
| | | | +- validation.xml
| | | | +- validator-rules.xml
| | | | +- web.xml
| | | | +- classes
| | | | | +- application.properties
| | | | | +- hibernate.cfg.xml
| | | | | +- log4j.properties
| | | | | +- log4j.xml
| | | | | +- oscache.properties
| | | | | +- org
| | | | | | +- mycompany
| | | | | | | +- kitapdemo
| | | | | | | | +- actions
| | | | | | | | | +- AddCarAction.class
| | | | | | | | | +- GetCarsAction.class
| | | | | | | | +- dao
```

```
| | | | | | | | | +- Dao.class
| | | | | | | | | +- pojo
| | | | | | | | | | +- Car.hbm.xml
| | | | | | | | | | +- Car.class
| | | | | | | | | +- service
| | | | | | | | | | +- AppStartup.class
| | | | | | | | | | +- AppStartupMBean.class
| | | | | | | | | | +- HibernateSession.class
| | | | | | | | | +- util
| | | | | | | | | | +- AllTest.class
| | | | | | | | | | +- ClassPathFile.class
| | | | | | | | | | +- RequestCharacterEncodingFilter.class
| | | | | | | | | | +- TestUtil.class
| | | | +- lib
| | | | | +- activation.jar
| | | | | +- ant-antlr-1.6.2.jar
| | | | | +- antlr-2.7.4.jar
| | | | | +- c3p0-0.8.4.5.jar
| | | | | +- cglib-full-2.0.2.jar
| | | | | +- commons-beanutils.jar
| | | | | +- commons-codec-1.3.jar
| | | | | ...
| | | | | ...
| | | | | +- standard.jar
| | | | | +- struts.jar
| | | | | +- xalan.jar
| | | | | +- xml-apis.jar
| | | | +- tags
| | | | | +- c.tld
| | | | | +- c-rt.tld
| | | | | +- struts-html.tld
```

Bu yapı JBoss tarafından işleme konmaya hazırdır. Yapı olarak, görüldüğü gibi WAR dizini, SAR dizini içine konmuştur.

3.3.3 Web Ayar Dosyaları

Bir SAR paketi içine konulan ayar dosyalarını teker teker tanıyalım.

jboss-service.xml

Bir SAR paketi JBoss tarafından yüklenirken o proje için yapılması gereken hazırlıklar, `jboss-service.xml` dosyasında belirlenir. JBoss bu dosyayı her zaman SAR paketinin `META-INF` dizini altında arayacaktır.

Bu dosyada, başlatılmasını istediğimiz MBean class'larını belirtmemiz gerekir (MBean'leri daha yakından JMX ile ilgili 6.3.1. bölümde tanıyacağız). Bir MBean, standart bir arayüzü (interface) gerçekleştiren (implement) bir Java

class'ıdır. Kendi yazdığımız kod olabileceği gibi JBoss, ya da diğer açık yazılım paketlerinden gelen bir MBean class'ı da olabilir.

`StrutsHibSimple` örneğinde, Log4J paketinin baslangıç işlerinin yapan MBean nesnesinin, ve demo'muzun kendi başlangıç kodlarının olduğu `AppStartup` adlı MBean'in başlatılmasımı istedik. Bunları yapmak için gereken `jboss-service.xml` şöyle olacaktır.

Liste 3.1: jboss-service.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE server>

<server>
  <!-- AppStartup adlı class başlangıç kodlarımızı taşıyor. -->
  <mbean code="org.mycompany.kitapdemo.service.AppStartup"
         name=":service=AppStartup"/>
  <!-- Log4J -->
  <mbean code="org.jboss.logging.Log4jService"
         name="jboss.system:type=KitapDemoLog4jService,
         service=KitapLogging">
    <!-- Ayar dosyasının yeri ve ismi. -->
    <attribute name="ConfigurationURL">resource:conf/log4j.xml
    </attribute>
    <attribute name="Log4jQuietMode">true</attribute>
    <!-- Ayar dosyasının ne kadar sıklıkla kontrol edileceği -->
    <attribute name="RefreshPeriod">60</attribute>
  </mbean>
</server>
```

log4j.xml

`Log4j.xml`, Log4J loglama sisteminin ayarlarını yapmak için kullanılır. Aslında, bu ayar dosya ismi isteğe göre değişebilir; Hangi ismin kullanılacağını `jboss-service.xml` içinde `resource:<dosya yeri, ismi>` üzerinden belirtmek gerekmektedir. Dosya yeri ve ismi, `kitapdemo.sar` referans alınarak aranacaktır. Örneğimizde `jboss-service.xml` içinde `conf/log4j.xml` ayarı kullanıldığı için, `kitapdemo.sar/conf/log4.xml` adlı bir dosya aranacaktır. Ayar dosyası `log4j.xml` içinde, hangi Java paketlerinin loglama yapabileceği, bu paketlerin *hangi seviyede* loglayacağı (`DEBUG`, `INFO`, vs) ve mesajların yazıldığı log dosyasının hangi dizinde olduğu gibi ayarlar belirlenir.

Demo'muzun yaptığı `log4j.xml` ayarları JBoss'un kendi **ana** Log4J ayarları ile (`server/default/conf/log4j.xml` altında) *uyumlu* çalışacak şekilde yazılmıştır. Bu kullanım şekli sayesinde, JBoss'un kendine özel diğer loglama işlemleri halâ devam edecektir, fakat bunun üstüne bizim uygulamamıza özel log'lar `JBOSS/server/default/log/kitapdemo.log` altına gider. Bu tür kullanım çoğu kurumsal uygulama'nın ihtiyacı olan bir kullanım şeklidir. Projelerde genelikle

JBoss'un kendi içinden gelen mesajlarının bilinen bir log dosyasına (`server¬ /default/log/server.log`) gönderilmesini istenir. Ek olarak kendi uygulamamızın logları ayrı bir dosyaya gitmesi beklenir.

MANIFEST.MF

Bu dosyanın basmakalıp bir içeriği vardır. Örnek kodlarda göreceğiniz içerik her proje için aynı olacaktır. Olduğu gibi kullanabilirsiniz.

web.xml

3.2.1 bölümünde gördüğümüz gibi, `web.xml`'in bir Struts projesinde önemli görevlerinden biri Struts'ı kullanmamızı sağlayan `ActionServlet` adlı merkezi class'ı JBoss'a tanıtmaktır. Buna ek olarak Servlet filtreleri `web.xml` içinde tanımlanır. Her uygulama için Servlet filtresi gerekmeyebilir. Bizim uygulamamızda JSP ve Action bazında Türkçe karakterleri destekleyebilmek için `RequestCharacterEncodingFilter` adlı filtreyi `web.xml` içinde tanımladık.

Bunların haricinde, Struts bazlı bir sistem `web.xml`'e ihtiyaç duymayacaktır. Struts uygulamalarında iş mantığı Action nesnelerinde, ve diğer tüm Web odaklı ayarlar (akış kontrolü, hata muamelesi, vs) `struts-config.xml` ayar dosyasında yapıldığı için, eski yöntem Model I ve II uygulamalarında olduğu kadar `web.xml` dosyasına ihtiyaç olmaz.

3.3.4 HibernateSessionCloseFilter

Hibernate'i kapsamlı olarak anlattığımız 2. bölümde veri tabanı ile işimiz bittiğinde bu durumu `HibernateSession.commitTransaction()` ve `HibernateSession¬ .closeSession()` belirteceğimizi söylemiştik. Fakat bu şekildeki `Session` ve `Transaction` kullanımı Web ortamında problemler doğuruyor.

Problemin çıkış noktası Struts Action'lar ve JSP sayfalarının içeriklerini göstermesinde olan işleyiş sırasıdır. Eğer elimizde `Garage` ve onun üzerinde bir `set` olarak tutulan `Car` nesneleri var ise, tipik olarak bir Struts Action ile bu listeyi alırız ve JSP ile sunum için bir `HttpSession` üzerine koyarız. Ve hemen arkasından (daha JSP bu listeyi görmeden) Hibernate transaction commit edilir ve session kapatılır. Bunun yapılma sebebi, elimizdeki son Java kodlama noktasının Action `execute` metotunun son satırı olmasıdır! JSP içine Java komutları koyamayız, çünkü View ve Controller kavramlarını birbirine karıştırmamak gerekir.

En sonunda işlem sırası JSP'ye geldiğinde ve sayfanın listeye bakması gerektiğinde, elinde daha içerikleri somutlanmamış bir `Car` listesi olacaktır. Bu listenin içeriğin erişmeye çalıştığınız anda, Hibernate `LazyInitializationException` hatasını verir [1, sf. 300], çünkü Hibernate elindeki `Car` nesnelerinin içeriğini doldurmaya uğraşmaktadır, ama elinde bunu yapacak Hibernate session yoktur.

Bu hatadan kurtulmak için, evet, bazı Hibernate ayarlarıyla oynamak suretiyle bir anda yükleme (fetching) seviyesini arttırabilir, ve `Car` listesi alındığında `Car` nesnelerinin yüklenmesini zorlayabiliriz. Ya da, listeyi alınca Action içinden elle/zorla bu listeyi *gezerek* `Car`'ların yüklenmesini zorlayabiliriz. Fakat bu iki seçimden birincisi Web dünyasının bir gerekliliğini eşleme dosyaları üzerine yansıtarak ileride yapabileceğiniz performans ayarlarlama (tuning) manevra alanınızı kısıtlar. İkinci seçenek ise, elle yazılmış, fazla ve gereksiz bir koddur (Kural #7 ihlâli).

Tavsiyemiz Web'e özel bir problemin Web'e özel ve *tek* bir yerde çözülmesidir. Servlet filtreleri burada yardımımıza yetişiyor. Altta gösterilen filtre Hibernate transaction'ı commit etme ve session'ı kapatma işini *JSP gösterimi bittikten sonra* yapar. Servlet işleyiş kurallarına göre, `chain.doFilter` çağrısı geri geldikten sonra, JSP görüntülenmesi bitmiş demektir. Bu nokta da Hibernate kapanış işlemlerini yapmak için en uygun yerdir.

Liste 3.2: HibernateCloseSessionFilter.java

```
public class HibernateCloseSessionFilter implements Filter {

    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain) throws IOException,
                                                   ServletException
    {
        try {
            chain.doFilter(request, response);
            HibernateSession.commitTransaction();
        } finally {
            HibernateSession.closeSession();
        }
    }

    public void destroy() { }
}
```

Liste 3.3: web.xml

```
<web-app>
  ...
  <filter>
    <filter-name>HibernateCloseSessionFilter</filter-name>
    <filter-class>
      org.mycompany.kitapdemo.util.HibernateCloseSessionFilter
    </filter-class>
  </filter>

  <filter-mapping>
```

```
    <filter-name>HibernateCloseSessionFilter</filter-name>
    <url-pattern>*.do</url-pattern>
  </filter-mapping>
  ...
</web-app>
```

Bu ayarlardan sonra, artık Struts Action kodlarında `commit` ve `close` işlemlerini yapmamıza gerek kalmaz. Hibernate kapanış işlemleri kaç tane Struts Action işlemiş olursa olsun JSP sayfasının görüntülenmesi bittiğinde hemen devreye sokulacaktır.

3.3.5 Hızlı Geliştirme

Kitap örneklerimizin hepsi geliştirme ortamında Ant programını kullanmaktadır (Ant hakkında ek detayları A.3 bölümünde bulabilirsiniz). Ant, aynen make gibi, bir komut dosyası kullanır; `build.xml` dosyasında derleme (compilation), test, deployment gibi geliştirme sürecinde lazım olacak tüm işlemler yazılmış hâlde bulunur. Demo'muz için hazırlanan Ant `build.xml` dosyaları birçok projeden ders alınarak hazırlanmıştır (bkz A.1 bölümü). Pek çok projede gereken hızlı geliştirme desteği, yâni sadece değişen dosyaların derlenmesi ve deploy edilmesi `build.xml`'deki `compile` ve `dist` hedefleriyle sağlanmıştır. Sadece değişen kodların derlenmesi oldukça açık olduğu için bu konu detayına girmeyeceğiz. Hızlı deployment desteği şöyledir: Eğer geliştirme sırasında sadece JSP kodu değiştirdiyseniz,

```
> ant dist
```

komutunu kullanarak sadece değişen JSP dosyalarını hedef JBoss dizinine gönderebilirsiniz. JBoss, değişen JSP dosyalarını anında işleme koyabileceği için yeni JSP dosyalarını test etmek için Uygulama Servisi'ni kapatıp/açmanıza gerek kalmaz.

Ne yazık ki aynı tekniği normâl Java kodları (`class` dosyaları) için kullanamıyoruz (ki bu durum piyasadaki tüm Uygulama Servisleri için geçerlidir). Java ClassLoader kullanımı ile alakalı bir durum yüzünden, yeni derlenmiş Java class kodlarını JBoss çalışırken sonuç dizinine göndermek, Uygulama Servisinin yeni kodları işleme koymasını sağlamaz. Yeni kodların görülmesi için, deployment sonrasında Uygulama Servisini kapatıp/açmanız gerekmektedir.

3.4 Türkçe Karakter Desteği

Daha ileri gitmeden "Web Uygulamarında Türkçe Karakter Sorunu" başlıklı sorunu işlememiz ve çözmemiz gerekiyor. Her seviyeye ve katmanı ilgilendiren bu sorunu çözer çözmez, iş mantığı odaklı Struts tekniklerine devam edebileceğiz.

İlk önce terminoloji: Literatürde internationalization gibi çok uzun bir kelime yerine genelde i18n ibaresi kullanılır, çünkü internationalization kelimesinin

ilk i ve son n harfi arasından 18 tane harf vardır; Kısaca bu kelimeye i18n denmiştir. Biz de yazının geri kalanında i18n kelimesini kullanacağız.

i18n, uygulamanızın *her seviyesinde* ayrı bir şekilde çözmeniz gereken bir sorundur. Daha basit olan `StrutsHibSimple` uygulamasından daha zor olan `StrutsHibTag` uygulamasına terfi ederken, özel olarak Türkçe karakter desteğinin genel olarak i18n probleminin hangi noktalara etki ettiğini teker teker göreceğiz. Tüm çözümü birarada `StrutsHibTag` örneğinin kodlarında bulabilirsiniz.

3.4.1 Apache

HTML belgelerinin söylediğinin aksine, HTML sayfalarımızın başında

```
<META http-equiv="Content-Type" content="text/html; charset=utf-8"/>
```

tanımı kullanmak, `utf-8` karakteri kodlaması kullanmak için yeterli olmamaktadır. Apache'nin her HTML sayfasını `utf-8` ile kodlaması için, `httpd.conf` dosyasındaki `<VirtualHost>` etiketi içine

```
AddDefaultCharset utf-8
```

satırını eklemeniz gerekmektedir.

3.4.2 Http Request

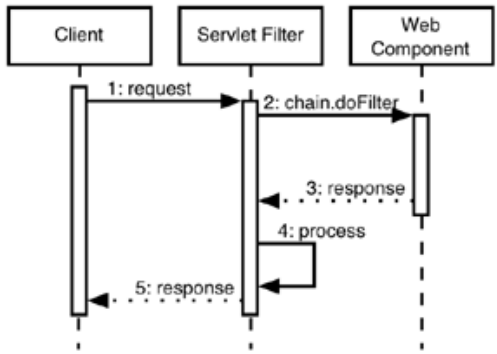
Bir Web uygulamasının Türkçe karakterleri ile çalışabilmesi için Java seviyesinde `request` nesnesi ve JSP sayfaları üzerinde karakter kodlamasını (character encoding) değiştirmemiz gerekiyor. Bu iki kodlamayı `UTF-8` bazlı yapmamız gerekmektedir.

`Request` üzerinde yapılması gereken değişikliği her Struts Action class'ı içinden yapabilirdik, fakat bu aynı kodun çok fazla tekrar etmesi demek olacaktır, ve bu Kural #7'nın ihlâli olurdu. Tekrarın her türlüsünü ortadan kaldırmak istediğimiz için, tüm `request` bazlı karakter kodlamasını tek bir Servlet filtresi ile yapabiliriz. Bu filtre `RequestCharacterEncodingFilter` adlı filtredir (`StrutsHibSimple` kodları içinde bulunabilir). Filtreyi işleme koymak için `web¬.xml`'de alttaki gibi bir değişiklik yeterli olacaktır.

```
<web-app>
  <display-name>KitapDemo</display-name>
  <filter>
    <filter-name>RequestCharacterEncodingFilter</filter-name>
    <filter-class>
      org.mycompany.kitapdemo.util.RequestCharacterEncodingFilter
    </filter-class>
    <init-param>
      <param-name>requestCharacterEncoding</param-name>
      <param-value>UTF-8</param-value>
    </init-param>
  </filter>
```

```
  <filter-mapping>
    <filter-name>RequestCharacterEncodingFilter</filter-name>
    <url-pattern>*.do</url-pattern>
  </filter-mapping>
  ....
</web-app>
```

Java Servlet standartına göre bir filtre class'ı sistemimizdeki herhangi bir Servlet (ve bilahere Struts Action) class'ı işletilmeden bile *önce* işleme konur. Bu an, istediğimiz karakter kodlaması değişikliğini yapmak için harika bir andır. Daha Servlet bile `request` nesnesine bakmadan karakter kodlama değişikliğini tek bir yerden yapabilmiş oluruz. Bir Servlet filtresi her `request` üzerinde işletilir, bu sebeple kodlama değişikliği her Servlet ve her Action için otomatik olarak yapılmış olacaktır. Böylece her Action üzerinde sürekli tekrar eden basmakalıp kodları yazmaktan kurtulduk (Kural #7).



Şekil 3.2: Servlet Filtresi

Liste 3.4: RequestCharacterEncodingFilter.java

```
import org.apache.commons.lang.StringUtils;
import org.apache.log4j.Logger;

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 java.io.IOException;

public class RequestCharacterEncodingFilter implements Filter {
```

```
    public static final String REQUEST_CHARACTER_ENCODING =
       "requestCharacterEncoding";

    private String encoding = null;

    public void init(FilterConfig filterConfig) throws ServletException
    {
        encoding = filterConfig.getInitParameter(REQUEST_CHARACTER_ENCODING);
    }

    private String getInitParameter(FilterConfig filterConfig,
                                String parameterName)
                                throws ServletException
    {
        String value = filterConfig.getInitParameter(parameterName);
        if (StringUtils.isEmpty(value)) {
             throw new ServletException(getClass().getName() +
                 ": " +
                 parameterName +
                 " is required");
        }

        return value;
    }

    public void doFilter(ServletRequest request,
                      ServletResponse response,
                      FilterChain chain) throws IOException,
                             ServletException
    {
        request.setCharacterEncoding(encoding);
        chain.doFilter(request, response);
    }

    public void destroy() { }
}
```

Filtre için gereken kodlama parametresi `web.xml` içinden (`UTF-8` değeri) `init-param` kullanılarak geçilmiştir. Değerin "kod içinden" alınmasını filtre içindeki `getInitParameter` metotu hallediyor. Parametrenin ismini aldıktan sonra

```
filterConfig.getInitParameter(‘‘requestCharacterEncoding’’)!
```

çağrısını yaparak, gereken parametre değeri okunup, `request.setCharacterEncoding` ile işleme konmaktadır.

3.4.3 JSP

JSP sayfasının Türkçe karakterleri gösterebilmesi için sayfa kodlamasının (page encoding) değişmesi gerekmektedir. Bu değişim, her sayfada yapılmalıdır. O zaman her sayfanın başına `UTF-8` kodlamasını kullanmak istediğimizi belirten bir ibare koymamız gerekiyor[1]. Bu da şöyle yapılır:

```
<META http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<%@ page language="java" contentType="text/html;
         charset=UTF-8" pageEncoding="UTF-8"%>
```

3.4.4 Struts Resources

Struts uygulamalarında JSP sayfasında dinamik olmayan sabit alan tanım değerleri (input labels) (meselâ *isim*, *soyad* gibi tarif değerleri) direk JSP içine gömülebilir. Fakat, değişik dilde müşteriye servis etmemiz gerektiği zaman bu tanım değerlerinin de dinamik bir şekilde (dilden dile) değişebilmesini isteriz. O zaman, tanım değerlerinin her dil için değişik bir dosyadan gelmesini sağlamamız gerekiyor.

Struts bu türden bir değişimi uygulama seviyesindeki *ayar dosyasının seçimini* dile bağlı olarak yapabilmekle destekler. `StrutsHibSimple` örneğinde sonuç dizini `WEB-INF\classes` altında `application.resources` adlı dosyada, uygulamamız için gerekli hata mesajlarını koymuştuk. Aslında bu dosya içine *istediğimiz her tanımı* koyabileceğimiz bir yerdir. Gerekli tanım, normâlde JSP içine gömülen alan tanım değerleri olabilir. Bu yapılınca bu dosya içindeki tanımlara JSP içinden `bean:message` Struts etiketi ile erişebiliriz. Daha önce

```
<tr>
  <td>
    License Plate
  </td>
  <td>
    <html:text property="licensePlate" size="32" />
  </td>
</tr>
<tr>
```

gibi gömülmüş değer kullanmak yerine, artık

```
<tr>
  <td>
    <bean:message key="main.license.plate"/>
  </td>
  <td>
    <html:text property="licensePlate" size="32" />
  </td>
```

[1] Bulut F. Ersavaş, `http://www.teknoturk.org/docking/yazilar/tt000144-yazi.htm`.

```
</tr>
<tr>
```

kullanabiliriz. JSP kodu, bu sayede, hem İngilizce hem Türkçe için aynı kalır.

Bu yapıldıktan sonra,tanımların değişik bir dilde (meselâ Türkçe'de) değişik çıkması için `classes` dizini altına `application_tr.properties` adında yeni bir dosya koyarız. Bu dosyada, `application.properties`'deki her label için yeni dildeki karşılıklar konacaktır.

Artık tarayıcımızın dil seçimini değiştirdiğimiz anda, meselâ `en` English'den `tr` Türkçe'ye geçtiğimiz zaman, Struts otomatik bir şekilde `application.¬ properties` yerine `application_tr.properties` dosyasını seçmesi gerektiğini bilecektir.

Örnek olarak `main.license.plate` tanımını alalım. İngilizce ve Türkçe için iki ayrı dosya şöyle gözükecektir.

Liste 3.5: application.properties

```
main.license.plate=License Plate
```

Liste 3.6: application_tr.properties

```
main.license.plate=Plaka
```

Tarayıcınızda gerekli değişikliği yapıp, tarayıcıyı açıp kapattığınızda ve JSP'yi tekrar yüklediğinizde, İngilizce yerine Türkçe mesajların çıktığını göreceksiniz.

Resources Dosyası İçinde Türkçe Karakterler

Resources dosyası hakkında aşılması gereken bir daha handikap vardır. `application_tr.properties` dosyası içinde ne yazık ki üğişçö harflerinden sadece ü, ç, ve ö harflerini direk kullanabiliyoruz. Diğer Türkçe karakterler Struts tarafından ekrana yanlış basılmaktadır. Bu harflerin yerine 3.1 tablosundaki kod değerlerini kullanmak zorundayız.

Tablo 3.1: Kodlar

Ş	`\u015E`
ş	`\u015f`
ü	`\u00FC`
ğ	`\u011f`
ö	`\u00F6`
Ü	`\u00DC`
Ö	`\u00D6`
ı	`\u0131`
İ	`\u0130`
ç	`\u00e7`
Ç	`\u00c7`

Bu kod değeri dönüşümlerini komut satırından `native2ascii` programını kullanarak otomatik olarak yaptırabiliriz. JDK'nizin `bin` dizini altında olan bu program (`javac` ile aynı dizin) eğer içinde türkçe karakterler olan bir resources dosyası üzerinde işletilirse 3.1 tablosundaki dönüşümleri yapıp ekrana basacaktır. Bu sonuçları herhangi bir yeni dosyaya yönlendirmeniz (pipe) komut satırından çok basittir. Örnek olarak `application_tr.properties` içindeki türkçe karakterleri dönüştürmek istersek:

```
native2ascii application_tr.properties > application_tr_new.properties
```

Yeni kodların olduğu dosya, `application_tr_new.properties` dosyasıdır.

3.4.5 Hibernate

Hibernate'in UTF-8 bazlı karakter kodlaması ile çalışabilmesi için UTF-8 kodlamasının Hibernate bağlantısı üzerinde set edilmesi gerekmektedir. Bu tanım yapıldıktan sonra Hibernate bu değeri aynen alıp JDBC veri taban bağlantısı üzerinde set edecektir.

Hibernate için gereken ekler 2.1 kod listesinde gösterilen tanımlara ek olarak `hibernate.connection` ayarı içeren alttaki iki satırın eklenmesi demektir[2].

```
<hibernate-configuration>
  <session-factory name="foo">
    ...
    <property name="hibernate.connection.useUnicode">
      true
    </property>
    <property name="hibernate.connection.characterEncoding">
      UTF-8
    </property>
    ...
  </session-factory>
</hibernate-configuration>
```

3.4.6 Veri Tabanı

MySQL

Versiyon 4.0'dan itibaren, MySQL'de Türkçe karakter depolamak için yapılması gereken özel hiçbir şey yoktur. Paketi olduğu gibi kurabilir, başlatabilir, ve eğer tariflerimizin geri kalanını takip ettiyseniz hemen kullanmaya başlayabilirsiniz.

Oracle

Testlerimizi yaptığımız Oracle 10g üzerinde, Oracle kuruluş aşamasının kendi olağan değerlerini kullanarak yarattığı veri tabanında Türkçe karakter prob-

[2] `http://www.hibernate.org/74.html`.

lemi meydana gelecektir. Oracle'da Türkçe karakter desteği için, *veri tabanını yaratırken* "character set" için `AL32UTF8` ve "national character set" için `AL¬16UTF16` kullanmanız gerekiyor. Bunlar yapıldıktan sonra, hem OCI hem de Thin JDBC sürücülerini kullanarak Oracle ile Türkçe karakter alışverişi yapabilirsiniz. Oracle kuruluşunda seçilmesi gereken karakter setini gösteren ekran görüntüsünü ve diğer kuruluş ile ilgili detayları 9.5.1 bölümünde bulabilirsiniz.

PostgreSQL

PostgreSQL ile Türkçe karakter kullanmak için hem ilk kuruluş ve veri tabanı yaratma aşamasında `UNICODE` karakter seti kullanılması PostgreSQL servisinde belirtilmelidir. Bu komutlar sırasıyla

```
> initdb -E UNICODE
..
> createdb test
```

Diğer kuruluş detayları için 9.5.2 bölümüne bakınız.

3.5 Etiketler

Şekil 3.3: Örnek Sayfa

3.5.1 Form Alâkalı Etiketler

Struts/JSTL uygulamamızda görsel dünya ile alışveriş elimizdeki görsel veriyi işlemek, ve ekrana basmak için birçok etiket türünün kullanıldığını göreceksiniz. Etiketleri ana amaçlar bağlamında iki büyük guruba ayırabiliriz: Form, yâni kullanıcıdan bilgi almaya yarayan etiketler, ve pür prezentasyon amaçlı etiketler.

Form ile bilgi alışverişinde Struts'ın kendi etiketlerini kullanacağımızı söylemiştik. Pür prezentasyon için, birkaç Struts etiketi dışında ağırlıkla JSTL etiketlerini kullanacağız.

Checkbox

JSP sayfasında checkbox (seçim kutusu) kullanımı için `DynaActionForm` üzerinde `java.lang.Boolean` tipli bir öğe tanımlamamız gerekiyor.

```
<form-beans>
  <form-bean name="AddCarForm"
            type="org.apache.struts.action.DynaActionForm">
    <form-property name="available"
        type="java.lang.Boolean" initial=""/>
    ...
</form-beans>
```

Bu öğeyi Form'a bağlamak için, JSP içinde `html:checkbox` etiketi kullanılmalı.

```
<td>
  <html:checkbox property="available"/>
</td>
```

Düz Metin Girişi

Bu tür giriş için Form üzerinde `String` alanı tanımlamak yeterlidir.

```
<form-beans>
  <form-bean name="AddCarForm"
            type="org.apache.struts.action.DynaActionForm">
    <form-property name="licensePlate"
                type="java.lang.String" initial=""/>
    ...
  </form-bean>
</form-beans>
```

JSP içinde ise, `html:text` kullanılır.

```
<td>
  <html:text property="licensePlate" size="32" />
</td>
```

Çoklu Checkbox

Birden fazla checkbox'ı gurup hâlinde işleyebilmek için Struts multibox yaklaşımını kullanabiliriz. Bu yaklaşımda her seçim için ayrı checkbox yaratmak yerine, Form nesnesi üzerinde tek tanımladığımız `String[]` dizisi içine, her seçilen checkbox için daha önceden bizim tanımladığımız bir kimlik değeri Action nesnemize gelecektir. Bu yaklaşım çoklu checkbox işlemesini büyük ölçüde rahatlatmaktadır.

Kullanım için Form üzerinde `String[]` tanımını yapmalıyız.

```
<form-beans>
  <form-bean name="AddCarForm"
            type="org.apache.struts.action.DynaActionForm">
    <form-property name="selectedItems"
        type="java.lang.String[]" initial=""/>
    ...
  </form-bean>
</form-beans>
```

JSP içinde her seçilen checkbox'ı ötekilerden ayıracak bir kimlik tanımı `<html:multibox>...</html:multibox>` içinde basılmalıdır. Örneğimizde bu kimlik, `Car` nesnesi için tekil olan `licensePlate` öğesidir.

```
<td>
  <html:multibox property="selectedItems">
    <c:out value="\${car.licensePlate}"/>
  </html:multibox>
</td>
```

Kullanıcı seçimi yaptıktan ve Form'u Struts Action'a gönderdiğinde (yâni "gönder" düğmesine basıldığında), Action seçilmiş olan checkbox'ları şu şekilde işleyecektir.

```
public ActionForward execute(ActionMapping mapping,
                         ActionForm form,
                         HttpServletRequest request,
                         HttpServletResponse response)
    throws Exception {

    DynaActionForm daf = (DynaActionForm) form;
    String selectedItems[] = (String[])daf.get("selectedItems");
    for (int i=0;i<selectedItems.length;i++) {
      ...
      // selectedItems[i], seçilmiş bir checkbox'dan gelen
      // kimlik değerini taşır
      ...
    }
}
```

Giriş Olarak Liste

Liste (dropdown box) için Form üzerinde bu listeden seçilecek değerin tipi belirlenmelidir. Bu genelikle `java.lang.String` olacaktır.

```
<form-beans>
  <form-bean name="AddCarForm"
            type="org.apache.struts.action.DynaActionForm">
    <form-property name="size"
       type="java.lang.String" initial=""/>
    ...
  </form-bean>
</form-beans>
```

Örneğimizde daha önceden belli üç araba ölçüsü (size) için üç kod tanımlamışız: Küçük için `s` (small), orta için `m` (medium) ve büyük için `l` (large). Bu kodları JSP sayfasında kullanıcı tarafından okunabilir bir tarif ile ilintilendirmemiz gerekiyor. Bunu da `html:select` ve `html:option` ile yapacağız.

```
<td>
  <html:select property="size" size="1">
    <html:option value=""/>
    <html:option value="s" key="car.size.small" />
    <html:option value="m" key="car.size.medium" />
    <html:option value="l" key="car.size.large" />
  </html:select>
</td>
```

Gördüğümüz gibi kodu `html:option` etiketinin `value` tabiri tanımlıyoruz. Fakat `key` de ne? Bu değer, yâni `key=''..''` içinde olacak değer, `application¬.properties` dosyanızdan alınacak değerin anahtarıdır. Bu değer, seçim listesinin kullanıcıya dönük olan tarafı olduğu için `html:option`'ın `application¬.properties`'den gelmesi gerekiyor (ya da `application_tr.properties`).

Liste 3.7: application.properties

```
car.size.small=Small
car.size.medium=Medium
car.size.large=Large
```

Liste 3.8: application_tr.properties

```
car.size.small=Küçük
car.size.medium=Orta
car.size.large=Büyük
```

3.5.2 Prezentasyon ve İşlem Amaçlı Etiketler

Resim Göstermek

Resim göstermek için `html:img` Struts etiketini kullanabilirsiniz. Resmin boyutlarını ayarlamak için `width` adlı değişken kullanılır. Width, sadece genişlik ayarını yapsa da boy ölçüsü genişliğe göre değişeceği için tek parametre yeterli olmuştur.

```
<html:img page="/pages/images/green.jpg" width="10"/>
```

URL Hazırlamak

Dinamik olarak URL üretmek için JSTL `c:url` etiketini kullanmamız gerekiyor. `c:url` birden fazla parametreyi ekleme yeteneğine sahiptir. Kullanımı:

```
<td>
  <a href="<c:url value="/struts-action-vesaire.do">
    <c:param name="param1" value="\${object.value1}"/>
    <c:param name="param2" value="\${object.value2}"/>
    ...
    </c:url>">
    <c:out value="\${object.description}"/>
  </a>
</td>
```

Bu url, üretildikten sonra, `http://host/struts-action-vesaire.do?param¬ 1=xxx&param2=yyy` gibi bir değeri taşıyacaktır. Url üzerinde olan xxx ve yyy değerleri, `object.value1` ve `object.value2` hangi değerleri taşıyorsa, onlar olacaktır. Değişken `object`, genelde bir `c:forEach` komutundan gelir.

Liste Gezmek

Veri tabanından gelen sonuçları göstermesi gereken bir JSP sayfasına çoğu zaman tek bir sonuç yerine, içinde birçok sonucun olduğu bir *liste* gelir. Bu listeyi JSTL kütüphanesinin bir etiketi olan `c:forEach` ile hiç etiket dilinin dışına çıkmadan gezebilir, ve görsel olarak kullanıcıya sunabiliriz.

Örnek olarak `StrutsHibTags` örneğindeki `GetCarsAction` Action'ını alalım. Bu Struts Action, Hibernate kullanarak (`CarDAO.java` üzerinden) veri tabanından bir `Car` listesi alır. Bu liste, bir `java.util.List` üzerindedir. Bu liste, aynen olduğu şekilde JSP sayfasına verilebilir. Bunu yapmak için listeyi `Session` üzerinde koymak yeterlidir.

```
CarDAO dao = new CarDAO();
request.getSession().setAttribute("carList", dao.fetchCarList());
```

Bu kod parçası Hibernate'den gelen listeyi `carList` adlı `Session` değişkeni üzerine koymuştur. Bu yapıldıktan ve kontrol JSP sayfasına geçtikten sonra listeyi pür etiket kullanarak gösterebiliriz.

```
<c:forEach var="car" items="\${sessionScope.carList}">
 <tr>
   <td>
     <c:out value="\${car.licensePlate}"/>
  </td>
  <td>
     <c:out value="\${car.description}"/>
   </td>
  </tr>
</c:forEach>
```

Bir Struts Action'ın `request.getSession().setAttribute` ile `Session` üzerine koyduğu bir değere `JSTL`'in nasıl eriştiğine dikkat ediniz: `sessionScope` adlı JSTL komutu kullanılmıştır. Bu sayede `Session` üzerine konan herhangi bir değişkene erişmek mümkün olmaktadır.

Üstte görülen `c:forEach` komutuna göre, `carList` listesi gezilecek, ve listedeki her eleman için `car` değişkenine listede sırası gelen eleman konulacaktır. Bu eleman için `c:forEach`'in altında olan tüm diğer etiket işlemleri gerçekleştirilir. Üstte bu işlemler `<tr>`, `<td>` ve `c:out` gibi etiket işlemleridir. JSTL `c:out`, `car` nesnesine eriştiğinde (`car.description` ile meselâ) o anda listede sırası gelmiş olan `Car` nesnesine bakıyor olacaktır.

Basit Değerleri Basmak

Basit tipli bir Java değişkenini akrana basmak için `c:out` etiketini kullanabilirsiniz. Meselâ elimizde `car` referansı var ise, bu referanstan erişilebilen `licensePlate` değerini `<c:out value="\${car.licensePlate}"/>` ile basabiliriz.

JSTL, `\$` işareti ile `String`, `Integer` ya da `Boolean` bazlı tüm değerleri dinamik olarak String'e çevirme yeteneğine sahiptir. Bu yetenek diğer JSTL değer okuyucu ve karşılaştırıcı etiketler için de geçerlidir. Bu açıdan JSTL, Perl ve Ruby dilleri gibi *dinamik* bir dil kategorisine girer. Java dili gibi güçlü tip kontrolleri yapmaz. Bir tipi gereken diğer bir tipe anında çevirir.

Değer Karşılaştırmaları

Basit şekilde bir `if` karşılaştırması için JSTL `<c:if>` etiketi kullanılır. Meselâ `StrutsHibTags` örneğindeki `detail.jsp` dosyamızda bu şekilde bir karşılaştırma kullandık. Eğer `car.available` `true` ise bir renk, değil ise diğer bir renk resim ekrana basılıyor.

```
<td>
  <c:if test='\${car.available == "false"}'>
    <html:img page="/pages/images/red.jpg" width="10"/>
  </c:if>
  <c:if test='\${car.available == "true"}'>
    <html:img page="/pages/images/green.jpg" width="10"/>
```

```
  </c:if>
</td>
```

Daha önce bahsedilen dinamik tip çevirebilme yeteneğini burada da kullanıldığına dikkat çekmek isteriz. `Boolean` olduğunu bildiğimiz bir öğeyi direk `true` ya da `false` `String` değerleri ile karşılaştırabilmemizi bu yeteneğe borçluyuz.

Kademeli Karsılaştırma

Java dilinde `switch...case` kullanımının JSP içinde kullanabileceğimiz karşılığı JSTL dilinin `<c:choose>...<c:when>` kullanımıdır. Mesela, bir `Car` nesnesinin üzerinde olan `Boolean` tipli değer `available`'ın doğru ya da yanlış olma şartına göre bir karşılaştırma şöyle yazılabilir.

```
<td>
  <c:choose>
    <c:when test='\${car.available == "false"}'>
      <html:img page="/pages/images/red.jpg" width="10"/>
    </c:when>
    <c:when test='\${car.available == "true"}'>
      <html:img page="/pages/images/green.jpg" width="10"/>
    </c:when>
    <c:otherwise>
      <html:img page="/pages/images/gray.jpg" width="10"/>
    </c:otherwise>
  </c:choose>
</td>
```

Eğer `car.available` `false` ise kırmızı bir jpeg gösterilecek, eğer `true` ise yeşil bir jpeg gösterilecektir. Eğer bu şartlardan hiçbiri doğru değil ise `c:otherwise` şartına düşeriz, bu durumda gri renkli bir resim ekrana basılacaktır (tabii bir `Boolean` öğenin değeri kesinlikle ya `true` ya da `false` olebileceği için, `otherwise` şartına düşmek imkansızdır).

Komutlar, Form Göndermek

Kullanıcı bir Form üzerinden giriş yaptıktan sonra girilen değerleri gönderebilmesi için "Gönder (submit)" gibi bir düğmeye ihtiyacı vardır. Bu düğmeyi kodlamak için değişik yöntemler mevcuttur.

İlk (ve tavsiye etmediğimiz) yöntem JSP başında `<html:form action=¬ ‘‘/add-car.do’’>` gibi bir Struts hedefi tanımlamak ve gönderme düğmesini `<html:submit>Ekle</html:submit>` gibi bir tanımdan ibaret bırakmaktır. Bu yöntem oldukça basit olsa da, bazı sorunlara sebebiyet verecektir. Birincisi, aynı form altında olan ama değişik Struts hedeflerine gönderme yapması gereken düğmeler yanyana olunca çıkar. Eğer bu düğmeler sayfanın çok değişik yerlerinde iseler, parça parça `<html:form>` açıp sonra hemen kapatıp yerine bir

tane daha açmak (her değişik hedef için) mümkün olabilir, fakat çoğu zaman HTML dizaynı buna izin vermemektedir.

Hep işe yarayacak *tek* bir yöntem öğrenmenin yararları ortada olduğu için tavsiyemiz, `html:form`'un belirlediği hedefi dikkate almayan, ve kendi hedefini kendi belirleyen "düğmeler" kullanmanızdır. Bunun için `html:submit` yerine, `html:link` ve JavaScript'in bir birleşimi olan aşağıdaki stili kullanacağız.

```
<html:link href=‘‘’’ styleClass=‘‘action’’
          onclick=‘‘document.AddCarForm.action=add-car.do’;
          document.AddCarForm.submit();return false;’’>
  Ekle
</html:link>
```

Kendi kodlarıza uyarlamak için bu örnekte değiştirmeniz gereken iki yer, `AddCarForm` yerine kendi formunuz, ve `add-car.do` Struts hedefi yerine kendi hedefinizdir.

Bu komut JSP sayfasında bir URL görüntüsü basar. Eğer URL görüntüsü yerine daha "düğmevari" bir görüntü istiyorsanız, `html:img` ile bir JPG düğme görüntüsü yaratıp onu kullanabilirsiniz. Fakat prensip olarak kullanılan teknik `html:link` örneğine çok benzer.

```
<td>
  <html:img page="/pages/images/add.jpg" styleClass="action"
           onclick="document.AddCarForm.action='add-car.do';
                    document.AddCarForm.submit();return false;"/>
</td>
```

Burada düğme görüntüsü `/pages/images/add.jpg` adlı dosyadan alınarak, JSP sayfasına konmuştur. Bu görüntü tıklandığında JavaScript üzerinden `onclick` ile Struts `add-car.do` hedefi çağırılacaktır.

Bu yaklaşımın tek dezavantajı düğme üzerindeki metni artık `String` bazlı değil ama bir JPG resimden aldığımız için, dil değiştirdiğimizde (İngilizce yerine Türkçe gibi), görüntü değiştirmek için *başka bir JPG* kullanmak zorunda kalmamızdır. Bunun için sayfa dilini JSP içinden alıp `<c:if>` ile değişik diller için hazırlanmış JPG görüntüleri arasından bir seçim yapabilirsiniz. Fakat bu kullanım, i18n bağlamında, String bazlı kullanım kadar temiz olmayacaktır. Tavsiyemiz `html:link`'i kullanıp URL görüntüsünün fontları ve belki alt çizgisi ile oynayarak, (meselâ iptal ederek) düğmeye benzer şekle getirmeniz, ve Struts'ın mevcut i18n desteğinden faydalanmanızdır.

Sayfa İçine Sayfa Eklemek

Eğer bazı JSP kodlama kalıpları birçok sayfa içinde tekrar ediyorsa, bu tekrar eden kodları tek bir JSP dosyası içine koymak, ve gereken yerde sadece bu tek dosyayı tek bir komutla dahil etmek kod idaresi açısından yararlı bir davranıştır. Bunun için, JSP etiketi `<%@ include file ..%>` komutunu kullanabilirsiniz. Kullanılış şekli çok basit:

```
<%@ include file="/pages/common.inc"%>
```

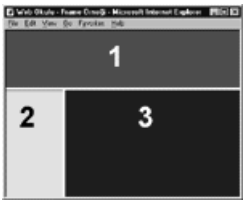
Bu örnekte `common.inc` adlı sayfa, çok tekrar eden kodları içeren merkezi sayfadır. Gereken yerden `<%@include file>` ile dahil edilmiştir. Sonek `.jsp` yerine `.inc` kullanılmasının sebebi, tek başına çalışamayacak JSP kodlarını (`.inc`) diğer sayfa kodlarından ayırmaktır.

3.6 Tiles

Tiles (kiremit, fayans) kavramı, bütün sayfaların şablonunu merkezi olarak tanımladıktan sonra, her sayfa için boşlukları doldurma suretiyle ekran yaratma tekniğidir. Her parça bir tile (kiremit) olarak görülebilir.

Bilgi işlem dünyasındaki Web sayfaların çoğunun belli bir şablonu takip ettiğini görmüşsünüzdür. En karmaşık, görsel birçok şey içeren alışveriş sitesinde bile sayfalar arasındaki benzerlikten hangi boşlukların ne zaman dolduğuna dikkat ederek bu şablonun niteliğini takip edebilirsiniz.

En basit şablon türü şöyledir: Üst bir reklam panosu, solda seçim menüsü, altta gün, ay, site kopya haklarını belirten bir ibare ve tam ortada içerik.



Şekil 3.4: Örnek Tiles Kullanımı

Bu yazımızda Struts Tiles teknolojisini kullanarak şablon, ve şablon kullanarak sayfa yaratma tekniklerini göreceğiz. Bu bölüm için yazılan kodları `StrutsTiles` dizini altında bulabilirsiniz.

3.6.1 Kurmak

İlk önce `struts-config.xml` dosyasına şu ifadeyi ekleyin.

```
<plug-in className="org.apache.struts.tiles.TilesPlugin" >
   <set-property property="definitions-config"
        value="/WEB-INF/tiles-defs.xml" />
   <set-property property="definitions-parser-validate" value="true" />
   <set-property property="moduleAware" value="true" />
</plug-in>
```

Şimdi, (eğer yoksa) `struts-config.xml` ile aynı dizin seviyesinde tiles-defs.xml adlı bir dosya yaratın. Şablonu kullanarak her sayfayı teker teker tanımladığımız

yer burası olacak. Dikkat edin, şablonu burada tanımlamıyoruz. Şablon kullanarak sayfaları hayata geçiriyoruz, ve onlara bir isim veriyoruz. `Tiles-defs¬ .xml` sözdizimi detaylarını ileride vereceğiz.

3.6.2 Şablon

Şablon için `sablon.jsp` adında bir JSP sayfası yaratalım.

```
<META http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<%@ page language="java" contentType="text/html; charset=UTF-8"
        pageEncoding="UTF-8"%>

<%@ taglib uri="/WEB-INF/tags/struts-tiles.tld" prefix="tiles" %>
<%@ taglib uri="/WEB-INF/tags/struts-html.tld" prefix="html" %>
<%@ taglib uri="/WEB-INF/tags/struts-bean.tld" prefix="bean" %>

<tiles:importAttribute name="menuSecim" scope="request"/>
...
....
<body>
  <tiles:insert attribute="ustkisim"/>
  <table cellpadding="5" cellspacing="0">
    <tr>
      <td>
        <table width="250">
          <tr width="100%">
            <td valign="top">
              <br/>
              <tiles:insert attribute="menu"/>
            </td>
          </tr>
        </table>
      </td>
      <td valign="top">
        <table>
          <tr>
            <td>
              <br/>
              <tiles:insert attribute="icerik"/>
            </td>
          </tr>
        </table>
      </td>
    </tr>
  </table>
  <tiles:insert attribute="altkisim" />
</body>
</html>
```

`tiles:insert attribute` ifadesine dikkat edelim. İşte bu ifade, şablon içinde bir "boşluk" tanımlayan ifadedir. Şablon dosyası bir sayfanın ana hatlarını belirlemekte, fakat her sayfa için değişik olacak yerlere bir isim vererek boş bırakmaktadır.

Üstteki örnekte `menu`, `icerik`, `altkisim`, `ustkisim` gibi bos yerler var. Buraya `tiles-defs.xml` dosyası her sayfa için gereken "gerçek" JSP içeriğini koyacak.

Bunun da bir örneğini görelim.

```
<definition name="ornek@ekran" extends="dizim">
    <put name="icerik" value="/sayfa123.jsp"/>
    <put name="menuSecim" value="kategori_1"/>
</definition>
```

Bu örnek sayfayı göstermek istiyorsak, `struts-config.xml` içinde bir Action üzerinden tarayıcıyı bu sayfaya yöneltmemiz lazım. Hemen yapalım:

```
<action path="/ornek_Sayfa123Goster"
    type="com.sirket.filanca.GosterAction"
    name="OrnekForm">
    <forward name="basari" path="ornek@ekran"/>
    <forward name="hata" path="/hata.jsp"/>
</action>
```

`Ornek@ekran` ifadesi, sayfamıza `tiles-defs.xml` içinde verdiğimiz "isimdir". Tarayıcının tanıdığı öteki "ismi de" `struts-config.xml`'de tanımladıktan. İşimizi bitti, artık tarayıcımızdan

```
http://localhost:8080/kitapdemo/ornek_Sayfa123Goster.do
```

yazdığımız zaman, ekranımızı görebileğiz.

3.6.3 Şablonda Olağan Değer Tanımlamak

Şablonumuzun birçok tiles tanımladığını gördük. Her sayfa için bu boş yerlere her seferinde bir değer atamaktan kurtulmak için, Tiles teknolojisi olağan değerler tanıma imkanı vermiştir. Sayfadan sayfaya fazla değişmeyen şablon değerlerini bir sözde sayfa olarak tanımlayabiliriz. Sonra, öteki "gerçek" sayfalar bu sayfadan kalıtım suretiyle bu değerleri alırlar.

Altta bunun örneğini görüyoruz. `Dizim` adı verilen sözde sayfa, fazla değişmeyen değerleri tanımlıyor, ve kalıtım yolu ile sayfa123 bu değerleri alıyor.

```
...
<definition name="dizim" path="/sablon.jsp">
    <put name="icerik" value="/bos.jsp"/>
    <put name="menuSecim" value="baslangic"/>
    <put name="ustkisim" value="/ust123.jsp"/>
    <put name="menu" value="/menu123.jsp"/>
    <put name="icerik" value="/bos.jsp"/>
```

```
    <put name="altkisim" value="/alt123.jsp"/>
</definition>
...
...
<definition name="ornek@ekran" extends="dizim">
    <put name="icerik" value="/sayfa123.jsp"/>
    <put name="menuSecim" value="kategori_1"/>
</definition>
```

JSP Sayfalara Parametre Geçmek

Tiles teknolojisinin diğer özelliği her sayfaya JSP bazında "okunabilen" bir parametreyi `tiles-defs.xml` tarafından geçilebilmesidir. Bunun faydaları, mesele, dinamik bir menü gösterirken ortaya çıkmaktadır: Bir menünün gösterdiği kategori bağlantısına ya da bir sayfa bağına tıkladığımızda `menu.jsp` tile'ının hangi kategoride olduğunu bilmesi gereklidir. Çünkü `menu.jsp` tek bir tile'dır, ama farkli sayfalara göre farklı görüntüler vermesi gerekmektedir.

Bunun çözümü sablon.jsp içinde en üsste yaptığımız

```
<tiles:importAttribute name="menuSecim" scope="request"/>
```

ifadesidir. Böylece tiles-defs'da her sayfa için tanımlayabildiğimiz

```
<put name="menuSecim" value="kategori_1"/>
```

ifadesi yetecektir. Üstteki örnek menuSecim olarak kategori_1 değerini gönderiyor. Bu değer `menu.jsp` tarafından okunarak, `logic:equal` ile test edilerek dinamik JSP göstermesi mümkün olacaktır.

3.7 Hata Mesajları İdaresi

Struts Action'larını yazarken bu nesnelerin Hibernate veya diğer alt tabaka kalıcılık (persistence) programlarından, ya da kullanıcının veri girerken yaptığı hataları yakalamak için en iyi yer olduğunu unutmamamız gerekir. Programımızın Java istisnası (exception) olarak fırlatabileceği hatalarının Action'ından yukarıya çıkmasına izin vermememiz, bu hataları yakalayıp, kullanıcıya daha anlamlı mesajlar aktarmamız gerekir. Tabii bütün Action execute islevini koskoca bir `try {} catch(Exception ..)` ile çevrelemek de fazla genelci olabilir; En iyisi ilgilendiğimiz Exception'ları ilgilendiğimiz tip seviyesinde yakalamamız ve onlara özel hatalar göstermemizdir.

Kullanıcıya güzel formatlanmış şekilde gösterilecek hatalar gerektiği zaman şu Struts kod kalıbını kullanabilirsiniz.

```
public ActionForward execute(ActionMapping mapping,
                             ActionForm form,
                             HttpServletRequest request,
                             HttpServletResponse response) {
```

```
...
ActionErrors errors=new ActionErrors();
try {

  // Hibernate ile işlemler, işlem mantığı vs..

} catch (RecordNotFoundException e) {
  errors.add(ActionMessages.GLOBAL_MESSAGE,
            new ActionError("kayit.yok.hatasi"));
  saveErrors(request,errors);
  return mapping.findForward("fail");
}
```

Bunu yaptıktan sonra ve kontrol, hata durumundan sonra JSP sayfasına geçince, hata şu ibare ile ekrana basılabilecektir.

```
<html:errors/>
```

application.properties

`kayit.yok.hatasi` adı verilen cümle, `application.properties` dosyanızın içinde tanımlanmış bir hata mesajıdır. Bu mesaj şu şekilde gözükebilir.

```
kayit.yok.hatasi=Kay\\u0131t Bulunamad\\u0131
```

Hata mesajında içinde Türkçe karakterler olduğu için, 3.4.4 bölümünde öğrendiğimiz üzere, esas karakter yerine onun ascii kodlaması kullanılmıştır.

3.7.1 Parametre Gerektiren Hata Mesajları

Statik (değişmeyen) hata mesajları, her durumda yeterli olmayabilir. Mesela eğer veri tabanında bir ürün bulamamışsak, hangi ürünün bulunamadığını kullanıcıya bildirmek isteyebiliriz. Bunun için, uygulamamızın `properties` dosyasında Struts `{0}` kullanımı işimize yarayacaktır.

```
urun.yok=Ürün kodu {0} veri tabanında bulunamadı.
```

Bu boş bırakılan yer içine, gerçek ürün değerini Struts Action içinden doldurabilirsiniz.

```
ActionErrors errors=new ActionErrors();
...
try {
  ...
} catch (UrunBulunamadiException e) {
  errors.add(ActionMessages.GLOBAL_MESSAGE,
            new ActionError("urun.yok", form.getUrunKodu()));
  saveErrors(request,errors);
  return mapping.findForward("fail");
}
```

Bu örnek, veri tabanında ürün bulunamaması halinde `UrunBulunamadiException` atıldığını, ve ActionForm üzerinde `getUrunKodu` adlı bir alanın varlığını varsaymaktadır.

3.7.2 Action'larda Muamele Görmeyen Hatalar

Kodunuz içine `catch` koymanızı gerektirmeyecek bazı hatalar olabilir. Mesela kullanıcının oturum zamanı tükenmiştir (expire) ve bu durum sürekli `NullPointerException`'a sebebiyet veriyordur. Bu Exception'ın nereden ne zaman geleceği belli olmadığı gibi, bütün Action'ları sırf bu nadir hata için `try {} catch`'lere boğmak yanlış olur. Struts, kodlamadan ziyade ayarlar ile birçok işi yapmanızı sağlar. Hiçbir Action'ın yakalamamış olduğu hataları yakalamak için `struts-config.xml`'de genel bir tanım ile kullanıcıyı **bütün** bu yakalanmamış Exception'lar için belli bir sayfaya gönderebilirsiniz.

```
<struts-config>
  ...
  <global-exceptions>
    <exception
      key="global.error.message"
      type="java.lang.Exception"
      handler="org.mycompany.kitapdemo.util.ErrorHandler"/>
  </global-exceptions>
  ...
</struts-config>
```

Bu tanıma göre her Action'da muamele görmeyen Exception'lar için kullanıcıyı `ErrorHandler` adlı bir nesneye gönderiyoruz. Bu nesne bizim tanımladığımız ve `ExceptionHandler` adlı nesneyi uzatan bir nesnedir. Kodunu `StrutsHibAdv` altında bulabilirsiniz.

```
public class ErrorHandler extends ExceptionHandler {
    private static Logger logger = Logger.getLogger("appLogger");

    public ActionForward execute(Exception ex,
                                 ExceptionConfig ae,
                                 ActionMapping mapping,
                                 ActionForm formInstance,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
        throws ServletException {

        logger.error("Struts Error", ex);

        return mapping.findForward("error");
    }
}
```

Bu kod sadece yakalanan genel `Exception`'ı Log4J ile loglayıp, ve sonra `error` adı verilen bir `ActionForward` geriye dönmektedir. Siz kendi uygulamanız için, `ActionForward` dönmeden önce daha fazla "kurtarıcı işler" yapan kodlar yazabilirsiniz.

`ErrorHandler`'dan ismi `error` adında bir `ActionForward` dönüleceğine göre, bu forward'ı belirleyen bir `struts-config.xml` tanımı daha yapmamız gerekiyor. Bunu da `global-forwards` altında yapabiliriz.

```
<global-forwards type="org.apache.struts.action.ActionForward">
  <forward name="error" path="/pages/error.jsp">
  </forward>
</global-forwards>
```

Bu forward'a göre, `error` kodu `error.jsp` adlı sayfaya yönlendirilecektir. Bu sayfa üzerinde genellikle kullanıcıya sistem bakıcısına başvurması gerektiği gibi genel mesajlar verilir.

Bu tür hata muamelesinin bütün Exception'lar için devreye gireceğini biliyoruz, çünkü Java nesnesel hiyerarşisine göre, bütün Exception'lar `java.lang.Exception` adlı nesneyi uzatmak durumundadır.

`<global-exceptions>` altında `exception` etiketini aynı `struts-config.xml` içinde birden fazla kullanabilirsiniz. Burada her özel Exception için bir yakalayıcı tanımlayabilirsiniz, Struts sistemi, elinde muamele görmemiş bir Exception olunca, özelden genele doğru bütün yakalayıcıları teker teker kontrol edecektir. Hiç bir özel yakalayıcı yoksa, `Exception` için olan en genel yakalayıcı işleme konur (eğer varsa).

3.8 Web Kodlama Kalıpları

Struts temellerinin ve görsel birimlerin anlatılması tamamlandığına göre, örnek kurumsal uygulamamız olan **bu pasta** üzerinden dikey dilimler kesmeye başlayabiliriz. Pastanın dilimlerinden her biri tüm katmanlara bir kez dokunan işlevler/görevler olacaklar. Her değişik dikey dilim ile değişik bir Kurumsal Web kodlama kalıbını göstermeye uğraşacağız.

3.8.1 Tek Nesne Yükle ve Göster

Bu pasta diliminde detayı istenen bir nesnenin seçilmesi, yüklenmesi, gösterilmesi görevini yerine getireceğiz. Yükle ve göster işlevi genellikle bir JSP URL'inden nesne üzerine tıklanması ile başlar. Yüklemek ile görevli Struts Action'ı ile gereken yüklenir ve JSP ile gereken bilgiler ekrana basılır.

Üzerine tıklanan URL'in `main.jsp` sayfasındaki listede hazırlanmış olduğunu farz ediyoruz. Bu bağlantı URL parametresi olarak bir arabayı tekil olarak kimliklendirebilmemiz için gereken `licensePlate` parametresinin taşımaktadır. Eğer `main.do` hedefini yükledikten sonra tarayıcınızda `View | Page Source` seçeneğini kullansanız, aşağıdaki gibi HTML kodlarını göreceksiniz.

```
<table>
  ...
  <tr>
      ...
      <td>
        <a href="/kitapdemo/show-car-details.do?licensePlate=plate123">
          plate1:tarif123
        </a>
      </td>
      ...
   </tr>
</table>
```

URL'e bakarsanız, tarif olarak `plate1:tarif123` metnini taşıyan URL'e tıklanınca `/kitapdemo/show-car-details.do` Struts Action'ına `licensePlate` parametresi için `plate123` değerinin geçileceğini görüyoruz. Struts Action'ların nasıl tanımlandığını 3.2.3 bölümünde görmüştük.

Struts `show-car-details.do` hedefinin muamale (handle) edilmesi için bir Action yazmamız gerekiyor. Bu Action tanımı ve Java kodları (`StrutsHibTag` projesi) şöyle gözükecektir.

Liste 3.9: struts-config.xml

```
<struts-config>
  <action-mappings>
    <action
        path="/show-car-details"
        scope="session"
        type="org.mycompany.kitapdemo.actions.ShowCarDetailAction"
        validate="false">
      <forward name="success" path="/pages/detail.jsp"/>
      <forward name="fail" path="/main.do"/>
    </action>
    ...
  </action-mappings>
</struts-config>
```

Liste 3.10: ShowCarDetailAction.java

```
1 package org.mycompany.kitapdemo.actions;
2
3 public class ShowCarDetailAction extends Action
4 {
5     private static Logger logger = Logger.getLogger("appLogger");
6
7     public ActionForward execute(ActionMapping mapping,
8                                  ActionForm form,
9                                  HttpServletRequest request,
```

```
10                                  HttpServletResponse response)
11          throws Exception {
12
13          String licensePlate = request.getParameter("licensePlate");
14
15
16          Session s = HibernateSession.openSession();
17          HibernateSession.beginTransaction();
18
19          Car car = (Car)s.get(Car.class, licensePlate);
20          request.getSession().setAttribute("car", car);
21
22          return mapping.findForward("success");
23      }
24  }
```

Satır satır açıklama:

- **13**: Action'a gelen ve `HttpServletRequest` üzerinde olan `licensePlate` değişkenine erişmek için, `getParameter` çağrısı kullanılmıştır. Bu çağrı, pür servlet (Struts olmayan) ortamdan tanıdık gelebilecek bir çağrıdır. Değişkenin değeri, yine aynı ismi taşıyan bir yerel değişken üzerinde saklanır.
- **16-17**: Hibernate Session açılır ve yeni bir transaction başlatılır.
- **19**: Reguest'den gelen parametre kullanılarak, Hibernate `get` çağrısı ile `licensePlate`'e sahip olan `Car` nesnesi yüklenir.
- **22**: Başarı kodu olan `success` geri döndürülür. Buna dayanarak, `strust-config.xml`'deki ayar, işlem sırasını `detail.jsp` adlı sayfaya devam ettirecektir.

Kontrol, JSP sayfasına gelince `c:out`, `c:when`, `c:if` gibi etiketler kullanılarak erişilen `\${car.licensePlate}`, `\${car.carSize}` gibi değerler kullanıcıya sunulabilir.

3.8.2 Nesne Ekle

Bu kodlarda bir `Car` nesnesinin sisteme eklenmesi örneğini göreceğiz. Tüm kodları `StrutsHibAdv` kodları projesinde görebilirsiniz. Eklenmesi gereken `Car` formunda, arabanın plakası, tarifi gibi `StrutsHibTags` projesindeki öğelere ek olarak, arabanın elde olup olmadığı, ölçüsü gibi yeni öğeleri ekledik. Bu öğeler JSP sayfasında yeni Form `html:` etiketleri olarak ortaya çıkacaklar. Ayar dosyası `struts-config.xml` içinde aynı şekilde JSP'deki forma uygun bir Struts `DynaActionForm` nesnesi yaratmamız gerekiyor.

Liste 3.11: Form Bean'leri

```
<struts-config>
  <form-beans>
    <form-bean
        name="AddCarForm"
        type="org.apache.struts.action.DynaActionForm">
      <form-property
          name="licensePlate"
          type="java.lang.String"
          initial=""/>
      <form-property
          name="description"
          type="java.lang.String"
          initial=""/>
      <form-property
          name="available"
          type="java.lang.Boolean"
          initial=""/>
      <form-property
          name="size"
          type="java.lang.String"
          initial=""/>
      ...
    </form-bean>
  </form-beans>
  ...
  <action
      path="/add-car"
      scope="session"
      type="org.mycompany.kitapdemo.actions.AddCarAction"
      name="AddCarForm"
      validate="false">
    <forward name="success" path="/main.do"/>
    <forward name="fail" path="/main.do"/>
  </action>

</struts-config>
```

Liste 3.12: AddCarAction.java

```
1 package org.mycompany.kitapdemo.actions;
2 ...
3 public class AddCarAction extends Action
4 {
5     private static Logger logger = Logger.getLogger("appLogger");
6
7     public ActionForward execute(ActionMapping mapping,
```

```
8                               ActionForm form,
9                               HttpServletRequest request,
10                              HttpServletResponse response)
11          throws Exception {
12
13          DynaActionForm daf = (DynaActionForm) form;
14
15          Car car = new Car ();
16          BeanUtils.copyProperties(car, daf);
17
18
19          Session s = HibernateSession.openSession();
20          HibernateSession.beginTransaction();
21          s.save(car);
22
23          // form icindeki degerleri sil
24          HttpSession session = request.getSession();
25          session.removeAttribute(mapping.getAttribute());
26
27          return mapping.findForward("success");
28      }
29  }
```

Satır satır açıklama:

- **13**: Struts `execute` metotundan gelen `ActionForm`, `DynaActionForm` nesnesine çevirilir (casting).
- **16**: Struts, JSP ile Form nesnesinin içindeki değerleri birbirine eşitleyebildiği için, eldeki `DynaActionForm`, kullanıcının doldurduğu bilgilerle dolu gelmiştir. Bu bilgileri boş olarak yarattığımız `Car` nesnesine taşımak için, `BeanUtils` yardımcı nesnesindeki `copyProperties` komutunu kullandık. Bu çağrı, arka planda *dinamik olarak* metot keşfedebilen ve çağırabilen Java Reflection teknolojisini kullanarak, iki nesne arasındaki *isimleri aynı olan öğeleri* birbirine set etme özelliğine sahiptir. Struts ActionForm ve Hibernate POJO arasındaki veri taşımaları için vazgeçilmez bir yardımcı araçtır.
- **18-27**: `Car` nesnesini veri tabanına eklemek için gereken Hibernate işlemleri.
- **24-25**: ActionForm nesnesinin oturumdan (session) çıkartılması. Bunun yapılma sebebi, ekleme işlemi bittikten ve `main.do` hedefine (bilahere `main.jsp` sayfasına) tekrar gelindiğinde, formda biraz önce girdiğimiz değerlerin tekrar gözükmemesini sağlamaktır. Bildiğimiz gibi `struts-¬ config.xml`'de `AddCarForm`, session-süreli (scope) olarak tanımlanmıştı, bu yüzden, eğer session'dan özellikle atılmazsa, session ortada olduğu sürece orada kalacak, ve bir önceki ekleme işleminin değerlerini sürekli gösterecektir.

3.8.3 Tek Nesne Yükle ve Değiştir

3.8.1 bölümünde bir nesneyi `ShowCarDetailAction` ile yükleyip, arkasından Struts ile `detail.jsp` adlı sayfaya gösterim için göndermiştik. Şimdi, `detail¬ .jsp` gösterimi yerine `edit.jsp` adlı bir *güncelleme sayfasına* yönlendirme yapacağız (yâni Struts ayarları biraz değişecek), ve bu sayfa üzerinden, kullanıcı "güncelle" düğmesine tıkladığında işlemesi gereken yeni bir güncelleme aksiyonunu, `UpdateCarAction` kodlarını tanıyacağız. Fakat ilk önce, güncelleme gündeme gelince çok meydana çıkan bir sorundan bahsetmemiz gerekmektedir: Çakışan Güncelleme Sorunu.

Çakışan Güncellemeler

Çok kullanıcılı kurumsal uygulamalarda "güncelleme" sözü telâfuz edilince hemen akla gelen bir yan sorun ortaya çıkar; Çakışan Güncelleme. Bu sorun hasır altına atılabilecek bir sorun değildir, çünkü sistemin doğru çalışması bu sorunun doğru çözülebilmesine bağlıdır.

Sorunu anlayabilmek için şöyle bir senaryo hayal edin: Kullanıcı A ve B, aynı kayıt #123'e erişiyorlar. İkisi de bu kayıda önce ekranında şöyle bir bakıyor. Daha sonra, ikisi de bu kayıdı kendine göre değiştiriyor (diyelim ki bu değiştirme hakları sistemde tanınmış). Güncelleme aksiyonuna gelince, kullanıcılardan biri, "güncelle" düğmesine basıyor, öteki kullanıcı kahve molası için kalkıyor, ve aynı düğmeye 20 saniye sonra basmış oluyor. (Ya da, tamamen raslantı sonucu bir kullanıcı düğmeye ötekinden 2 milisaniye önce basıyor, sonuç aynı olacaktır). Bu durumda, düğmeye en son basan kullanıcının değişiklikleri sisteme girmiş, bir öncekinin değişiklikleri yokolmuş olacaktır.

Burada diyebilirsiniz ki "güzel, zaten en güncel veriyle çalışmak doğru değil midir?". Fakat duruma iyice bir bakmak gerekir: Tarif edilen türden bir sistem en son verinin ne olduğunu *tamamen raslantısal bir şekilde* seçmiştir, ve bu türden bir iş süreci çoğu (hatta hiçbir) şirket için kabul edilir değildir. Kullanıcı A'nın değişiklikleri, ona hiçbir haber verilmeden *yokedilmiştir* ve kullanıcı B'nin de bu önemli değişikliklerden haberi olmamıştır. Evet belki de, aynı değişikliklerin "bazılarını" B'de yapmıştır, fakat B'nin yapmadığı ama A'nın yapmış olduğu bazı güncellemeler B için lazımdır.

Çakışan güncelleme sorununu (kurumsal bir uygulama olmasa da) meselâ CVS gibi kaynak kod idare sistemleri için de geçerlidir. Ve dikkatinizi çekerim, *dosya* bazlı çalısan bu sistemlerde bile, bir önceki değişiklik, kaybedilmez.

Veri tabanı uygulamalarında çözüm kitleme tekniğini kullanmaktan geçer. Modern mimarilerde bu kitlemeyi gerçekleştirmenin iki yolu vardır: İyimser Kitleme ve Kötümser Kitleme.

Kötümser Kitleme

Kötümser kitleme bir kayıda erişen ilk kullanıcının o kayıdı kitlemesi anlamına gelir. Arkadan gelen diğer kullanıcılar aynı kayda eriştiklerinde (yâni okumak

için bile) en iyi ihtimalde bekler, en kötü ihtimalle bunu yapamayacaklarına dair bir mesaj görürler. Eğer o kaydı değiştirmek istiyorlarsa, ilk kullanıcının o kayıt ile işinin bitmesine kadar beklemeleri gerekir. İlk kullanıcının işi bitince kilit çözülür, ve diğer kullanıcılar değişen kaydı en son hâliyle görmüş olurlar. Bu noktadan sonra eğer halâ aynı kaydı değiştirmek istiyorlarsa kendileri kayda girip kitlemiş olurlar (tabii ki halâ bunu sadece bir tanesi yapabilir) güncelleme işlemlerine başlarlar. Bu tür kitlemeye "kötümser" denmesinin sebebi, her an bir başkasının aynı kayıda gireceği beklentisiyle kilidin en baştan alınması, ve öteki kullanıcıların dışarıda tutulmasıdır.

Yukarıda tarif edilen türden bir kilit Web mimarisinde request bittikten sonra bile tutulması gerekeceği için, kitlemenin tabloda ayrı bir kolon, ya da merkezi bir kilit idarecisi tarafından kontrol edilmesi gerekmektedir, çünkü veri tabanlarının iç kilit sistemleri bir veri taban bağlantısı ve o bağlantıdaki transaction kavramına yakından bağlıdır, ve request'ler arası taban bağlantısını hiçbir zaman bağlı tutmamamız gerektiğini biliyoruz. Yoksa sistemi ölçekleme sırasında problemler yaşarız.

Bu tür bir kitlemenin iş süreçleri açısından kesin gerekli olduğu yerler vardır. Fakat, kilit üzerinde bekleme merkezi bir kilit idarecisine gitme gibi basamakları içerdiği için, yüksek ölçekli kurumsal uygulamalarda tercih edilmeyen bir seçenektir. Daha iyi ölçeklenebilen ve kullanım açısından veri doğruluğunu bozmayan diğer bir seçenek, İyimser Kitleme tekniğidir.

İyimser Kitleme

Bu kitleme tekniğinde bir kayıda erişen her iki kullanıcının "güncelle" düğmesine basmasına izin verilir (kilit önceden alınmaz), fakat güncelleme işlemi veri tabanına **ikinci** erişen kullanıcı bir hata mesajı görür. Bu mesaj, ona, kayıdın o yeni bilgileri girmekteyken değiştiğini söyler, ve ikinci kullanıcı kayıdın son hâlini tekrar yükleyip, değişikliklerini tekrar yapmalıdır. Bazı önyüz dizaynlarında kaydın en son hâli ile ikinci kullanıcının değişiklikleri merge edilerek verilebilir, yâni çetrefil bir seç/beğen türünden birleştirim ekranı sağlanabilir. Envai türden seçeneklerin arasında en basit olanı bir hata mesajıdır. Arka planda, yâni veri tabanı ve uygulama servisi seviyesinde iyimser kitleme tekniğinin gerçekleştirimi birkaç türlü olabilir.

1. Uygulama servisi seviyesinde bir kayıt ekranda gösterilir gösterilmez hemen o değerler kullanıcının oturumu (`HttpSession`) üzerine kopyalanır. Daha sonra, kullanıcı güncelle düğmesine bastığında o kaydın veri tabanındaki son hâli `SELECT` ile *bir daha* alınır, ve oturum üzerindeki değerler ile `if` kullanılarak karşılaştırmaları yapılır. Eğer uyuşmazlık var ise, hata verilir.

2. Gösterilen kayıdın değerleri oturumda kopyalanır, fakat bu sefer güncelleme sırasında veri tabanına gönderilen `UPDATE` komutuna bu değerler `WHERE` altında filtre parametresi olarak verilir. Eğer o değerler tabanda çoktan değişmiş ise `UPDATE` komutu başarısız olacaktır, çünkü filtre değerleri

hiçbir kayıt bulamaz. `UPDATE`'den geri gelen satır sayısı (JDBC bunu kontrol edebiliyor) sıfır ise hata mesajı verilebilir.

3. Veri tabanındaki her tabloda `int` tipinde bir **versiyon** kolonu bulunur, ve bu kolonunu değeri her satır için sıfırdan başlar. Her `UPDATE` işleminde bu kolon bir arttırılır. Her okuma sırasında da bu versiyon tabii ki okunur, ve `UPDATE` sorgularına `WHERE` olarak eklenir. Bu yöntemin öncekine göre bir avantajı `WHERE` filtresinin gereksiz şekilde çok büyüyecek olmamasıdır. Tek bir ekstra kolon ile iş halledilir.

Hibernate altyapısı 2 ve 3. seçenekleri desteklemektedir. Hiç fazladan kod yazmadan, özel Hibernate komutlarını kullanarak bu yöntemleri kullanabiliriz. Bizim tavsiyemiz en basit ve etkili olan 3. yöntemi kullanmanızdır. Eğer mevcut bir şema ile çalışıyor ve bu şemada istediğiniz kolonlara yeni bir versiyon kolonu eklemenize izin verilmiyor ise 2. yöntemi kullanabilirsiniz. Bu yöntem için [1, sf. 174]'e danışabilirsiniz. Biz altta 3. yöntemi tarif edeceğiz. `StrutsHibAdv` kodları içinde versiyon bazlı iyimser kitleme kullanan kodları bulabilirsiniz.

Kopuk Nesneler ile Güncelleme

Kodun detayına inmeden önce son anlatmamız gereken konu "kopuk (disconnected)" bir Hibernate nesnesi ile güncelleme yapmaktır. Web uygulamalarında `Car` ya da `Garage` gibi nesneleri detay ekranında gösterdikten sonra Hibernate oturumunu kapatırız ve detay için kullandığımız nesne kopuk (disconnected) bir nesne haline gelir.

Aynı şekilde Form'dan güncellenmiş bir veri aldığımızda yeni bir `Car` nesnesini `new` ile yaratırız, ve içini form'dan gelen değerler ile doldururuz. Bu nesne de kopuk bir Hibernate nesnesi olarak kabul edilir.

Tek problem, kopuk bir nesnenin direk alınarak `saveOrUpdate` çağırılması durumunda ortaya çıkacaktır; Hibernate, o anda aynı oturum içinde aynı kimliği taşıyan başka bir `Car` nesnesini oturumda tutuyorsa yeni bir `Car` nesnesi üzerinde güncelleme isteğine `NonUniqueObjectException` hatasını verir. Bunu yapmasının sebebi Hibernate'in iç işleyişi ile alâkalıdır, her tekil nesneden oturum üzerinde bir tane tutmak, Hibernate mekanizması açısından verilmiş bir mimari karardır. Fakat Web mimarilerinde de dışarıdan gelen kopuk nesne oldukça kullanılan bir kodlama kalıbı olduğu için bu kullanıma izin vermek için, Hibernate programcıları bize `merge` adlı yeni bir metot sağlamışlardır. Eğer kopuk bir nesneyi alıp direk güncelleme yapmak istiyorsanız, `saveOrUpdate` yerine `merge` fonksiyonunu kullanmanız gerekiyor.

Evet! Artık güncelleme kodlarının kendisine ilerleyebiliriz.

Kodlar

İlk önce yapılması gereken, eski tablomuza bir "versiyon" kolonu eklemektir.

```
DROP TABLE IF EXISTS car;
CREATE TABLE car (
  version int(10),
  license_plate varchar(30) default '',
  description varchar(30) default '',
  available int(1) default 0,
  car_size varchar(1) default ''
) ;
```

`Car` POJO'su üzerinde aynı şekilde yeni bir versiyon kolonu gerekiyor.

```
public class Car {

    int version = -1;

    public int getVersion() {
        return version;
    }

    public void setVersion(int newVersion) {
        this.version = newVersion;
    }

    ...
}
```

Eşleme dosyası `Car.hbm.xml`'da, `<version>` adında özel bir etiket kullanılması gerekiyor. Bu etiket, Hibernate'e, "versiyon kontrollü" güncelleme yapması için direktif verecektir.

```
<hibernate-mapping package="org.mycompany.kitapdemo.pojo">
  <class name="Car" table="car">
    <id name="licensePlate" column="license_plate">
      <generator class="assigned"/>
    </id>
    <version name="version" column="VERSION" unsaved-value="negative" />
    <property name="description" column="description"/>
    <property name="available" column="available"/>
    <property name="size" column="car_size"/>
  </class>
</hibernate-mapping>
```

> **Dikkat:** `version` komutunun `id` etiketinden hemen sonra gelmesi mecburidir.

Etiket `unsaved-value`'nun `id` etiketinde değil `version` etiketinde altında kullanılmış olmasının sebebi şudur: Eğer bir nesnenin kimlikleme yöntemi `assigned` ise Hibernate bir POJO'nun iç verisine bakarak o POJO'nun veri tabanına yazılıp yazılmadığını anlayamaz, çünkü kimlik POJO'nun içinde hep

olacaktır. Kimlik "üretiliyor" olsaydı, kimlik "olmadığı" zaman Hibernate bu nesnenin tabana daha yazılmadığını anlardı.

Peki bunu anlamak niçin önemlidir? Çünkü Hibernate `saveOrUpdate` ya da `merge`, bir nesnenin yeni mi eski mi olduğunu "bir şeylere bakarak" anlayabilmelidir. Bu bilgiye dayanarak arka planda eski nesne için `UPDATE`, yeni nesne için `INSERT` üretilecektir.

Versiyon kullanımında nesnenin yeniliği ya da eskiliği `version` öğesine bakarak anlaşılabilir. Versiyon değerleri sıfırdan başladığı için, yeni nesne işareti -1 olur. Hibernate yeni/eski olmanın kriteri olarak `<version>` etiketi altında `negative` tanımlamamıza izin veriyor. O zaman her yeni POJO `Car` için `version`'un olağan değeri de `-1` olmalıdır ki böylece nesne hafızada ilk `new` ile yaratıldığında `version` değeri `-1` olur, ve Hibernate bu nesnenin veri tabanında mevcut olmadığını hemen anlar. Eğer bu nesne üzerinde `saveOrUpdate` ya da `merge` komutu kullanırsak, `UPDATE` değil, `INSERT` üretilecektir.

Son olarak versiyon bilgisini kullanıcı düşünme zamanı (think time) sırasında, yâni detay gösterme ve güncelleme ekranları arasında, tutacak bir yere ihtiyacımız var: Bu yer, Form nesnesi olabilir. Nasıl olsa Form üzerindeki her öğenin gösterilebilir türden olması gerekmiyor. O zaman `struts-config.xml` şöyle değişecek.

```
<form-beans>
  <form-bean
     name="AddCarForm"
     type="org.apache.struts.action.DynaActionForm">
    ....
    <form-property
       name="version"
       type="int"
       initial="-1"/>
    ....
  </form-bean>
</form-beans>
```

İlk değer (`initial`) olarak -1 değerini *form üzerinden de* vermiş olmamızın sebebi, `AddCarAction` durumunda yeni `Car` nesnesine bu değerin yazılmasını zorlamaktır. Eğer bunu yapmasaydık, ilk değer 0 olacak (Struts tarafından verilir) ve `0` değeri bu `Car` nesnesinin "yeni bir nesne" olduğunu göstermeyecekti.

Son olarak, `UpdateCarAction` kodunu görelim.

Liste 3.13: UpdateCarAction.java

```
1  public class UpdateCarAction extends Action
2  {
3      private static Logger logger = Logger.getLogger("appLogger");
4
5      public ActionForward execute(ActionMapping mapping,
6                                   ActionForm form,
7                                   HttpServletRequest request,
```

```
8                                HttpServletResponse response)
9         throws Exception {
10
11        DynaActionForm daf = (DynaActionForm) form;
12
13        ActionErrors errors=new ActionErrors();
14
15        String id = "";
16        String result = "";
17
18        try {
19            Session s = HibernateSession.openSession();
20            HibernateSession.beginTransaction();
21
22            Car car = new Car();
23            id = car.getLicensePlate();
24            BeanUtils.copyProperties(car, daf);
25            s.merge(car);
26
27            result = "success";
28
29        } catch (org.hibernate.StaleObjectStateException e) {
30            errors.add(ActionMessages.GLOBAL_MESSAGE,
31                      new ActionError("optimistic.car.update.failed", id));
32            saveErrors(request,errors);
33            HibernateSession.rollbackTransaction();
34            result = "fail";
35        }
36
37
38
39        // form icindeki degerleri sil
40        HttpSession session = request.getSession();
41        session.removeAttribute(mapping.getAttribute());
42        return mapping.findForward(result);
43    }
44 }
```

Satır satır açıklama:

- **11**: `ActionForm` nesnesi `DynaFormAction` nesnesine çevirilir (cast).
- **19-20**: Hibernate Session ve yeni bir transaction başlatılır.
- **22**: Güncellenmesi istenen nesne, boş olarak yaratılır.
- **24**: `BeanUtils` metotlarından `copyProperties` ile form üzerindeki değerler otomatik olarak `Car` üzerine taşınır. Versiyon değeri de bu taşıma sırasında

detay gösterme anındaki hâli ile geri gelecektir, çünkü form üzerinde hatırlanan hali odur.

- **25**: Hibernate'in güncellemeyi başlatması için `merge` çağırılır. Hibernate, versiyon numarasına bakarak bu işlemin bir `INSERT` değil, `UPDATE` olduğunu anlayacaktır, ve ayrıca `version` değerini `WHERE` filtresi içinde kullanarak güncelleme çakışması sorununu çözecektir.

- **27**: Bu Action sonunda dönülecek durum kodu bu aşamada `success` olarak seçiliyor, çünkü bu noktaya gelmişsek exception atılmamış demektir.

- **29-35**: Eğer güncelleme çakısması olursa, Hibernate bu durumu `StaleObjectStateException` hatası ile belirtecektir. "Stale" kelimesi İngilizcede "eskimiş" anlamına gelir, yâni üzerinden güncelleme yaptığımız nesnenin "eskimiş" değerleri taşıdığı bize söylenmeye çalışılmaktadır. Bu hatayı yakalayınca, ilk önce veri tabanındaki transaction'ı geriye sarmamız/iptal etmemiz (rollback) gerekir, ve, hatalı durumdan kullanıcıyı haberdar etmemiz gerekir. Biz de `application.properties` üzerinde tanımlanmış `optimistic.car.update.failed` hatasını kullanarak bu durumu rapor edeceğiz. Struts'ta hata mesajlarının tanımlanması ve kullanılması için 3.7 bölümüne bakınız. Veri tabanları kavramları için 9 bölümüne danışabilirsiniz.

- **35-41**: Form değerleri oturum üzerinden çıkartılır.

- **42**: Durum kodu geri döndürülür.

3.8.4 Büyük Sonuç Listelerini Sayfa Sayfa Göstermek

Kullanıcılar, eğer uygulamadaki herhangi bir nesnenin listesini görmek isterler ve bu liste tek sayfaya sığmayacak kadar büyük olması mümkün ise, programcılar listeyi bölüm bölüm gösterecek teknik stratejiler geliştirmelidir. Aslında bu soruna teknik *olmayan* basit bir çözüm, müşteriye "filtreleme seçenekleri" sağlayarak sonuç listesini küçültülmekten geçer. Meselâ demo'muzdaki `Car` nesneleri, büyük bir sistemde 10,000 sayısına ulaşmış ise, bu kadar arabayı hiçbir kullanıcı hiçbir listeleme şekliyle görmek istemeyecektir! Bu tür bir liste filtreleme ile küçültülmelidir. Ama filtre sonrası bile elimizde çok sayıda nesne varsa sayfalama tekniklerini kullanmak zorundayız.

Hibernate, sayfalama için `Query` nesnesi üzerinde set edilebilecek `setFirstResult` ve `setMaxResults` çağrılarını destekler.

```
Query query = ..
query.setFirstResult(0);
query.setMaxResults(20);
List l = query.list();
```

Bu iki çağrıya verilen `int` parametresi, sırasıyla, *kaçıncı satırdan* itibaren, *kaç tane* satır gösterileceğini belirler. İstediğimiz büyük sonuç listesinden daha fazla parça görmek istersek, "kaçıncı satır" parametresine yeni bir değer vererek aynı `maxResults` parametresi ile `Query`'den yeni bir `List` almalıyız. Her yeni sayfa, veri tabanı üzerinde yeni bir SQL işletilmesi demek olacaktır.

Arka planda üretilecek SQL komutu her ticari veri taban ürünü için farklıdır. Hibernate kullanmamızın faydasını burada hemen görmuş oluyoruz, çünkü Hibernate, her taban için lâzım olan SQL komutunu bilir, ve SQL üretimini ona göre yapar. Böylece bizim işlem Java kodumuzun tabandan tabana değişmesi gerekmez.

Tekniğin Struts/JSTL ortamında kullanılmasını görmek için `StrutsHibAdv` kodlarına bakabiliriz. Bu örnekte `list.jsp` sayfası, büyük bir listeyi 0'ıncı ilk sayfadan başlayarak ekrana basmaktadır. Veri tabanına çok miktarda veri yüklemek için `sample_paging.sql` dosyasındaki örnek verileri kullanabilirsiniz. Kullanıcı komutunu karşılayan Struts Action, `ListCarPageAction` içinde görülebilir.

Page (Sayfa) Nesnesi

Şimdiye kadar takip ettiğimiz kod kalıplarında action'a git, liste üret, sayfaya dön zincirlemesini yapmıstık. Bu üçlüyü sayfalama için değiştirmemiz gerekecek. `ListCarPageAction` içinden `List` dönmek yerine yeni yazacağımız bir `Page` nesnesi döneceğiz. Bunu yapmamızın sebebi, listeleme sayfalarında çokça kullanılan **sonraki**, **önceki** türünden düğmelerin ne zaman aktif ne zaman pasif olacağını bilebilmek için soru soracağımız bir nesnenin gerekliliğidir. Eğer sadece `List` döndürmüş olsaydık, bu soruları soramazdık, çünkü `List`, içinden geldiği `Query` nesnesi ile tüm alakasını o noktada kaybetmiş olurdu. Yeni yazacağımız `Page` class'ı sonuç listesi `List`'i içinde barındıracak, ayrıca JSP sayfasının "hangi sayfa", "daha kayıt var mı?" gibi soruları sorabilmesini yardımcı fonksiyonları sayesinde sağlayacaktır.

Liste 3.14: Page.java

```
public class Page {

    private List results;
    private int pageSize;
    private int page;

    public Page(Query query, int page, int pageSize) {
        this.page = page;
        this.pageSize = pageSize;
        results = query.setFirstResult(page * pageSize)
            .setMaxResults(pageSize+1)
            .list();
    }
```

```
    public boolean isNextPage() {
        return results.size() > pageSize;
    }

    public boolean isPreviousPage() {
        return page > 0;
    }

    public List getList() {
        return isNextPage() ?
            results.subList(0, pageSize-1) :
            results;
    }

    public Integer getNextPageCount() {
        return new Integer(page + 1);
    }

    public Integer getPreviousPageCount() {
        return new Integer(page - 1);
    }
}
```

`Page` class'ının kullanımı `setFirstResult` ve `setMaxResults`'dan bile rahat, çünkü `Page` nesnesine geçilen parametre, "kaçıncı satır" yerine 'kaçıncı sayfa" değeri olacaktır. Sayfadan hareketle gerekli satır hesapları arka planda `Page` tarafından yapılır. `CarDAO` kodlarındaki `fetchCar` sorgu metotu `Page`'i şöyle kullanır.

```
public Page fetchCar(String currPage, String nextPage) {
    Session s = HibernateSession.openSession();
    return new Page(s.createQuery("from Car"),
                    new Integer(currPage).intValue(),
                    new Integer(nextPage).intValue());
}
```

CarDAO'yu kullanan Struts Action ise şöyle olacaktır.

```
public class ListCarPageAction extends Action
{
    private static Logger logger = Logger.getLogger("appLogger");

    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
        throws Exception {

        String currPage = request.getParameter("currPage");
```

```
        String maxPage = request.getParameter("maxPage");

        CarDAO dao = new CarDAO();

        request.getSession().setAttribute("page", dao.fetchCar(currPage,
                                                               maxPage));
        request.getSession().setAttribute("maxPage", maxPage);

        return mapping.findForward("success");
    }
}
```

Metot `execute` içinde ilk yapılan iş hangi veri sayfasının ve bu sayfada kaç tane satır görüntülemek istenildiğine dair (`currPage` ve `maxPage`) adındaki parametrelerin okunmasıdır. Bundan sonra `CarDAO`'yu yaratıp bu parametreleri geçerek `fetchCar` çağrısını yapabiliyoruz. Geriye geri gelen sonuç `Page` nesnesini alıp session üzerine yerleştiriyoruz. Artık JSP sayfası session üzerinden `page` referansını kullanarak sonuç nesnemizi bulabilecektir. JSP sayfası şöyle olacaktır (kısaltılmış kod olarak).

Liste 3.15: list.jsp

```
1  <table>
2    <tr>
3      <td>
4        <c:if test='\${sessionScope.page.previousPage == "true"}'>
5          <a href="<c:url value="/car-list.do">
6            <c:param name="currPage"
7               value="\${sessionScope.page.previousPageCount}"/>
8            <c:param name="maxPage"
9               value="\${sessionScope.maxPage}"/>
10           </c:url>">
11           <bean:message key="main.previous.page"/>
12         </a>
13       </c:if>
14       <c:if test='\${sessionScope.page.previousPage == "false"}'>
15         <bean:message key="main.previous.page"/>
16       </c:if>
17     </td>
18     <!---- ayni şeyi sonraki düğmesi için de yap ---->
19     ...
20   </tr>
21 </table>
22 <html:form action="/add-car.do">
23   <table>
24     ...
25     <c:forEach var="car" items="\${sessionScope.page.list}">
26       <tr>
```

```
27          ...
28          <td>
29            <a href="<c:url value="/edit-car-details.do">
30              <c:param name="licensePlate" value="\${car.licensePlate}"/>
31              </c:url>">
32              <c:out value="\${car.licensePlate}"/>
33              :
34              <c:out value="\${car.description}"/>
35            </a>
36          </td>
37        </tr>
38      </c:forEach>
39    </table>
40  </html:form>
```

Satır satır açıklama:

- **4-13**: "Önceki" düğmesinin gösterilmesi. Bu düğmenin `c:if` koşulu üzerinden gösterilmesini sebebi, sadece gösterilecek sayfa var ise üzerine tıklanan "önceki" bağlantısına ihtiyaç olmasıdır. `c:if` ile yapılan kontrolün `sessionScope` üzerinden session'a eriştiğini görüyoruz. İlginç bir kullanım daha var: Session'daki `page` üzerinden geride sayfa olup olmadığı sorusu. Bunu `Page` class'ının `isPreviousPage` metotunu çağırarak yapıyoruz. Zincirleme olarak çağrı yapabilmek JSTL'in faydalı bir özelliğidir, `sessionScope.page.previousPage` gibi. Bu çağrıdan geriye gelen cevap bize daha fazla veri sayfası olup olmadığını söyleyecek. Bu satırlar arasında, daha fazla sayfa olması sorusuna `true` cevabı gelmesi planlandığımıza göre, o zaman, `c:url` ile "önceki" düğmesini gösterebiliriz. Peki bu düğme hangi URL değerlerini taşıyacak? Eğer biz `currPage=3` üzerinde isek, "önceki" URL'i `currPage=2`'yi göstermeli. Bu `-1` işlemini yaptırmak için `Page` class'ına `getPreviousPageCount` adlı bir metot koymuştuk. O komut işte burada işe yarayacak:

  ```
  <a href="<c:url value="/car-list.do">
    <c:param name="currPage"
             value="\${sessionScope.page.previousPageCount}"/>
  ```

  ile URL'imizi bir önceki sayfaya işaret edecek şekilde gösterebiliriz. Niye `-1` gibi basit bir işlemi hemen sayfa üzerinde yapmadık? Çünkü prensip olarak JSP üzerinde (`<% %>` ile) Java kullanmaktan kaçınmamız gerekiyor. Bu tür kodlar, görsel mantık ile işlem mantığını birbirine karıştırırlar ve bakımı zor kodlara sebebiyet verirler. Eksiltme ve çoğaltma, tek bir yerde, ve bir Java class'ının içindedir ve ait olduğu yer de orasıdır.

- **14-17**: Buradaki `c:if` şartı, geriye gitme işlemi için veri satırı olmaması durumunu kontrol eder. Eğer `Page` nesnesi bize geride daha fazla satır yok diyor ise, bu noktada "önceki" bağlantısı yerine, "önceki" kelimesini

String olarak basarız, ama tıklanabilir bir bağlantı vermeyiz. Böylece kullanıcı geriye gitme işlemini yapamayacaktır.

- **18**: Aşağı yukarı aynı işlemleri bu sefer "sonraki" düğmesi için yapılır. Tekrardan kaçınmak için bu kodları göstermedik. Fakat, `previousPage` yerine `nextPage`, ve `previousPageCount` yerine `nextPageCount` çağrıları kullanılır. Bu metotların tanımları için `Page` class'ına bakabilirsiniz.

- **25-38**: Önceki ve sonraki düğmeleri sayfanın üstünde yer alır. Hemen altında, listenin gösterimi yapılır. Artık tanıdık olan `c:forEach` ile `\${¬ sessionScope.page.list}` değişkenini geziyoruz. `Page` nesnesi üzerinde `CarDAO`'dan gelen sonuçların bir `List` olarak tutulduğunu belirtmiştik. Listeyi göstermek için erişimin yapıldığı her burasıdır.

3.8.5 Dosya Yüklemek (File Upload)

Uygulamamız için, bazen, sadece kullanıcı tarafından yüklenebilecek bir dosyanın sisteme alınması gerekir. Bu dosya, bir JPEG resmi, bir PDF dosyası, ya da bir text düz metni olabilir.

Struts ile dosya yüklemesi gerçekleştirmek için, Commons FileUpload projesini kullanmamız gerekiyor. Bu projenin jar'ı ve bu jar'ı kullanan örnek dosya yükleme işlemi gerçekleştiren kodları, `StrutsUpload` projesinde bulabilirsiniz.

Projede gösterildiği şekilde bir dosya yükleyebilmek için, öncelikle, dosya ismini kullanıcıya soran bir JSP sayfası yazmamız gerekiyor.

```
<html:form action="/upload.do" enctype="multipart/form-data">
  Please enter the file you want to upload:<br/>
  <html:file property="formFile" /><br/><br/>
  <html:submit/>
</html:form>
```

Etiket `<html:form>` içindeki `<html:file>` ile, sayfa üzerinde kullanıcıya bir dosya seçmesini sağlayan bir öğenin yerleştirilmesini sağlıyoruz. Form'un gönderileceği yer olarak ta `/upload` adlı bir Struts Action'ı tanımladık. Bu Action, bizim yazdığımız (ve her proje içinde olduğu gibi kullanılabilecek) `UploadAction` kodudur. Ayar dosyası `struts-config.xml` içindeki tanımı aşağıdaki gibidir:

```
<action path="/upload"
        type="org.mycompany.kitapdemo.actions.UploadAction"
        name="UploadForm">
  <forward name="success" path="/pages/success.jsp"/>
  <forward name="error" path="/pages/error.jsp"/>
</action>
```

Gördüğümüz gibi Action, form olarak `UploadForm` alıyor. Bu form'un içinde sisteme yüklenmiş olan dosya binary olarak taşınmaktadır. Zaten bu sebeple `<html:form>` içinde `UploadAction`'ın çağırımında `enctype` kullandık. Bunun

yapılma sebebi, `UploadAction`'a verilen form'un, üzerinde basit tipler olan bir form değil, bütün bir dosya taşıyor olmasıdır.

Action, bu bütün dosyayı binary olarak alınca tek yapması gereken belli bir dizinde bir dosya açıp, dosyayı o dizine yazmasıdır. Dosya yazılması bitip geriye başarı kodu dönülünce, önceden `UploadAction` içinde tanımlanmış olan dizin içinde yeni bir dosya olduğu görülecektir.

Eğer bu yeni yüklenen dosya üzerinde ek işlemler yapılmasını istiyorsak, Struts'ın modüler zincirleme yapısını kullanarak yeni bir Action yazıp Struts çağrı zincirine dahil edebiliriz. Yâni, meselâ bir `UploadedFileProcessorAction` yazarak `struts-config.xml`'de tanımlarız

```
<action path="/process-upload"
       type="org.mycompany.kitapdemo.actions.UploadedFileProcessorAction"
       name="UploadForm">
  <forward name="success" path="/pages/success.jsp"/>
  <forward name="error" path="/pages/error.jsp"/>
</action>
```

ve önceki `/upload` zincirine şu şekilde ekleriz

```
<action path="/upload"
       type="org.mycompany.kitapdemo.actions.UploadAction"
       name="UploadForm">
  <forward name="success" path="/process-upload.do"/>
  <forward name="error" path="/pages/error.jsp"/>
</action>
```

Görüldüğü gibi `UploadAction` başarı (success) kodu artık JSP sayfası yerine, yeni işleyici Action'a yönlendirilmiştir. Nihai JSP sayfasına yönlendirmeye yapan, işleyici işlemleri bittikten sonra, `UploadedFileProcessorAction` class'ı olacaktır. İşleyici yeni Action kodu altta görülebilir. Kodun tek ihtiyacı olan yeni yüklenen dosyanın yeri ve ismidir. Bu bilgileri kendisine gecilecek olan `UploadForm` üzerinden hemen alınır.

```
public class UploadedFileProcessorAction extends Action {

    public UploadedFileProcessorAction() { }

    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
        throws Exception {

        UploadForm newForm = (UploadForm) form;

        String uploadedFile = newForm.getUploadedFileName();
        // yeni yüklenmiş uploadedFile ile istediğimiz işlemleri
        // burada yapabiliriz
```

```
        // ..

        return mapping.findForward("success");
    }
}
```

3.8.6 Kullanıcı İsim ve Şifre Kontrolü (Login)

Kurumsal uygulamalarda çok ihtiyaç duyulan kodlama kalıplarından biri sisteme giriş kontrolü için kullanıcı ismi ve şifre istenmesi ve kullanıcı ismine dayanarak kullanıcıya daha önce admin tarafından verilen roller ışığında, kullanıcının değişik içerik görmesini sağlamaktır. Meselâ bir finans uygulamasında banka hesaplarını sadece *görebilen* kullanıcılar olabilir, ve bu hesap bilgilerini hem görebilen hem de *değiştirebilen* kullanıcılar olabilir.

Login kodlama kalıbını gerçekleştirmek için, kullanıcı, şifre ve rol bilgisi idaresini Hibernate üzerinden yapacağız. Kullanıcı için `User`, rol için `Role` adlı iki POJO yaratılacak. Her `User`'ın birden fazla `Role` altında olabileceğini ve her `Role`'un birden fazla kullanıcı olabileceği gerekliliğinden hareketle, bu iki nesne arasında çoka çok (2.5.3 bölümü) türünden bir nicelik ilişkisi kuracağız.

Rol odaklı dinamik içerik için özel etiket (custom tag) teknolojisini kullanacağız. Aynen Struts `html:`, ya da JSTL `c:` etiketlerin gibi, biz de kendimize özel bir etiket kütüphanesi yaratabiliriz. Bu etiket kütüphanesine `mycompany:` etiketi üzerinden erişeceğiz. Etikette tanımlı `inRole` gibi bir soru ile, o anda sayfaya bakmakta olan kullanıcının bir rol içinde olup olmadığı sorusunu sorabileceğiz.

```
<mycompany:inRole role="Admin">
  Bunu sadece admin'ler görebilir
</mycompany:inRole>
```

Bu örneğe göre kullanıcı "Admin" rolünde bir kullanıcı ise, "bunu sadece admin görebilir" mesajını görebilecektir.

Bu kodun doğru çalışabilmesi için `inRole` etiketinin kullanıcı bilgisine erişmesi gerekir. O zaman, etiket işleme konmadan önce bir başka kodun kullanıcı bilgisini daha önceden erişilebilir, merkezi bir yere yerleştirmiş olması gerekmektedir. Bu kod parçası `login.jsp` adlı bir sayfanın gideceği `LoginAction` adlı bir Struts Action olabilir. Bu kod içine gelen `LoginForm`'daki kullanıcı ve şifre, veri tabanında (Hibernate ile) kontrol edilir, ve eğer kullanıcı girişine izin verilirse, session'a o kullanıcıya ait `User` konabilir. Böylece daha sonra işleme konacak olan `mycompany:inRole` etiketleri, session üzerinden alacakları `User`'dan, `Role`'larını vermesini isteyerek, sorulan rol altında olup olmadıklarını kontrol edebilirler.

Tag Library Yaratmak

Üstte tarif edilen tasarımın kodlanmış hâlini `StrutsHibLogin` projesinde bulabilirsiniz. Bu kodu parça parça açıklayalım. Bir etiket kütüphanesi (tab library) yaratmak için, etiket Java kodu, etiket tanım dosyası (tld), ve `web.xml`'de değişiklikler gerekir. Etiket kodu, alttaki gibi olacaktır.

```
public class InRoleTag extends TagSupport {

    String role;

    public String getRole() {
        return role;
    }

    public void setRole(String newRole) {
        this.role = newRole;
    }

    public int doStartTag() throws JspException {
        HttpServletRequest request =
                (HttpServletRequest)pageContext.getRequest();
        HttpSession session = request.getSession();

        User user = (User)session.getAttribute("user");

        Set roles = user.getRoles();

        boolean inRole = false;
        for (Iterator it = roles.iterator(); it.hasNext();) {
            Role role = (Role)it.next();
            if (role.getRoleName().equals(this.role)) {
                inRole = true;
            }
        }

        if (inRole == true) {
            return EVAL_BODY_INCLUDE;
        } else {
            return SKIP_BODY;
        }
    }
}
```

JSP sayfasında etiket yüklendiği zaman çağırılan kod önce `setRole` daha sonra `doStartTag` kodudur. Etiketin `role=''..''` parametresini kullanması Java kodu üzerinde `setRole` metotunun çağırılmasını tetikler. Tüm parametreler set edildikten sonra (bizim şartlarımızda bir tane), sıra `doStartTag` metotuna gelir.

Bu metot içinde etiket altına düşen HTML'in gözüküp gözükmeme kararını vermemiz gerekiyor. Bunlardan birincisi için `EVAL_BODY_INCLUDE`, diğeri için `SKIP_BODY` sabit değerlerini döndürmemiz yeterlidir.

Bu karar verilirken session üzerinden `User` nesnesinin alındığına dikkat ediniz. O zaman önceden `User`'ı oraya koyan (şifresini kontrol ettikten sonra) birileri olmalı. Bu kod, yâni `LoginAction`, şöyle olabilir.

```
public class LoginAction extends Action
{
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
        throws Exception {

        HttpSession session = request.getSession();

        Session s = HibernateSession.openSession();
        HibernateSession.beginTransaction();

        DynaActionForm daf = (DynaActionForm) form;
        if (logger.isDebugEnabled()) logger.debug("daf=" + daf);

        User user = (User)s.get(User.class, (String)daf.get("userName"));

        if (user.getPassword().equals((String)daf.get("password"))) {
            if (logger.isDebugEnabled())
                 logger.debug("user is authenticated");
            session.setAttribute("user", user);
        }

        // form icindeki degerleri sil
        session.removeAttribute(mapping.getAttribute());

        // geri don
        return mapping.findForward("success");
    }
}
```

Bu kodun görevi form'dan kullanıcı ve şifreyi almak, kontrol edip `session.setAttribute("user", user)` çağrısını kullanarak session üzerine bir kullanıcı nesnesi koymaktır. Bu Action'ı çağıran `login.jsp` adlı bir giriş sayfası olacaktır.

```
...
<html:form action="/login.do">

   <html:text property="userName" size="20" />
```

```
  ...
   <html:password property="password" size="20"/>

</html:form>
```

Şifre girilen form alanı, yazılırken gözükmemesi için `html:text` ile değil, `html:password` komutu ile alınmıştır. Bu etiket özel bir Struts etiketidir, ve şifre alanları için kullanılır. İçine yazılan bilgiler kullanıcı yazarken yıldız ('*') olarak gözükecektir.

Etiket Kütüphanesi Tanımı (TLD)

Bir etiket kütüphanesini JSP sayfalarına tanıtmak için üç işlem yapmamız gerekir:

1. TLD dosyasını yaratmak
2. web.xml'den bu TLD'ye referans etmek
3. JSP sayfası içinde tablib'i tanımlamak

TLD dosyası, alttaki gibi olacaktır.

```
<taglib>
  <tlib-version>1.0</tlib-version>
  <jsp-version>1.2</jsp-version>
  <short-name>mycompany</short-name>
  <uri>http://www.mycompany.com</uri>
  <display-name>Bilgidata TLD</display-name>
  <description>Bilgidata Tag Library (taglib)</description>
  <tag>
    <name>inRole</name>
    <tag-class>org.mycompany.kitapdemo.util.InRoleTag</tag-class>
    <attribute>
      <name>role</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

Etiket kodunun `<tag-class>` ve parametrelerin `<attribute>` altında tanımlanması gerekmektedir; `inRole` parametresi (Java koduna tekabül edecek şekilde) `<attribute>` altıda `<name>` alt etiketi ile tanımlanmıştır. Bu ayar dosyasını `web.xml`'e etiket kütüphanesi olarak tanıtmak için:

```
<web-app>
  ....
  <taglib>
```

```
    <taglib-uri>/tags/mycompany</taglib-uri>
    <taglib-location>/WEB-INF/tags/mycompany.tld</taglib-location>
  </taglib>
</web-app>
```

Ve son olarak, JSP'den tld'mizi yükleyip kullanmak için

```
<%@ taglib uri="/tags/mycompany" prefix="mycompany" %>
...
<table>
  <tr>
    <td>
      <mycompany:inRole role="Admin">
        Bunu sadece admin'ler görebilir
      </mycompany:inRole>
    </td>
  </tr>
  <tr>
    <td>
      <mycompany:inRole role="Regular User">
        Bunu sadece normal kullanıcılar görebilir
      </mycompany:inRole>
    </td>
  </tr>
</table>
```

kullanımı gerekir. Geliştirme dizin yapısı altta görülebilir.

```
+- StrutsHibLogin
| +- dd
| | +- META-INF
| | +- tags
| | | +- mycompany.tld
| | | +- c-rt.tld
| | | ..
| | +- jboss-service.xml
| | +- struts-config.xml
| | +- web.xml
| | +- ..
| +- etc
| +- lib
| +- resources
| | +- application.properties
| | +- application_tr.properties
| | +- hibernate.cfg.xml
| | +- log4j.properties
| | +- log4j.xml
| | +- oscache.properties
| +- src
```

```
| | +- java
| | | +- org
| | | | +- mycompany
| | | | | +- kitapdemo
| | | | | | +- actions
| | | | | | | +- LoginAction.java
| | | | | | +- pojo
| | | | | | | +- Role.hbm.xml
| | | | | | | +- Role.java
| | | | | | | +- SimpleUserTest.java
| | | | | | | +- User.hbm.xml
| | | | | | | +- User.java
| | | | | | +- service
| | | | | | +- util
| | | | | | | +- InRoleTag.java
| | | | | | | +- ...
| | +- pages
| | | +- login.jsp
| | | +- main.jsp
| | +- sql
| | | +- sample_data.sql
| | | +- tables_mysql.sql
| +- build.properties
| +- build.xml
```

3.9 Özet

Web uygulamaları yazmak için gerekli geliştirme ortamını, MVC kavramlarını bu bölümde işledik. JSTL, Struts etiketleri ve Struts Action'ları beraber kullanılınca çok güçlü bir üçlü oluşturmaktadırlar, zannediyorum bu potansiyeli gösterebilmiş olduk. Sürekli ortaya çıkan Web programlama kalıplarını, destek kodları ile sunduk; Bu kalıplar güncelleme, büyük sonuç listelerini sayfalama ile gösterme, kayıt ekleme gibi kalıplardır. Hatalı durumları idare etmek diğer önemli bir husustur. Hem genelci, hem de sayfa/action başına özel durumlarda nasıl yapılacağı bu bölümde açıklanmıştır. Türkçe karakterleri idare edebilmek, ve aynı uygulamadan değişik dilleri destekleyebilme hakkında yazılanlar Türkiye'deki yazılımcılar için oldukça faydalı olacaktır zannediyorum. Bu bölümde işlenen kodları `StrutsTiles`, `StrutsHibSimple`, `StrutsHibTag` ve `StrutsHibAdv` altında bulabilirsiniz. Projenize örnek almanız için en ideal kodlar `StrutsHibAdv` projesi olacaktır.

Bölüm 4

Dağıtık Nesneler

Bu Bölümdekiler

- Dağıtık nesne mimarisi
- RMI
- JNDI
- EJB (Session Bean)
- JMS

Uzaktan nesne çağırma teknolojisi, üç seviyeli (three tiered) mimarilerin yükseldiği 90 başlarında oldukça popüler bir yaklaşım idi. Üç katmanlı mimarilerde, orta katmanda bulunan ve network üzerinden çağırılabilen önyüze servis eden "nesnelere" Uzak Metot Çağrısı (Remote Method Invocation) ile bağlanılıyor, bu nesnelerle bilgi alışverişi yapıldıktan sonra kullanıcıya dönülüyordu.

Fakat takip eden yıllarda Web ile yükselen Servlet odaklı yeni orta katman mimarisi ortadaki iş mantığı katmanını ele geçirerek, kurumsal programcıların ilgisini dağıtık nesnelerden Servlet odaklı teknolojilere çevirmesine sebep olmuştur. Orta katmandaki iş mantığı pür Java kodları üzerinden, yâni `import` ile Servlet tarafından aynı JVM/süreç içine dahil edilip çağırılabilen türden kodlar olduğu için, artık network'den çağırılabilen ve üzerinde nesnelerin olduğu ayrı bir katmana gerek kalmıyordu. Evet bazı mimariler bir *dördüncü katman* koyarak halâ Uzaktan Metot Çağrısı yapmaya devam etmişlerdir, fakat bu mimariler hem azınlıkta, hem de optimal oldukları da şüphe götürür bir durumda idiler.

Günümüzde zengin önyüz (rich client) teknolojilerinde hareketlenme gözükmektedir ve zengin önyüzler, Java dünyasında Swing ile inşa edilirler. Ve, "zengin önyüz" kelimesini telâfuz eder etmez orta katmanda uzak nesnelerden bahsetmemiz gerekir çünkü zengin önyüzün HTML üreten Web orta katmanı ile konuşması mümkün değildir. Bu bölümümüzde uzaktaki bir nesneyi network üzerinden çağırmanın yollarını göreceğiz.

Ama ondan önce "neden orta katman" sorusuna cevap vermemiz gerekiyor.

4.1 Neden Orta Katman

Orta katman, üç katmanlı bir mimaride önyüz ile veri tabanı arasında duran seviyedir. Servlet/JSP üzerinden HTML üreten, ya da zengin önyüze servis eden uzak nesnelerin olduğu katmanın ikisi de orta katmana iyi bir örnektir.

Eğer mimarimizde orta katman bulundurmasaydık, veri erişimini direk zengin önyüz tarafına koymamız gerekecekti. Zengin önyüzün veri tabanına direk bağlantı açması demek, her kullanıcının veri tabanına *tek bir* bağlantı açması demektir, ve kullanıcının tüm veriye erişim istekleri bu bağlantı üzerinden gerçekleştirilecektir.

Bu yöntemin avantajı, basit yapısıdır; Her kullanıcıya verilen bağlantı üzerinden istenilen veri erişim ihtiyacı karşılanabilir. Dezavantajı, aynı şekilde, her kullanıcıya bir bağlantının açılması ve o bağlantının o kullanıcıya bağlı kalmasıdır. Her veri tabanının eşzamalı destekleyebildiği bağlantı sayısı kısıtlıdır. O zaman her kullanıcıya bir bağlantı ayırırsak, sistemizdeki eşzamanlı kullanıcı sayısı, veri taban bağlantı sayısı limitini hiçbir zaman geçemeyecektir. Meselâ bir Linux makinasında kurulmuş PostgreSQL veri tabanının optimal eşzamanlı bağlantı sayısı 200 olsun. O zaman sistemde aynı anda çalışabilecek kullanıcı sayısı 200'dür. Bundan daha fazla olamaz.

Bu sayı, tabii ki modern yüksek ölçekli kurumsal sistemler için çok düşük bir sayıdır. Daha fazla kullanıcıyı destekleyebilmek için modern mimarilerde veri erişimi merkezileştirerek, taban bağlantılarını tek bir kişiye bağlamak yerine, merkezi erişimin kontrolünde bir havuzda tutmak yolu seçilmiştir. Artık önyüzün kendisi değil, önyüze servis eden uzak nesneler *ihtiyaçları olduğu anda* bağlantıyı havuzdan alıp, işleri bitince bağlantıyı havuza hemen vereceklerdir; Bu sayede eşzamanlı *taban bağlantısı* kadar *eşzamanlı işlem* gerçekleştirilebilmiş olur. Zaten ölçekleme kıstasımız bu olmalıdır: Daha fazla kullanıcıyı, daha fazla eşzamanlı işlemi karşılıyamadan destekleyemeyiz.

Her kullanıcıya tek bağlantı verdiğimiz yaklaşım niye verimsizdir? Bu yaklaşımda kullanıcının düşünme zamanı (think time) sırasında veri taban bağlantısı hiç kullanılmıyordu, ve bu değerli kaynağın zamanı israf edilmiş oluyordu. Yeni yaklaşımda bağlantılar ortaktır, paylaşılabilir, ve bir kullanıcı tarafından kullanılmadıkları zaman bir başkasının kullanabilmesi için havuzda bekliyor olurlar. Yeni yaklaşımda, kullanıcı düşünme zamanı sırasında hiçbir zaman veri taban bağlantısı tek kullanıcıya bağlı tutulmaz.

Orta katmanın bir diğer faydası, kodlama disiplini açısından, görsel kodları işlem mantığından ayırmamız için hatırlatıcı bir faktör olmasıdır. Bu ayırım tekniği, yazılım mühendisliği açısından tavsiye edilen bir yöntemdir, çünkü görsel teknolojiyi değiştirdiğimiz zaman, işlem mantığının aynı kalabilmesini isteriz. Özellikle birden fazla çeşit önyüz teknolojisine hizmet vermesi gereken sistemlerde, işlem mantığının görsel kodlardan ayrı olması hayati önem taşımaktadır. Eğer bu ayrım yapılmazsa, işlem mantığı kodu her önyüz teknolojisi için tekrar tekrar yazılıyor olurdu (Kural #7 ihlâli). Fiziksel olarak ayrı bir orta katman ile çalışınca, bu katmana gidecek kodların ayrı olduğu daha bariz olmakta, ve yararlı bir yazılım prensibini mimarimiz üzerinde dolaylı yoldan zorlamış olmaktayız.

En son olarak Hibernate, veri tabanından okuduğu nesneleri önbellekleme özelliğine sahip olduğu için, merkezi bir orta katmanın gerekliliği bir kez daha oraya çıkmaktadır. Önbellek, bir kullanıcının istediği nesneyi, ikinci kullanıcıya önbellekten servis edebildiği için, performans arttırıcı bir faktördür, çünkü veri tabanına gitme ihtiyacını azaltır. Önbelleğin bu şekilde kullanımı için de, iki kullanıcı da aynı merkezi yere gitmek zorundadır (Hibernate ikinci seviye önbelleği `SessionFactory` seviyesinde, her JVM'de bir tane olmak üzere tutar). Bu merkezi yer de orta katmandan başkası değildir.

4.2 Genel Mimari

Uzak nesneler ile iletişim kurmanın Java dünyasında birçok değişik yöntemi vardır. RMI, EJB (Session Bean) ve JMS üzerinden bu iletişimi gerçekleştirmenin yollarını bu bölümde göreceğiz. Ama ondan önce, tüm bu iletişim teknolojileri üzerinden kullanabileceğimiz, teknolojiden bağımsız bir uzak nesne mimarisi göreceğiz. Bu mimariye, Command mimarisi adını veriyoruz.

4.2.1 Command Mimarisi

Command kalıbı ilk kez Design Patterns kitabında [2, sf. 233] Gamma ve arkadaşları tarafından ortaya konulmuştur. Tam tanımı şöyledir: "Command, bir isteği (request), *nesne* olarak tanımlayan ve bize, zengin bir parametre listesi olarak gönderebileceğimiz, log'a yazabileceğimiz, queue üzerinde kaydedebileceğimiz, kendi hatasını nasıl düzelteceğini (undo) bilen bir birim ile çalışabileceğimiz bir mimari sağlar".

Command kalıbının ilk çıkış noktasının GUI odaklı hata düzeltme işlevleri (undo) olduğu zannedilmektedir. Bir nesnenin nesne olması için alışveriş listesi metotları gerekiyorsa [3, sf. 111], Command'ın GUI için kullanılmış olması anlaşılabilir; Hata düzeltmek, bir işlemi yapmayı bilen bir nesne üzerinde eklenebilecek faydalı bir alışveriş listesi (8.2.4) işlevidir. Biz bu bölümde, Command kalıbını dağıtık nesneler mimarimizi desteklemek için kullanacağız [1, sf. 320]. Getireceğimiz mimarinin kuralları ve kısıtlamaları şunlar olacak.

1. Önyüzden servis tarafına (orta katman) yapılan her istek, bir Command nesnesi olmak zorundadır.

2. Her Command kurucu metotu kendisi için gereken parametreleri alıp içinde saklamakla yükümlüdür. Olağan (default) kurucu metot çağrısı, yanlışlıkla çağırılmaması için `private` tanımı ile engellenecektir (Kural #3,#4)

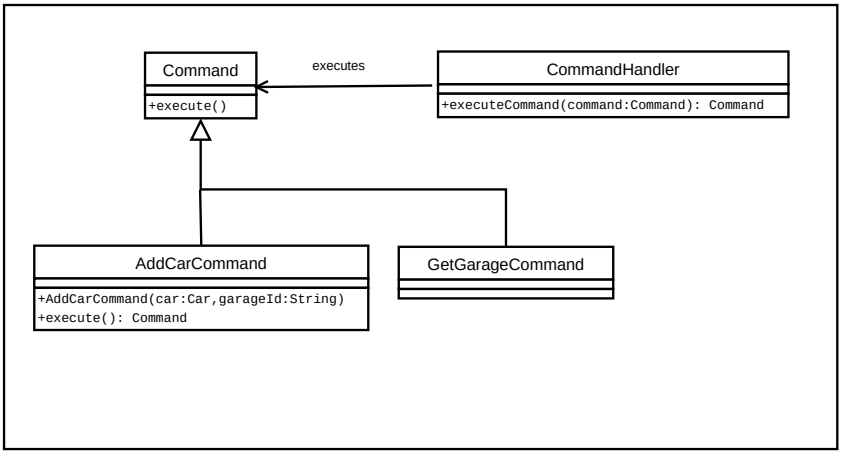
3. Command nesnelerini işletmek, tek bir işletici nesnenin sorumluluğu olacaktır.

4. Bir Command'ın görevi, geriye bir cevap getirmek ise, bu cevap Command nesnesinin içinde tutulacak, ve bu cevap nesnelerine `get` erişimi sağlanacaktır.

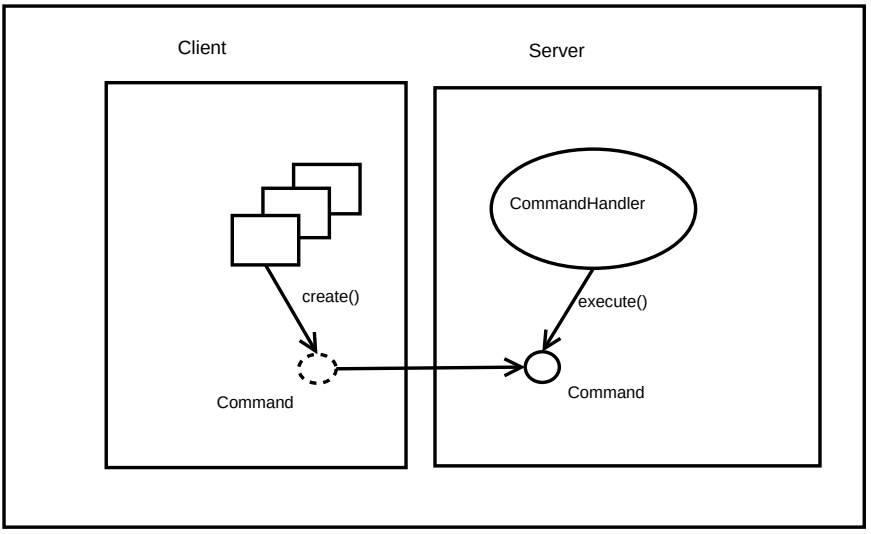
5. Command nesneleri, pür Java nesneleri olacak, hiçbir iletişim teknolojisine (EJB Session Bean, JMS, RMI) bağımlı yazılmayacaklardır.

CommandHandler

Command mimarisinin dağıtık mimarilere getirdiği faydaları nedir? Birincisi, servis tarafına `CommandHandler` üzerinden *tek bir giriş noktası* sağlanmasıdır. Çok yüzlülük (polymorphism) sayesinde, üst seviye `Command` class'ından miras alan (inherit) alt seviye iş yapıcı Command nesneleri, kendilerini sadece `Command` arayüzü üzerinden gören bir `CommandHandler` tarafından, *merkezi bir yerde* işletilebiliyor hâle gelirler (Şekil 4.1). Bu tek merkez, uzaktan çağrı teknolojisi değiştirilmesi gerektiğinde tekrar yazılacak yegâne noktadır! Eğer EJB Session Bean'den JMS'e geçiyorsanız, sadece `CommandHandler`'ı JMS ile işleyecek hâle getirebilirsiniz, ve bundan sonra tüm servis tarafı mimariniz JMS'te çalışmaya başlayacaktır.



Şekil 4.1: Command Nesne Yapısı



Şekil 4.2: Network Üzerinden Command Gönderimi

Ayrıca, bu tek giriş noktasında, tüm işlemler için gerekli olabilecek türden "ek işlemleri" rahatlıkla yerine getirebiliriz. Meselâ her istek sonucunda Hibernate oturumunu kapatmak, bir Command'den gelebilecek hataları yakalayıp Hibernate transaction'ı geri sarmak (rollback), ya da her Command için bir işletici (worker) Thread başlatmak gibi ek işler, bu tek merkezi noktada yapılabilir. Eğer, klasik "her istek için ayrı bir metot" yöntemini takip ediyor olsaydık, tüm işlemlere lâzım olan ek işlemleri merkezi bir yerde yapmak daha zor olurdu. Evet, Spring Framework Interceptor tekniği kullanarak her metot için kendi istediğimiz ek kodları çengel kod takarak işletebilirdik, fakat bu ek, teknoloji çorbasına bir teknoloji daha eklemiş olacak, Spring ayar dosyasını fazla şişirecek, hem de, zâten paketlenmiş bir istek (Command) bekleyen JMS seçeneği için tamamen gereksiz olacaktı.

Command Nesneleri

Command class'larının basit Java nesneleri olduğundan bahsetmiştik. Bunun üstüne, bir Command `Serializable` da olmalıdır, çünkü Command'ler network üzerinden gönderilirken içerikleri bozulmadan gidebilmelidirler. Tüm Command'lerin `Serializable` olması için bu arayüzden en üst seviyede miras almak gerekiyor.

```
public interface Command extends Serializable {
    public void execute() throws Exception;
}
```

Her özel Command, `Command` üst sınıfından miras aldığı için artık otomatik olarak `Serializable` olacaktır.

Her özel Command'ın işlem mantığı için yapması gerekenler, o Command'ın `execute` metotu içinde kodlanır. Command için gereken parametreler, bir kurucu metot üzerinden verilmelidir. Geri dönmesi gereken değerler ise, Command class'ının iç öğeleri olarak tutulmalıdırlar. Örnek olarak `GetCarCommand` kodlarını görelim.

```
public class GetCarCommand implements Command {

    String licensePlate;

    Car car;

    public Car getCar() {
        return car;
    }

    public GetCarCommand(String licensePlate) {
        this.licensePlate = licensePlate;
    }
```

```
    public void execute() throws Exception {
        Session s = HibernateSession.openSession();
        HibernateSession.beginTransaction();
        car = (Car)s.get(Car.class, licensePlate);
    }
}
```

Hibernate transaction'ın commit edilmesi ve oturumun kapatılması işlemlerinin Command içinde yapılmıyor olması belki dikkatinizi çekmiştir. Bu işlemler, merkezi bir yerde (`CommandHandler`) yapıldıkları için, ayrıca Command içinde tekrarlanmarlına gerek yoktur.

Bu bölümün geri kalan kısmında, RMI, EJB (Session Bean) ve JMS teknolojilerini teker teker tanıyacağız. Her teknolojiyi ilk ele aldığımızda, önce bize verdiği dağıtık nesne özelliklerini göreceğiz. Daha sonra, Command mimarisini alıp, bu yeni dağıtık teknoloji üzerinden çalışmasını sağlayacağız. Bu bölüm bittiğinde, elimizde üç değişik teknolojiyle çalışabilecek bir mimari yapımız olacak. Bu yapı öyledir ki, bir uzaktan çağırma teknolojisinden diğerine *projenin ortasında bile* geçebilme şansını elde edeceksiniz. Bu tür bir esnekliğin, her proje için faydalı olacağınız düşünüyoruz.

4.3 RMI

RMI, Uzak Nesne Çağrısı (Remote Method Invocation) Java dünyasında ilk uzak nesne çağrı sistemidir. Kullanım açısından en basit ve servis nesneleriniz üzerinde en az kısıt getiren teknoloji RMI'dır.

Sağladığı yetenek olarak, RMI ile uzaktaki bir nesneye network üzerinde çağrı yapabilme özelliğine kavuşuyoruz. Önemli bir nokta, bu servis nesnesinin tekil nesne (singleton) olacağıdır. Yâni RMI birçok bağlanan müşterinin çağrılarını, tek bir nesneye yönlendirir. Herkesin aynı objeyi kullanacak olması, servis tarafının çok thread'e hazır (thread-safe) olmasını gerektirir.

Servis tarafında metotları işleten Thread'lerin nasıl çalışacağı ürün geliştiricisine bırakılmıştır. Bu sebeple projenizde Thread politikasını kendinizin belirlemenizi tavsiye ediyoruz. Her gelen istek için yeni Thread başlatmak, bir Thread politikası olabilir.

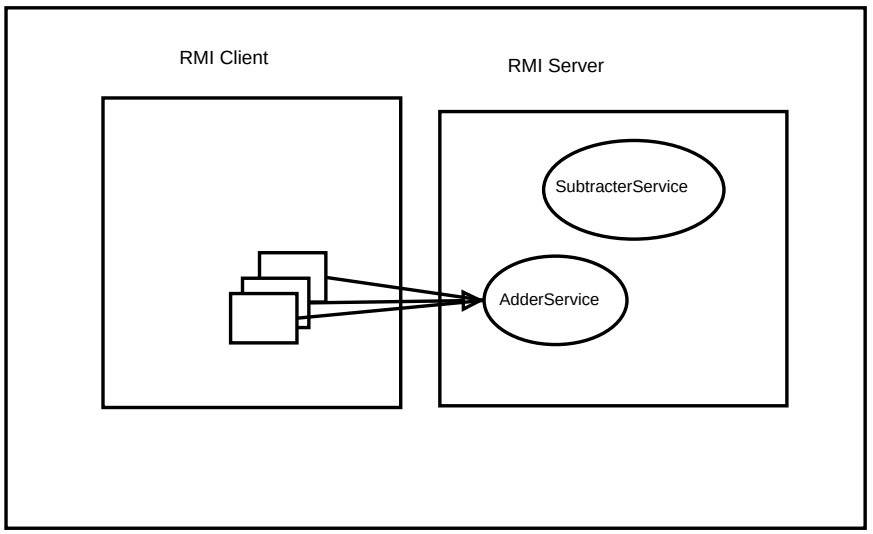
Bir nesneyi RMI üzerinden kullanıma açmak için, bir arayüz (interface) bir de gerçekleştirim (implementation) kodu gerekir. Arayüz bildiğimiz Java `interface` kelimesi ile tanımlanan arayüzdür. Gerçekleştirim bu arayüzü alır, ve her metotun işler kodunu tanımlar. RMI için, eski yöntemde, `rmic` adlı bir komut satırı programını arayüz ve gerçek kod üzerinde işleterek, içinde network kodları taşıyan ve Stub ve Skeleton olarak bilinen iki class üretilmesini gerektiriyordu. Biz, RMI teknolojisini Spring Framework üzerinden kullanacağız.

4.3.1 Spring

Spring Framework, J2EE kullanımını basitleştirmeyi ve amaçlayan bir altyapı projesidir. Spring'den bahsedilirken iki terimi sürekli duyacaksınız: Spring *bir IoC kabıdır* ve Spring *AOP yapmanıza izin verecektir.*

AOP, bildiğimiz gibi, birden fazla obje, tip, metota uygulanabilecek kod parçalarının nesnesel bir şekilde değil de çengel takar gibi koda nesne dışından eklemlenebildiği bir programlama şeklidir. Sonradan eklenebilen bu kodlara AOP dünyasında Aspect deniyor. Klasik örnek loglama örneğidir; Loglama, her kodun içine direk konması yerine, AOP dünyasında bir Log Aspect'i yaratılır, ve meselâ her metotun başında uygulanabilecek bir Aspect üzerinden dış loglama kodunun çağırılması sağlanır. Kodunuza sonradan takılan bu çengelin işlem anında çağrılması, AOP kabının sorumluluğudur, yâni nesnelerinizin işleyişi işlem anında AOP sistemi tarafından izleniyor olacaktır. Hayal edebileceğiniz gibi arka planda müthiş baytkod cambazlıkları dönmesi gerekecektir.

IoC, İngilizce Inversion of Control kelimelerinin kısaltılmışıdır, yâni Kontrolün Tersine Çevirilmesi anlamına gelir. Burada anlatılmak istenen, programcının bir nesneyi/kaynağı gidip bulması ya da `new` ile yaratması yerine, o nesnenin/kaynağın bir isim ile bir ayar dosyasında tanımlanması ve Spring'in bu nesneyi o tanıma göre yaratması, sonra da bu nesneyi sizin tarif ettiğiniz bir referans değişkenine set etmesidir. Yâni siz almıyorsunuz, size takdim ediliyor. Kontrolün tersine çevirilmesi tanımı işte buradan gelmektedir. Daha detaya inmek gerekirse (kod bağlamında) bahsedilen referans değişkeni, o referansı kullanacak işlem mantığı nesnesinin içinde yer alır. Bu referansa set eden bir



Şekil 4.3: RMI

metotu programcı yazmış olmalıdır, çünkü Spring, Java Reflection kullanarak (ve ayar dosyasındaki referans ismine göre) belirlenen set metotunu çağırır (program başında ve dinamik olarak).

Ayrıca Spring, size sadece baz anlamda AOP ve IoC özellikleri sağlayan bir paket değil, *tüm J2EE servislerinin* ve gözde açık yazılım paketlerini, *IoC ve AOP üzerinden kullanmanızı sağlayan* bir aracı servistir. Spring'i yazan programcılar, hem bir IoC/AOP kabı yazmış, hem de ek olarak bu temel kabı kullanarak J2EE arayüzleri/servisleriyle Ioc/AOP kabını teker teker bağlayarak bir ek seviye daha ortaya çıkarmıslardır. Spring'i Spring yapan esas seviye budur. Spring, J2EE servislerini IoC/AOP üzerinden sunmaktadır; Meselâ bir J2EE xyz servisi size *takdim edilen* bir nesne olacaktır, ya da, tüm nesnelerinize uygulanabilecek *J2EE Aspect'leri* olarak karşınıza çıkacaktır.

4.3.2 Basit Bir RMI/Spring Örneği

RMI servisi için bir Spring sarmalayıcı IoC servisi vardır. Servis tarafında `RmiServiceExporter`, çağıran tarafında `RmiProxyFactoryBean` üzerinden basit Java interface'inizi, kodunuzdan tek RMI arayüzü çâğırmadan RMI'a hazır hâle getirebilirsiniz! Servis tarafında:

Liste 4.1: ServerInterface.java

```
public interface ServerInterface {
    public int add(int a, int b);
}
```

Liste 4.2: ServerImpl.java

```
public class ServerImpl implements ServerInterface {
    public int add(int a, int b) {
        return a + b;
    }
}
```

Liste 4.3: springServer.xml

```
<beans>
    <bean id="servis" class="org.vs.vs.ServerImpl"/>

    <bean class="org.springframework.remoting.rmi.RmiServiceExporter">
       <property name="serviceName"><value>AdderService</value></property>
       <property name="service"><ref bean="servis"/></property>
       <property name="serviceInterface">
          <value>org.vs.vs.ServerInterface</value>
       </property>
       <property name="registryPort"><value>41199</value></property>
    </bean>
```

```
</beans>
```

Görüldüğü gibi gerçek kod (implementation) `ServerImpl` nesnesi Spring'de `servis` olarak isimlendirilmiş. Daha sonra bu isim, yani nesne, `RmiServiceExporter` adlı, `rmic` yerine geçecek olan Spring büyüsünü yapacak yardımcı class'a geçilmiş. Spring bu tanımları kullanarak J2EE standartına uyumlu bir network servisi `AdderService`'i yaratacaktır. Bu servis objesi üzerinde gerekli tüm RMI kod üretme işlemleri yapılmış, ve network'den kullanıma hazır hâle gelmiştir. Dikkat ederseniz, bütün bunları yapabilmek için servis tarafı kodumuz içinde tek bir RMI API'ı çağırmamız gerekmedi. Ayrıca, normal şartlarda `build.xml` içinde gerekecek `rmic` ile yapılan Stub/Skeleton üretim işlemi de artık tamamen Spring tarafından yapılmaktadır (koşma zamanında ama, uygulamanın başında ve bir kere olduğu için hiçbir performans farkı hissedilmeyecektir).

Servis tarafında bu objeyi aktif hâle getirmek için, başlangıç sırasında şu çağrıları yapabilirsiniz.

```
ClassPathXmlApplicationContext appContext =
    new ClassPathXmlApplicationContext(new String[] {"springServer.xml"});
```

Çağıran tarafta ise

Liste 4.4: springClient.xml

```
<beans>
    <bean id="remoteServer"
          class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
       <property name="serviceUrl">
           <value>rmi://localhost:41199/AdderService</value>
       </property>
       <property name="serviceInterface">
           <value>org.vs.vs.ServerInterface</value>
       </property>
    </bean>
</beans>
```

Burada görüldüğü gibi, uzaktaki `ServerImpl` nesnesini çağırmak için hiçbir RMI koduna ihtiyaç duymadık. Normâlde `Naming.lookup` gerektiren nesne bulmak işlemi, sadece Spring tanımları ile `RmiProxyFactoryBean` üzerinden yapılmaktadır. Spring'in network üzerindeki `AdderService`'ine erişebilecek bir nesne yaratabilmesi için, ona gereken tüm interface ve network erişim bilgileri `springClient.xml` tanım dosyası içinde tanımlıdır. Bu bilgileri kullanarak servise bağlanacak kod parçası altta gösterilmiştir.

```
ClassPathXmlApplicationContext appContext =
    new ClassPathXmlApplicationContext(new String[] {"springClient.xml"});
BeanFactory factory = (BeanFactory) appContext;
AdderService as = factory.getBean(‘‘remoteServer’’);
```

Ayar dosyası `springClient.xml` içinde belirtilen port değerinin `springServer.xml` içindeki port değerine uyması mecburidir. Bu port'lar, RMI kayıt (registry) port'udur. Tüm objelerin kayıt edildiği merkezi kayıt nesnesinin servis edildiği port adresidir.

4.3.3 RMI ve Command Mimarisi

Şimdi, bölüm 3'de tarif edilen ve `StrutsHibAdv` kodlarında temsil edilen arabalar ve garajlar örneğini, RMI ve Command mimarisine geçireceğiz. Kodların tamamlanmış hâlini `CarsRMI` dizini altında bulabilirsiniz.

Bu bölümün girişinde, zengin önyüzler sözkonusu olduğunda Command mimarisinin orta katmandaki Web kodlarının yerini aldığını söylemiştik. Hem Web, hem servis nesneleri birarada olmuyordu. Bu bölümün geri kalanında, *sadece örnek amaçlı olarak*, Web kodlarımızı servis nesneleri ile konuşturacağız. Bunun sebebi, örnek kodlarımız için Swing teknolojisine girmek istemeyişimizdir. Hem elde mevcut olan kodları kullanmak, hem de pür Struts bazlı kodlara servis nesne desteği ekleyince hangi noktaların değiştiğini görmek için, fiziksel mimari açısından optimal olmayanı yapacağız. `CarsRMI` projemiz, dört katmanlı olacak.

Command mimarisini RMI'a geçirirken, uzaktan çağrılmaya açmamız gereken tek class `CommandHandler` olacak.

CommandHandler

Liste 4.5: CommandHandler.java

```
public interface CommandHandler {
   public Command executeCommand(Command command) throws Exception;
}
```

Liste 4.6: spring.xml

```
<beans>
  <bean id="commandHandlerImpl"
        class="org.mycompany.kitapdemo.service.CommandHandlerImpl" />

  <bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <property name="serviceName"><value>CommandHandler</value></property>
    <property name="service"><ref bean="commandHandlerImpl"/></property>
    <property name="serviceInterface">
      <value>org.mycompany.kitapdemo.service.CommandHandler</value>
    </property>
    <property name="registryPort"><value>3045</value></property>
  </bean>
</beans>
```

`CommandHandler` arayüzünü gerçekleştiren `CommandHandlerImpl` class'ı üzerinde, `executeCommand` aşağıdaki gibi olacak.

Liste 4.7: CommandHandlerImpl.java

```
1  package org.mycompany.kitapdemo.service;
2  import org.apache.log4j.Logger;
3  import org.hibernate.HibernateException;
4
5  public class CommandHandlerImpl implements CommandHandler {
6
7          private Logger logger = Logger.getLogger("appLogger");
8
9          public Command executeCommand(Command command) throws Exception {
10
11            final Command param = (Command)command;
12            final Exception ex[] = {null};
13
14            // her yeni işlem için bir Thread aç
15            Thread t = new Thread (new Runnable() {
16                    public void run() {
17                        try {
18                            param.execute();
19                            HibernateSession.commitTransaction();
20                        } catch (HibernateException exx) {
21                            HibernateSession.rollbackTransaction();
22                            ex[0] = exx;
23                        } catch (Exception exx) {
24                            HibernateSession.rollbackTransaction();
25                            ex[0] = exx;
26                        } finally {
27                            HibernateSession.closeSession();
28                        }
29                    }
30                });
31            synchronized (this) {
32                t.start();
33                try { t.join(); } catch (Exception e) { }
34                if (ex[0] != null) {
35                    throw ex[0];
36                }
37            }
38
39            return command;
40        }
41 }
```

Satır satır açıklama:

- **11**: Metota gelen `command` referansı, `final` olarak tanımlanmış başka bir referansa transfer edilmelidir. Bunun sebebi, biraz altta bu referansa erişecek bir iç class (inner class) olmasıdır. İç class değişken erişim kurallarına göre, dışarıdan erişilecek tüm referanslar, `final` olarak tanımlanmalıdır. Eğer bu yapılmazsa, alttaki gibi bir hata alınır: ``‘‘local variable command is accessed from within inner class; needs to be declared final’’``

- **12**: Command'ı işletirken ortaya çıkabilecek `Exception`'ları iç class'ın dışına taşıyabilmek için, bir `Exception` *dizini* tanımlıyoruz. Niye sadece basit bir `Exception` referansı değil? Çünkü, iç class içinden, dışarıdaki bir referansa erişmemiz gerekirse bu referans `final` olmalıdır. Final olan bir referansa = ile hiçbir şey set edemeyiz! Ama bu engele tıkanıp kalmak yerine, bir hack ile işi çözüyoruz: İç class'tan bir referansa set edemesek te, `final` olan bir dizin *içine* atama yapmak, hâla legal bir harekettir. Biz de `Exception` yerine `Exception[]` kullanarak problemi çözüyoruz.

- **15,16**: `Runnable` arayüzünü gerçekleştiren bir class'ı anında oluşturup, anında Thread nesnesine çalıstırılmaya hazır olarak veriyoruz. Bu kullanım, biraz normâl dışı bir kullanımdır, fakat yeni bir Thread tarafından işletilecek kodları aynı kod sayfasında görebilmek için çok güzel bir tekniktir. Alternatif olarak, aynı kod bloğunu ayrı bir dosyaya ve class'a koyup ve Thread'e bu class'ı verebilirdik, ama bu, kod idaresi için pek faydalı olmazdı.

- **17-20**: İlk önce Command işletilir, ve işler yolunda gitti ise (hiç Exception atılmamışsa) transaction commit edilir.

- **20-23**: Hibernate tarafından atılan bir hata var ise, burada yakalanır. Bu durumda, transaction geriye sarılmalı (rollback) ve ele geçen hata `ex` dizini üzerine yazılmalıdır.

- **23-26**: Burada Hibernate hatası değil, daha genel bir hata olmuştur. Yapılan hata karşılama hareketleri yine aynıdır, burada ayrı bir bölüm olmasının sebebi, ileride değişik hata karşılama işlemlerini burada yapılabilecek olmasıdır. En azından her `catch` kendi log ifadelerini yazarak, ne tür bir hata meydana çıktığını servis tarafı log'larında gösterebilir.

- **26-28**: Hata olsa da olmasa da işleyecek `finally` bloğu içinde Hibernate oturumu kapatılır.

- **39**: Command işletildikten sonra *geri döndürülür*. Bu çok önemlidir çünkü geriye sonuç döndürmesi gereken Command'ler, sonuç değerlerini yine kendi üzerlerinde saklayacakları için, aynı Command'in geri dönmesi çok önemlidir. Çağıran tarafta istediği değerlere erişmek isteyenler, Command üzerinde `get` çağrıları yaparak servis tarafından gelen sonuçlara erişebilirler.

Çağıran Taraf

Çağıran (Web) tarafında Spring tanımı alttaki gibi olacaktır.

```
<beans>
  <bean id="commandHandler"
       class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
    <property name="serviceUrl">
      <value>rmi://localhost:3045/CommandHandler</value>
    </property>
    <property name="serviceInterface">
      <value>org.mycompany.kitapdemo.service.CommandHandler</value>
    </property>
  </bean>
</beans>
```

Bu tanım sayesinde, `getBean` kullanarak gereken her yerde `CommandHandler` referansını alabiliriz. Eğer bu referansın alımını hızlandırmak istiyorsak, Struts/Web ortamında, bu referansı oturum üzerinde sürekli hazır tutabiliriz, ve gerekince `getAttribute` ile oradan alırız.

`CommandHandler` Referansını hazırlamak ve set etmek için bir Servlet filtresi yazmamız gerekiyor. Bu filtre, her Web isteği başında "oturum üzerinde bir `CommandHandler` referansı olup olmadığını" kontrol eder. Eğer yoksa, `factory.getBean` ile bir tane alır ve oturum üzerine koyar.

Liste 4.8: ServiceReferencesFilter.java

```
public class ServiceReferencesFilter implements Filter {

    private Logger logger = Logger.getLogger("appLogger");

    public void init(FilterConfig filterConfig) throws ServletException { }

    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain) throws IOException,
                                                   ServletException
    {
        HttpSession session = ((HttpServletRequest) request).getSession();

        if (session.getAttribute("commandHandler") == null) {
            CommandHandler commandHandler =
           (CommandHandler)AppStartup.factory.getBean("commandHandler");
            session.setAttribute("commandHandler", commandHandler);
        }

        chain.doFilter(request, response);
    }
```

```
    public void destroy() { }
}
```

Artık her Struts Action'i içinden `ComnmandHandler` referansına erişebiliriz. Struts Action, servis ile iletişime geçmesi gerekince, gereken Command'i `new` ile yaratır, `CommandHandler`'a network üzerinde gönderir, ve Command'ın işletilmesini sağlar.

Liste 4.9: ShowCarDetailAction.java

```
public class ShowCarDetailAction extends Action
{
    private static Logger logger = Logger.getLogger("appLogger");

    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
        throws Exception {

        CommandHandler handler =
           (CommandHandler)request.getSession().getAttribute("commandHandler");

        String licensePlate = request.getParameter("licensePlate");

        GetCarCommand cmdGet = new GetCarCommand(licensePlate);
        Command resGet = handler.executeCommand(cmdGet);
        Car car = ((GetCarCommand)resGet).getCar();

        DynaActionForm daf = (DynaActionForm) form;
        BeanUtils.copyProperties(daf, car);
        if (car.getGarage() != null &&
            !car.getGarage().equals(new Integer(0)))
        {
            daf.set("garageId", car.getGarage().getGarageId());
        }

        request.getSession().setAttribute("car", car);

        // ....

        return mapping.findForward("success");
    }
}
```

Car nesnesi `setAttribute` ile oturum üzerine konduktan sonra, JSP sayfası arabanın detaylarını gösterecektir.

4.4 JNDI

Birazdan işleyeceğimiz EJB ve JMS teknolojilerini kullanmak için, JDNI adlı bir teknolojiye ihtiyacımız var. JNDI'ı biraz daha yakından tanıyalım.

JNDI (Java Naming and Directory Interface) arayüzleri, değişik türden dizin (directory) ve isimlendirme (naming) servislerine standart bir arayüz sağlar [4, sf. 24]. Bu açıdan JNDI, JDBC'ye benzer. Aynen JDBC'nin değişik türden veri tabanları ile konuşmamızı sağladığı gibi, JNDI da dizin ve isimlendirme servisleri olan LDAP, Novel Netware NDS, CORBA Naming Service ve şirketlere özel (proprietary) isimlendirme servislerine erişmemizi yardımcı olur. JBoss içinde de JNDI standart arayüzleri üzerinden erişebileceğimiz JBoss'a özel bir dizin ve isimledirme servisi vardır.

JNDI üzerinden birçok değişik servise erişebilirsiniz. Bu servisler, bir JMS Queue'su, bir Session Bean'i, ya da fiziksel bir yazıcı bile olabilir. Yâni JNDI, bir servis ile bir ismi birbirine ilintilendirerek, o servise o isim üzerinden erişebilmenizi sağlar.

RMI bölümünde, uzaktaki bir nesneye erişmek için JNDI'a konusuna girmemiz gerekmedi, çünkü hem servis hem bağlanan tarafı Spring üzerinden hallediyorduk. Spring, JDNI işlemlerini kapalı kapılar arkasında kendisi hallettiği için bizim fazladan JNDI işlemi yapmamıza gerek kalmamıştı. Fakat, görmek üzere olduğumuz EJB ve JMS teknolojileri için JNDI kullanacağız, çünkü JMS ve EJB teknolojilerini pür hâlde kullanıyoruz (böylesi daha kolay olacak). EJB bağlamında JDNI'ı, bir Session Bean'i ismiyle bulmak için, JMS için ise bir Queue'ya ismini kullanarak erişebilmek için kullanacağız.

JBoss için standart JNDI kodlama kalıbımız şöyle olacak:

```
import javax.naming.InitialContext;
import javax.naming.NamingException;
import java.util.Hashtable;
import javax.naming.Context;
import java.util.Properties;
import java.util.List;

...

InitialContext ic = null;

Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
     "org.jnp.interfaces.NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES, "jboss.naming:org.jnp.interfaces");
p.put(Context.PROVIDER_URL, "localhost:1099"); // JNDI port 1099 ise
try {
    ic = new InitialContext(p);
    Object objref = ic.lookup("buraya/jndi/ismi/yazilir");
    // ...
```

```
    // burada elde edilen referans ile işlemler yapılabilir
} catch (Exception e) {
    // hata durumunu rapor et
    e.printStackTrace();
}
```

Örnekte kullandığımız JNDI port'u, `1099` numaralı port'tur. Fakat bu port değeri bazı JBoss kuruluşlarında değişik olabilir (paketten çıkan hâli, `109¬ 9` olmak üzere ayarlıdır). Gerçek JNDI port'unuz değişik ise, bu değişik değerin ne olduğunu anlamak için JBoss açılırken basılan log mesajlarına bakmanız yeterli olacaktır. Örnek bir JBoss log ekranı aşağıda gösterilmiştir.

```
16:53:57,287 INFO [Server] Starting General Purpose Architecture (GPA).
16:53:58,539 INFO [ServerInfo] Java version: 1.4.2,Sun Microsystems Inc
16:53:58,549 INFO [ServerInfo] Java VM: Java HotSpot(TM) Client VM 1.4.
n Microsystems Inc.
16:53:58,549 INFO [ServerInfo] OS-System: Windows XP 5.1,x86
16:53:59,250 INFO [Server] Core system initialized
16:54:02,395 INFO [WebService] Using RMI server codebase: http://bio:80
16:54:03,126 INFO [NamingService] Started jndi bootstrap jnpPort=1099,
1098, backlog=50, bindAddress=/0.0.0.0, Client SocketFactory=null, Serve
Factory=org.jboss.net.sockets.DefaultSocketFactory@ad093076
16:54:11,338 INFO [Embedded] Catalina naming disabled
16:54:12,710 INFO [Http11Protocol] Initializing Coyote HTTP/1.1 on http
-8080
```

Bu mesaj satırları içinde `jnpPort` ibaresinin verildiği satıra bakınız. Eşitliğin sağındaki değer, JBoss'un JNDI servisi için kullandığı port değeridir.

4.5 EJB Session Bean

EJB teknolojisi, üç değişik kısımdan meydana gelir.

- Konumlu (Stateful) Session Bean (SFSB)
- Konumsuz (Stateless) Session Bean (SLSB)
- Entity Bean

Bu seçeneklerden Entity Bean teknolojisi, EJB'nin programcılar tarafından en tepki çeken ve kabul görmeyen kısmı olmuştur. Kalıcılık (persistence) problemini çözmeye çalışan Entity Bean'ler, çözdüklerinden daha fazla problemi beraberlerinde getirmişlerdir. Entity Bean teknolojisi, Kural #1, #5 ve #6'in çok ciddi şekilde ihlâlleridir.

Fakat, SFSB ve SLSB bileşenleri dağıtık mimarilere faydalı olabilirler. Özellikle, JBoss Uygulama Servisi'nin Session Bean'lere sağladığı çökmeden kurtulma (failover), yük dağıtımı (load balancing) servisleri sayesinde, sağlam ve ölçeklenebilen Session Bean bazlı sistemler kurmak mümkündür.

4.5.1 Sonuç Dizin Yapısı

JBoss üzerinde bir EJB'yi işletmek için `deploy` dizininde kurulması gereken dizin yapısı altta gösterilmiştir.

```
+- kitapdemo.ear
| +- META-INF
| | +- MANIFEST.MF
| | +- application.xml
| | +- jboss-app.xml
| +- conf
| | +- log4j.xml
| +- kitapdemo.sar
| | +- META-INF
| | | +- MANIFEST.MF
| | | +- jboss-service.xml
| | +- conf
| | +- activation.jar
| | ..
| | +- xml-apis.jar
| +- hibernate.cfg.xml
| +- kitapdemo.jar
| | +- META-INF
| | | +- ejb-jar.xml
| | | +- jboss.xml
| | | +- MANIFEST.MF
| | +- org
| | | +- mycompany
| | | | +- kitapdemo
| | | | | +- dao
| | | | | +- pojo
| | | | | | +- Car.class
| | | | | | +- Car.hbm.xml
| | | | | | +- Garage.class
| | | | | | +- Garage.hbm.xml
| | | | +- service
| | | | +- util
```

Bu yapı, örnek projelerin `build.xml`'i tarafından otomatik olarak kurulacaktır.

4.5.2 SFSB ve SLSB

Eğer uzaktan çağrı yaptığımız Session Bean bir SFSB ise, o Bean'e elimizde bir referans olduğu sürece, o Bean muhafaza edilir. SFSB'lerde, aynen `new` ile yaratılan Java nesnelerindeki gibi, bir önceki yapılan çağrı bir sonrakini etkiler çünkü eldeki SFSB, JBoss tarafından sabit/aynı tutulur. Kıyasla eğer eldeki referans bir S**L**SB'e işaret ediyor olsaydı, JBoss Uygulama Servisi konumsuz olarak bildiği bu nesneyi iki çağrı arasında değiştirebilirdi.

Peki Uygulama Servisi bu SLSB değiştirmesini niye yapar? Genellikle elde bir SLSB havuzu vardır ve gelen her yeni istek için havuzdan bir nesne alınıp çağrı üzerinde yapılır ve nesne havuza geri konulur. Önceden yaratılmış ve havuzdan servis edilen nesnelerin tekrar bir `new` aşamasından geçmesi gerekmez, ve hazır olan nesnelerin servis edilmesinin daha hızlı olduğu savunulduğu için bu yapı seçilmiştir. Havuzdan alma, havuza geri koyma işlemi, her yeni çağrıda yapılabilir, çünkü nesneler konumsuzdur; Bir çağrı, bir sonrakini etkilemez.

Tabii EJB belirtimleri (specification) aslında havuzlamadan bahsetmiyor; Belirtim sadece, SLSB'ler çağırılırken aynı nesnenin orada olacağına güvenilmemesinden bahsetmektedir (SFSB için ise tam tersi geçerlidir). Havuz kullanıp kullanmama gibi detaylar, her ticari uygulama paketinin kendi seçimine bırakılmıştır.

4.5.3 SFSB

Kodlama açısından bir SFSB için, üç tane class yazılması gerekiyor. Bunlar Home, Remote arayüzleri, ve gerçekleştirim kodlarıdır. Bu bölümde örnek olarak `MyTestSession` adında bir SFSB yazacağız. Bu Bean üzerinde `increment` ve `getCount` adlı iki metot olacak. `Increment` metotunu kullanarak SFSB üzerindeki bir sayaç değeri arttırabileceğiz, `getCount` ile mevcut sayının okunması mümkün olacak. Bu örneğin çok uygun bir Konumlu (stateful) Session Bean örneği olduğunu herhalde anlamışsınızdır, eğer `MyTestSession` konumlu değil konumsuz bir Bean olsaydı, arttırdığımız değerlerin hatırlanması mümkün olmazdı. Zaten SFSB ve SLSB arasındaki farkı anlamak için bu değiştirmeyi göstereceğiz.

Kod, alttaki gibi olacak (bitmiş kodları ve Ant `build.xml` dosyasını `CounterStateful` projesi altında bulabilirsiniz):

```
public interface MyTestSession extends javax.ejb.EJBObject{
   public void increment() throws java.rmi.RemoteException;
   public int getCount() throws java.rmi.RemoteException;
}

public interface MyTestSessionHome extends javax.ejb.EJBHome
{
    public MyTestSession create() throws
    javax.ejb.CreateException,
         java.rmi.RemoteException;
}

public class MyTestSessionBean implements SessionBean
{
    int counter = 0;

    public void ejbCreate() throws CreateException { }
```

```
    public void setSessionContext( SessionContext aContext )
    throws EJBException {}

    public void ejbActivate() throws EJBException { }

    public void ejbPassivate() throws EJBException { }

    public void ejbRemove() throws EJBException { }

    public void increment(){ counter++; }

    public int getCount(){ System.out.println(counter); return counter; }

}
```

Bu kodların deploy edilmesi için `ejb-jar.xml`, `jboss.xml`, `application.¬ xml`, `jboss-app.xml` ve `jboss-service.xml` adında beş dosya gerekmektedir. Tüm bu dosyalara teker teker bakalım (`jboss-service.xml`'i daha önce 3.3.3 bölümünde işlemiştik).

Liste 4.10: jboss-app.xml

```
<jboss-app>
  <module>
    <service>kitapdemo.sar</service>
  </module>
</jboss-app>
```

Liste 4.11: application.xml

```
<application>
  <display-name>KitapDemo</display-name>
  <description>KitapDemo Queue</description>
  <module>
    <ejb>kitapdemo.jar</ejb>
  </module>
</application>
```

Liste 4.12: ejb-jar.xml

```
<ejb-jar>
  <description>Kitapdemo</description>
  <display-name>Kitapdemo</display-name>
  <enterprise-beans>
    <!-- Session Beans -->
    <session>
      <display-name>My Test Session Bean</display-name>
```

```
      <ejb-name>test/MyTestSession</ejb-name>
      <home>org.mycompany.kitapdemo.service.MyTestSessionHome</home>
      <remote>org.mycompany.kitapdemo.service.MyTestSession</remote>
      <ejb-class>
         org.mycompany.kitapdemo.service.MyTestSessionBean
      </ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
  </assembly-descriptor>
</ejb-jar>
```

Liste 4.13: jboss.xml

```
<jboss>
   <enterprise-beans>
      <session>
         <ejb-name>test/MyTestSession</ejb-name>
         <jndi-name>ejb/test/MyTestSessionBean</jndi-name>
      </session>
   </enterprise-beans>
   <resource-managers>
   </resource-managers>
</jboss>
```

Görüldüğü gibi `application.xml` ve `jboss-app.xml` oldukça basmakalıp dosyalardır. Her proje için bir kez yaratılırlar, ve hiçbir EJB'ye has bir tanım içermezler. Bu iki dosyanın amacı, EJB kodlarının bulunduğu JAR ve SAR dosyalarının isimlerini EAR paketine tanıtmaktır.

Her EJB'ye özel ayarların yapıldığı yer `ejb-jar.xml` dosyasıdır. Meselâ eğer EJB'nin konumsuz olmasını istersek, yukarıdaki `<session-type>` etiketi içinde `Stateful` yerine `Stateless` kelimesini kullanabilirdik.

EJB tanımlaması hazırsa, `ejb-jar.xml` içindeki Session Bean tanımını bir JNDI ismine bağlayan yer de `jboss.xml` dosyası olacaktır. Bu dosyaya göre, EJB'mizin JNDI üzerinden bulunabileceği isim `ejb/test/MyTestSessionBean` ismi olarak belirtilmiştir.

Şimdi JNDI üzerinden `MyTestSessionBean` EJB'sini bulan ve çağrı yapan bağlantı kodunu görelim.

Liste 4.14: Mainline.java

```
public class Mainline {

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

```
        InitialContext ic = null;

        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
              "org.jnp.interfaces.NamingContextFactory");
        p.put(Context.URL_PKG_PREFIXES, "jboss.naming:org.jnp.interfaces");
        p.put(Context.PROVIDER_URL, "localhost:1099"); // JNDI port.
        try {
            ic = new InitialContext(p);
            Object objref = ic.lookup("ejb/test/MyTestSessionBean");
            MyTestSessionHome testSessionBean = (MyTestSessionHome)
                PortableRemoteObject.narrow(objref, MyTestSessionHome.class);
            beanRemote = testSessionBean.create();

            while (true) {
                beanRemote.increment();
                System.out.println("Count is : " + beanRemote.getCount());
                try {
                    Thread.currentThread().sleep(1000); // uykuya yat
                } catch (Exception e) { }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public Mainline() { }

}
```

Test kodumuzda önce JNDI üzerinden `MyTestSession` bulunuyor, daha sonra bu referans üzerinden, sonsuz bir döngü içinde SFSB üzerindeki sayaç bir arttırılıp, yeni değer ekrana basılıyor. SFSB içindeki servis kodunu hatırlarsanız, ekrana basma işlemi hem servis hem de çağıran tarafında yapılmaktadır. `Narrow` ve `create` çağrıları oldukça basmakalıp çağrılar oldukları için detaylarına inmeyeceğiz. Her EJB için ezbere takip edilmesi gereken metotlardır.

Çağıran Taraf ve Import

Bu noktada önemli bir soru, geliştirme ortamı açısından, çağıran taraf kodlarının `MyTestSessionHome` ve `MyTestSession` class tanımlarına nasıl sahip olduğudur. Yâni import'lar için gereken `.class` dosyaları nereden gelmiştir? Bu sorunun önemi, çağıran ve çağırılan tarafın bazen apayrı projeler olabilmesi durumunda daha da kuvvetle ortaya çıkacaktır.

Bu soruna çözüm olarak, servis tarafı kodlarının derlemesi bittikten hemen sonra, servis tarafı `build.xml`'i içinden, servis tarafı tüm class'ları bir `jar`'a

koyarak, çağıran projenin `lib` dizini içine kopyalamak uygundur. Bu kopyalama işleminin örneğini, `CarsEJB/Server` projesindeki `build.xml` içinde görebilirsiniz. Gereken class'lar `CarsEJB/Server`'den, `CarsEJB/WebClient/lib` altına kopyalanmaktadır.

4.5.4 EJB ve Command Mimarisi

SFSB üzerinden Command mimarisi kullanmak için, 4.1 bölümünde bahsedildiği gibi, tek "EJB'leştirmemiz" gereken yer `CommandHandler` kodu olacaktır. Bu nokta, servis tarafına network'den giriş noktası olduğu için, uzaktan çağırılacak tek noktadır. Bunun haricinde, eğer RMI bazlı Command mimarisini anlattığımız 4.3.3 bölümündeki Command ve alt sınıflarını olduğu gibi alıp kullanmak istersek, bunu yapabiliriz, çünkü Command class'ları hiçbir teknolojiye bağlı yazılmamıştır!

Şimdi yeni `CommandHandler` kodlarını görelim (EJB'ye geçerken, EJB stili isimlendirmeyi takip etmek için bu class'a `CommandHandlerBean` ismini vereceğiz).

```
public interface CommandHandler extends javax.ejb.EJBObject {
    public Command executeCommand(Command command)
        throws java.rmi.RemoteException;
}

public interface CommandHandlerHome extends javax.ejb.EJBHome
{
    public CommandHandler create()
        throws javax.ejb.CreateException, java.rmi.RemoteException;
}

public class CommandHandlerBean implements SessionBean
{
    public void ejbCreate() throws CreateException { }

    public void setSessionContext( SessionContext aContext )
        throws EJBException {}

    public void ejbActivate() throws EJBException { }

    public void ejbPassivate() throws EJBException { }

    public void ejbRemove() throws EJBException { }

    public Command executeCommand(Command command)
       throws java.rmi.RemoteException {

        final Command param = (Command)command;
        try {
            param.execute();
```

```
            HibernateSession.commitTransaction();
        } catch (HibernateException ex) {
            HibernateSession.rollbackTransaction();
            throw new RemoteException("",ex);
        } catch (Exception ex) {
            HibernateSession.rollbackTransaction();
            throw new RemoteException("",ex);
        } finally {
            HibernateSession.closeSession();
        }

        return command;
    }

}
```

Ne kadar basit olduğunu görüyoruz. `CommanHandlerBean` dısarıdan gelen bir Command'i alıp üzerinde `execute` metotunu çağırmakla yükümlüdür. Eğer metot başarıyla işletilirse Hibernate transaction commit edilecek, olmazsa rollback yapılacaktır. Her iki durumda da `finally` ile Hibernate oturumu kapatılır.

Yanlız EJB şartlarında RMI'a göre değişik, önemli bir nokta mevcuttur. EJB şartlarında eğer `execute` bir hata verir ise, `catch`'e gelen Exception'ı "`RemoteException` tipinde yeni bir Exception içine koyarak" geriye atmamız gerekmektedir. Yâni RMI şartlarındaki gibi, meselâ bir `HibernateException` nesnesini olduğu gibi network üzerinden geri atamayız. Niye?

Bunun sebebi, EJB belirtiminin, bileşenlerin arayüzleri üzerine getirdiği sınırlamalardır. EJB 2.1 belirtimine göre, çağıran tarafa geri atılabilecek hatalar sadece `RemoteException` tipinde olabilirler. Bu durum mimarimiz üzerinde biraz "sınırlayıcı" bir durum olabilir, fakat `RemoteException` içine diğer bir Exception gömmemiz mümkün olduğu için, bize gelen `HibernateException` nesnesini `RemoteException` içine gömerek geri gönderebiliriz (`RemoteException` kurucu metotu parametre olarak diğer bir Exception nesnesini alabilir, ve bir kez bu şekilde yaratıldıktan sonra içindeki nesneyle beraber geriye -çağıran tarafa- taşınabilir).

Bir `RemoteException` içinde saklanan diğer Exception'a `ex.detail` çağrısı ile erişebilirsiniz. O zaman çağıran tarafta örnek bir `catch` şöyle olacaktır:

```
try {
  CommandHandler handler = ...
  UpdateCarCommand cmd = new UpdateCarCommand(car, garageId);
  ...
  handler.executeCommand(cmd);

} catch (java.rmi.RemoteException e) {
    if (e.detail instanceof java.rmi.RemoteException) {
       java.rmi.RemoteException ee = (java.rmi.RemoteException)e.detail;
       if (ee.detail instanceof org.hibernate.StaleObjectStateException) {
```

```
    }
}
```

Exception'ın içine bakarken `e.detail.detail` diyerek niye iki seviye aşağı indik? Çünkü servis tarafında bir `RemoteException` atıldığı zaman EJB iletişim kodları bir Exception paketlemesi daha yaparak iki `RemoteException` içiçe koyulmaktadır. Bu aslında pek istenen bir durum değildir ve EJB teknolojisinin eksi hanesinde yazılması gereken bir gerçektir. Fakat idare edilebilir ve her şartta yapılması gerekmediği için fazla engelleyici bir durum teşkil etmez.

Üstte gösterilen Exception yakalama örneğini `CarsEJB/WebClient` projesi altında `CarUpdateAction.java` dosyasında bulabilirsiniz.

EJB Command mimarisi hakkında verilen detaylar bu kadar olacak. Tüm gereken ayar dosyaları belirtmeye gerek yoktur, çünkü ayarlar birkaç isim değişikliği yapıldıktan sonra `CounterStateful` projesininkiyle aynıdır. Bitmiş tüm arabalar ve garajlar projesi zaten `CarsEJB` altında bulunabilir.

4.6 JMS

JMS (Java Messaging Service), nesneler, JVM'ler ya da süreçler arasında asenkron iletişimi sağlayan bir Java servisidir. JMS, öncelikle bir standarttır; Bir belirtim (specification) belgesi ile ortaya konmuştur ve piyasadaki 'JMS uyumlu' yazılım paketleri bu belirtimi gerçekleştirip, desteklerler. JMS teknolojisinden bahsedilirken, yedi ana kavramı sürekli duyacaksınız.

- Queue
- Topic
- Subscriber
- Publisher
- Sender
- Receiver
- Message Broker

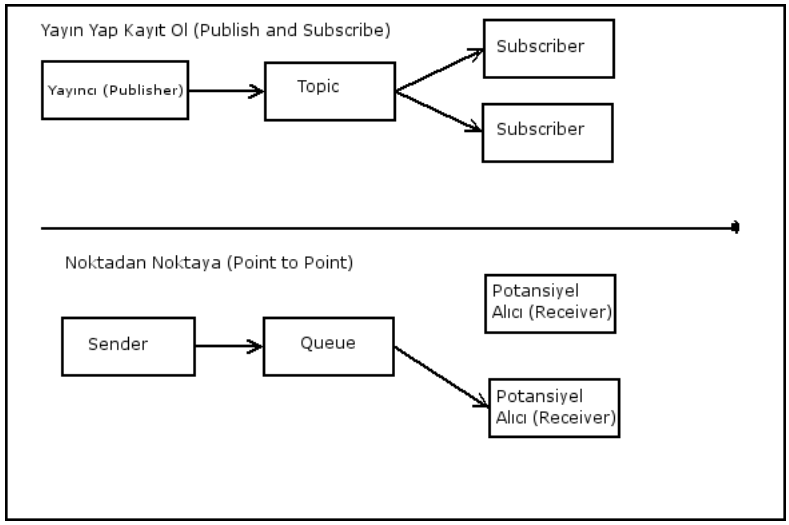
4.6.1 Ana Kavramlar

Queue ve Topic

Üzerine bilgi yazılan ve okunan birimlerdir. Queue ve Topic'in müşterileri (gönderen ve gönderilen uçlar) birbirleri ile asenkron iletişime geçmiş olurlar, yâni, gönderen taraf gönderme işleminden hemen döner, mesajın karşı tarafta alınıp alınmamış olduğuna emin olmak onun görevi değildir. Kıyasla metot çağrısı yaptığımız ve bu metot geri döndüğü zaman biliriz ki, metot içindeki

işlemler tamamlanmıştır. Asenkron iletişimde mesajın karşı tarafa güvenle ulaşmasını sağlamak Message Broker'ın görevidir.

Topic'in Queue'dan olan tek farkı, Queue dinleyicilerinin (listener) sadece bir tanesinin Queue üzerindeki bir mesajı okuması, Topic'te ise tüm okuyucuların (subscriber) tüm mesajları aynı anda almasıdır. Bir benzetme yapmak gerekirse, Queue telefon ise, Topic bir telsizdir. Birinin konuştuğunu, herkes duyabilir. Queue'ya gönderilen mesajlar, sadece ve sadece tek bir kişi tarafından okunabilir.



Şekil 4.4: Queue ve Topic

Sender ve Receiver

Queue'ya mesaj gönderene gönderici (sender), Topic'e mesaj gönderene yayıncı (publisher) ismi verilir.

Message Broker

Fiziksel anlamda queue ve topic'leri içinde barındıran yazılıma verilen isimdir. Ticari ya da açık yazılım paketi olarak isminden bahsedilen şey, Message Broker'dır. Piyasadaki örnekleri JBossMQ, ActiveMQ, SonicMQ, MQSeries ve WebSphereMQ olarak sayılabilir. Eğer JMS mantıki (logical) bir kavram ise Message Broker fiziksel (physical) bir kavramdır.

Bu kitapta kullandığımız, ve projeniz için tavsiye ettiğimiz Message Broker, JBossMQ ürünüdür. JBoss kurumundan gelen diğer yazılımlar gibi, bu yazılım da açık kaynaktır. Oldukça hızlı, projelerde ispatlanmış ve kullanımı

basit bir Broker ürünüdür. Queue ve Topic ile alâkalı ayarları zaten kullanmakta olduğumuz `jboss-service.xml` ayar dosyasından yapılabilmektedir. Örnek için `SimpleListenerServer` projesine bakabilirsiniz.

JMS sahneye çıkmadan önce, revaçta Tibco, MQSeries gibi broker'lar ve bu broker'ların kendi arayüzleri, kendi çağırma stilleri vardı. Tibco ve MQSeries hâla mevcutturlar, fakat JMS sahneye çıktıktan sonra tüm broker ürünleri JMS arayüzünü desteklemeyi seçtmişlerdir. Artık Tibco ve MQSeries ürünleri JMS üzerinden kullanılabilir durumdadır.

4.6.2 JBossMQ ile Queue ve Topic Oluşturmak

Statik (durağan) Queue ya da Topic oluşturmak için, `jboss-service.xml` ayar dosyasında bir MBean tanımı yapabiliriz. Dinamik (işlem anında) yaratılan Queue ve Topic'leri, tavsiye etmediğimiz bir kullanım olduğu için, bu kitapta işlemeyeceğiz.

JBoss evreninde çalışan her idare edilebilir nesne bir MBean'dir; Statik bir Queue ve Topic de MBean olarak tanımlanırlar. MBean tekniği sayesinde, idare edilen nesnelerin başlangıç değerlerini MBean teknolojisi üzerinden set etmek mümkün olmaktadır. Mesela aşağıda, `kitapDemoQueue` adında bir Queue'nun MBean üzerinden başlatıldığını ve ayarlarının yapıldığını görüyoruz.

```
<mbean code="org.jboss.mq.server.jmx.Queue"
       name="jboss.mq.destination:service=Queue,name=kitapQueue">
  <depends optional-attribute-name="DestinationManager">
     jboss.mq:service=DestinationManager
  </depends>
  <attribute name="MessageCounterHistoryDayLimit">-1</attribute>
</mbean>
```

Bir MBean'in başlangıç değerleri `<attribute name>` etiketi ile set edilebilir. Queue üzerindeki öğelerden biri olan `MessageCounterHistoryDayLimit`, bir Queue'ya gönderilen mesajların *sayaç değerinin* ne kadar uzun süre muhafaza edileceğini tanımlayan bir ayardır. Eksi değerler, sonsuza kadar anlamına gelir, artı değerler verilen gün değeri kadar sayaçın tutulmasını sağlayacaktır. Bu öğe ve diğer öğeler hakkında referans bilgisi için, [5, sf. 238]'e başvurabilirsiniz.

Üstte görülen ayarlar, JBossMQ'da bir Queue oluşturmak için yeterlidir. Topic yaratmak için, aşağıdaki gibi MBean tanımı yeterli olacaktır.

```
<mbean code="org.jboss.mq.server.jmx.Topic"
   name="jboss.mq.destination:service=Topic,name=kitapTopic">
  <depends optional-attribute-name="DestinationManager">
     jboss.mq:service=DestinationManager
  </depends>
  <depends optional-attribute-name="SecurityManager">
     jboss.mq:service=SecurityManager
  </depends>
</mbean>
```

Topic MBean ayarları için [5, sf. 239]'e başvurunuz. Ayar dosyası `jboss-service.xml`'in, geliştirme dizin yapısı içinde hangi dizinde tutulduğunu, ve `build.xml` ile nasıl paketlendiğini görmek için, `SimpleListenerServer` örnek projesine başvurabilirsiniz.

4.6.3 Listener ile Mesaj Okumak

Statik olarak oluşturduğumuz Queue ve Topic'den mesaj okuma tekniklerine bakalım. Önce, listener yöntemiyle okumayı işleyeceğiz.

Queue Listener

Listener yönteminde bir listener, `MessageListener` arayüzünden miras alır ve bir Queue üzerinde dinleyici nesne olarak set edilebilir. Bu yapıldıktan sonra, eğer dinlenen queue'ya bir mesaj gelirse, o mesaj bir `javax.jms.Message` olarak listener nesnesinin `onMessage` metotuna düşecektir. Dinleyici nesneler, *sen beni arama, ben seni ararım* mantığı ile işlerler. Örnek bir dinleyiciyi aşağıda görmekteyiz.

```
import javax.jms.*;
import javax.naming.Context;
import java.util.Properties;
import javax.naming.InitialContext;

public class KitapQueueListener implements MessageListener {

    public KitapQueueListener() throws Exception {

        QueueConnectionFactory qFactory = null;

        InitialContext jndi = null;
        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
              "org.jnp.interfaces.NamingContextFactory");
        p.put(Context.PROVIDER_URL, "jnp://localhost:1099"); // JNDI port
        InitialContext ctx = new InitialContext(p);

        QueueConnectionFactory qcf = (QueueConnectionFactory)
            ctx.lookup("ConnectionFactory");
        QueueConnection qc = qcf.createQueueConnection();
        Queue queue = (Queue) ctx.lookup("queue/kitapQueue");
        QueueSession queueSession =
            qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

        QueueReceiver qReceiver = queueSession.createReceiver(queue);
        qReceiver.setMessageListener(this);
        System.out.println("Listening on queue ..........");
```

```
        qc.start();
    }

    public void onMessage(javax.jms.Message message) {
        System.out.println(""+message);
        logger.debug(""+message);
    }
}
```

Örnek kodun `QueueReceiver` yaratılıncaya kadar olan kısmı JDNI üzerinden queue'nun bulunması, queue bağlantısı ve session açılması gibi basmakalıp işlemleri içermektedir. En son aşama ise listener nesnesinin, receiver üzerinde, kendisini (`this`), `setMessageListener` kullanarak bir dinleyici olarak set etmesidir. Tabii bu işlem, asıl dinleme işlemini başlatmak için yeterli değildir. Dinleme sürecini başlatmak için, `QueueConnection` üzerinde `start` çağrısı yapılarak okuma işlemi fiilen başlatılmalıdır.

Son olarak *listener kodunun kendisini* tetiklemek için, JBoss içindeki `AppStartup` ya da herhangi bir `main` işlevini kullanabilirsiniz. Bu çağrı çok basit olacaktır. Mesela komut satırından başlatılabilecek türden bir tetikleyici şöyle olabilir:

```
public class QListener {
    public static void main(String[] args) throws Exception {
        KitapQueueListener k = new KitapQueueListener();
    }
}
```

`SimpleListenerServer` projesinin `build.xml`'inde yukarıdaki class'ı başlatabilen `qlistener` adında bir Ant task'i bulacaksınız, yâni, komut satırında `ant qlistener` yazılınca yukarıdaki `main`'i çağırılmış olacak. İlginç bir nokta: Bu tetiklemeyi yaptıktan sonra, komut satırının geri gelmediğini farkedeceksiniz. Bu bloklanmanın sebebi, `start` çağrısı bir dinleyici thread başlatmasıdır, ve bu thread bitmeden, ant java komut çağrısı geri dönmez. Bu güzel, çünkü test amaçlı bir `main`'den beklediğimiz de zaten budur.

Topic Listener

Queue dinleme yöntemine neredeyse tıpatıp benzeyen topic dinleme işlemi, aynen queue için olduğu gibi `MessageListener` arayüzünden miras alır.

```
import javax.ejb.MessageDrivenBean;
import javax.jms.*;
import javax.naming.Context;
import org.apache.log4j.Logger;
import java.util.Properties;
import javax.naming.InitialContext;

public class KitapTopicListener implements MessageListener {
```

```
    public KitapTopicListener() throws Exception {

        TopicConnectionFactory qFactory = null;

        InitialContext jndi = null;
        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
              "org.jnp.interfaces.NamingContextFactory");
        p.put(Context.PROVIDER_URL, "jnp://localhost:1099"); // JNDI port
        InitialContext ctx = new InitialContext(p);

        TopicConnectionFactory qcf = (TopicConnectionFactory)
            ctx.lookup("ConnectionFactory");
        TopicConnection qc = qcf.createTopicConnection();
        Topic topic = (Topic) ctx.lookup("topic/kitapTopic");
        TopicSession topicSession =
            qc.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

        TopicSubscriber tSubscriber = topicSession.createSubscriber(topic);
        tSubscriber.setMessageListener(this);
        System.out.println("Listening on topic ..........");
        qc.start();
    }

    public void onMessage(javax.jms.Message message) {
        System.out.println(""+message);
        logger.debug(""+message);
    }
}
```

> **Not**: Queue kodlamasından topic kodlamasına geçerken güzel bir hatırlatıcı kural şudur: Bir queue kod çağrı kalıbını alıp, `queue` kelimesi yerine `topic`, `send` yerine `publish`, `receive` yerine `subscribe` kelimesini koyarsanız, queue işlem diziniz, topic işlem dizisine dönmüş olacaktır!

Yukarıdaki kodlama kalıbının queue okuma kodlamasına çok benzediği gözüküyor. Yazma ve okuma bağlamında çok benzeyen queue ve topic arasındaki ana fark, bir mesajı okurken o mesajı *kaç kişinin aldığı* ile alâkalıdır. Eğer üstteki gibi bir dinleyiciden birkaç tane olsa (değişik komut satırı pencerelerinden), topic'e bir mesaj geldiğinde *tüm dinleyiciler* bu mesajı alacaktır.

Tetikleyici kod, queue örneğine benzer olarak, alttaki gibi olacaktır:

```
public class TListener {
    public static void main(String[] args) throws Exception {
        KitapTopicListener k = new KitapTopicListener();
```

```
    }
}
```

Bu kod herhangi bir başlangıç kod bloğundan (meselâ `AppStartup` içinden) çağırılabilir.

4.6.4 Blok Eden Okuma

Listener yöntemi, *sen beni arama, ben seni ararım* tekniği ile çalışmaktadır. JMS altyapı kodları, dinleyici üzerindeki `onMessage` yöntemini her yeni mesaj geldiğinde çağırmakla yükümlüdür, böylece işlem mantığı kodlarınız hiçbir çağrı üzerinde blok etmemiş olur. Dinleyici tekniğini kullanan kodlar bu sebeple asenkron olarak addedilebilir; Dinleyen taraftaki program işleyişi aslında kimsenin `onMessage` çağırmasını beklemeden devam etmektedir.

Fakat, bazı programların *senkron* bir şekilde, yeni bir mesajın gelmesini *beklemeye* ihtiyaçları vardır. Bu şekildeki programlar için, blok eden türden `receive` adlı bir çağrısı kullanılır.

```
eQueueReceiver qReceiver = queueSession.createReceiver(queue);
qc.start();
TextMessage message = (TextMessage)qReceiver.receive();
```

Blok eden bu teknikle, `receive` çağrısı yapıldığı anda Java programınızın işleyişi beklemeye girer. Ta ki okunan queue üzerinde yeni bir mesaj gelinceye kadar, bu bekleyiş sürer, fakat yeni bir mesaj gelince, `receive` metotu geri döner. Metot çağrısından geri gelen değer, queue'ya (ya da topic) yeni gelen mesaj nesnesi olacaktır. Yukarıdaki örnekte bu yeni mesaj, `javax.jms.TextMessage` nesnesine cast edilmiştir.

4.6.5 Message Driven Bean İle Okumak

Listener tekniklerinden farklı olan bir okuma şekli, Message Driven Bean (MDB) kullanarak, queue ya da topic'lerden mesaj okumaktır. MDB teknolojisi, aslında EJB ve MessageListener tekniklerinin bir birleşimidir. MDB üzerinde, aynen bir listener'da olduğu gibi, bir `onMessage` metotu tanımlanır, ama aynı zamanda bir MDB bir EJB'dir, çünkü üzerinde `ejbCreate`, `ejbRemove` gibi işlevlerin tanımlanması gerekmektedir.

İşlevsellik açısından da MDB hem EJB hem Listener gibi davranır. MDB, aslında bir konumsuz (stateless) EJB olduğu için bir havuzda tutulabilir. MDB'yi konrolünde işletmekte olan kap (container), dinlenen queue'ya gelen *her mesaj için* havuzdan bir MDB alır, ve o MDB'nin `onMessage` metotuna yeni gelen mesajı aktarır. Mesaj MDB tarafından işledikten sonra, MDB havuza geri verilir.

MDB teknolojisinin tek dezavantajı, mesaj filtreleme tanımlarının (filtreleme tekniği detayları için 4.6.7 bölümüne bakınız) sadece ayar anında (XML ile)

yapılabilmesidir. Ne yazık ki, çoğu asenkron kurumsal uygulamanın bu ayar değişikliğine işlem anında ihtiyacı vardır.

Tanımlar

MDB aynı zamanda bir EJB de olduğu için, derleme ve paketleme sistemi aynı 4.5 bölümündeki EJB gibi *EAR içinde SAR* yöntemi olacak. Hattâ `build.xml` dosyalarının her iki proje için de aynı olduğunu görebilirsiniz.

Her MDB için, üç dosyada ayar yapılması gerekiyor:

- Queue ya da Topic tanımı (`jboss-service.xml`)
- MDB (`ejb-jar.xml`)
- MDB'nin hangi Queue ya da Topic'i okuduğu (`jboss.xml`)

Liste 4.15: jboss-service.xml

```
<server>
  <mbean code="org.jboss.mq.server.jmx.Queue"
        name="jboss.mq.destination:service=Queue,name=kitapServerQueue">
    <depends optional-attribute-name="DestinationManager">
       jboss.mq:service=DestinationManager
    </depends>
    <attribute name="MessageCounterHistoryDayLimit">-1</attribute>
  </mbean>
  ..
  <!-- Log4J alakalı tanımlar atlandı -->
</server>
```

Liste 4.16: ejb-jar.xml

```
<ejb-jar >
  <description>KitapDemo</description>
  <display-name>KitapDemo</display-name>
  <enterprise-beans>
    <!-- Message Driven Beans -->
    <message-driven >
      <description></description>
      <ejb-name>SampleMDB</ejb-name>
      <ejb-class>org.mycompany.kitapdemo.mdb.SampleMDB</ejb-class>
      <transaction-type>Container</transaction-type>
      <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
      <message-driven-destination>
        <destination-type>javax.jms.Queue</destination-type>
      </message-driven-destination>
    </message-driven>
  </enterprise-beans>
</ejb-jar>
```

Liste 4.17: jboss.xml

```
<jboss>
  <enterprise-beans>
    <message-driven>
      <ejb-name>SampleMDB</ejb-name>
      <destination-jndi-name>
         queue/kitapServerQueue
      </destination-jndi-name>
    </message-driven>
  </enterprise-beans>
</jboss>
```

Bu ayarlara göre, `SampleMDB` adlı MDB, `kitapServerQueue` adlı queue'ya gelen mesajları bekleyecektir. MDB'nin tanımı `ejb-jar.xml` dosyasında `<message-driven>` etiketi altında yapılmıştır. Bu etiket altında bir alt etiket olan `<destination-type>` etiketi altında ise, queue'mu yoksa topic'mi dinleneceği belirtilebilir. Queue tanımının kendisi 4.6.2 bölümünde anlatıldığı gibi, `jboss-service.xml` içinde yapılmıştır. Queue ile MDB arasındaki bağlantı da `jboss.xml` üzerinde `<ejb-name>`'i bir `<destionation-jndi-name>`'e bağlamak suretiyle gerçekleştirilmiştir.

Burada tarif edilen örnek kodları, `SimpleMdbServer` projesi altında bulabilirsiniz.

4.6.6 Mesaj Göndermek

Bir queue ya da topic'e mesaj göndermek için, öncelikle o queue ya da topic nesnesini JNDI ile bulmamız gerekiyor. Daha sonra, queue bağlantısı, queue oturumu ve en son olarak queue göndericisi yaratılarak, mesajın kendisi gönderilebilecektir. Aşağıda queue mesaj gönderim tekniğini görüyoruz.

```
Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
      "org.jnp.interfaces.NamingContextFactory");
p.put(Context.PROVIDER_URL, "jnp://localhost:1099"); // JNDI port 1099
InitialContext ctx = new InitialContext(p);

QueueConnectionFactory qcf =
           (QueueConnectionFactory)ctx.lookup("ConnectionFactory");
QueueConnection qc = qcf.createQueueConnection();
Queue queue = (Queue) ctx.lookup("queue/kitapQueue");
QueueSession queueSession =
           qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
QueueSender queueSender = queueSession.createSender(queue);

TextMessage textMessage = queueSession.createTextMessage();
textMessage.setText("merhaba dunya");
queueSender.send(textMessage);
```

Topic'e mesaj gönderme işlemi, queue'ya mesaj göndermeye oldukça benzer. Hattâ daha önce ortaya attığımız hatırlatıcı kuralı kullanabiliriz: Queue örneğindeki `queue` kelimesi yerine `topic`, `send` yerine `publish`, `receive` yerine `subscribe` kullanırsak, alttaki örneğe gelmiş oluruz.

```
// Context'i al (üstteki gibi)
// ..
TopicConnectionFactory qcf =
    (TopicConnectionFactory)ctx.lookup("ConnectionFactory");
TopicConnection qc = qcf.createTopicConnection();
Topic topic = (Topic) ctx.lookup("topic/kitapTopic");
TopicSession topicSession = qc.createTopicSession(false,
                                                  Session.AUTO_ACKNOWLEDGE);
TopicPublisher topicPublisher = topicSession.createPublisher(topic);

TextMessage textMessage = topicSession.createTextMessage();
textMessage.setText("merhaba dunya");
topicPublisher.publish(textMessage);
```

ObjectMessage Göndermek

`javax.jms.TextMessage` ile gönderebileceğimiz mesajlar ne yazık ki sınırlıdır. Kurumsal Java programları için, genellikle, `Serializable` arayüzünden miras alan kompleks bir Java nesnesini göndermek gerekecektir. Bu tür bir nesneyi yollamak için, `javax.jms.TextMessage` yerine `javax.jms.ObjectMessage` tipini kullanmanız gerekir.

Yeni (boş) bir `ObjectMessage`, `queueSession` üzerinden `createObjectMessage` çağrısı ile yaratılır (`new` ile kendimiz yaratamayız). Boş bir mesaj aldıktan sonra, göndermek istediğiniz kopmleks Java nesnesini `setObject` ile `ObjectMessage` *üzerinde* set etmemiz gerekiyor. Gerisi, bildiğimiz `send` çağrısından ibarettir.

```
ComplexObject obj = ...
ObjectMessage objectMessage = queueSession.createObjectMessage();
objectMessage.setObject(obj);
queueSender.send(objectMessage);
```

Okuyan tarafta, `javax.jms.Message` nesnesini alınca `ObjectMessage`'a cast etmemiz gerekir. Mesaj *içindeki* `ComplextObject`'e erişmemiz için ise, `ObjectMessage` üzerinde `getObject` çağrısı yapmamız gerekir. `ObjectMessage` üzerinde `getObject` çağrılması, bu tekniği ilk görenler için kafa karıştırıcı olmaktadır, çünkü "elimde zaten bir object var, niye bir daha `getObject` çağırıyorum" düşüncesi ortaya çıkar. Burada `TextMessage`'a bir paralel çizmek gerekir; `TextMessage`'ın da üzerinde de `getText` çağrısı yapmaktayız.

Kalıcı ve Uçucu Mesajlar

Bir JMS mesajı gönderirken eğer hiçbir ek parametre tanımlamazsanız, olağan (default) olarak mesajınız *kalıcı* mesaj olarak gönderilecektir. Kalıcı mesaj kullanımı, eğer Message Broker çökse bile, mesajların kaybolmasını engeller. Arka planda Message Broker, kalıcı mesajların kaybolmaması için, bir veri tabanı ya da düz dosyaya her mesajı yazmaktadır. Yâni eğer kalıcı mesaj gönderiyorsanız, her mesajınız bir şekilde disk'e yazılıyor olacaktır.

Kabul edilmesi gerekir ki, her mesajın disk'e yazılmasının bir performans bedeli olacaktır. Eğer mesajlarınızın daha hızlı gönderilmesini istiyorsanız, mesajlarınızı *uçucu* olarak ta göndermeniz mümkündür. Uçucu mesaj göndermek için, `send` çağrısına bazı ek parametreler vermemiz gerekir. Bu ek parametre `DeliveryMode.NON_PERSISTENT` parametresi olacaktır.

```
queueSender.send(textMessage,
                 DeliveryMode.NON_PERSISTENT,
                 Message.DEFAULT_PRIORITY,
                 0);
```

> **Dikkat**: Yapılan en yaygın JMS hatalarından biri, `DeliveryMode` değişkenini *mesaj* üzerinde set etmektir. Bu çağrının gönderim işlemi üzerinde hiçbir etkisi yoktur (niye hala JMS arayüzlerinden depracate edilmediği, ayrı bir konudur).

Topic üzerinden uçucu mesaj göndermek ise, queue mantığına benzer olacaktır.

```
topicPublisher.publish(textMessage,
                       DeliveryMode.NON_PERSISTENT,
                       Message.DEFAULT_PRIORITY,
                       0);
```

Kalıcı ve uçucu mesajlar arasındaki farkı görmek istiyorsak, JBossMQ'nun ve queue'larımızın çalıştığı makinaya `http://host:8080/jmx-console` url'inden bağlanıp, JmxConsole uygulamasını çalıştırabiliriz. JmxConsole programı, web arayüzü üzerinden çalışan ve bir JBoss servisi içindeki tüm MBean'leri ve içeriklerini listeleyebilen bir bakım programıdır. Biz daha önce queue tanımlamak için `jboss-service.xml` içinde bir queue MBean'i tanımladığımız için, JmxConsole'dan bu queue'yu ve içeriğini görebiliriz. JmxConsole url'ine gidince tarayıcınızdan queue'ların listelendiği bölüme gidin. Şekil 4.5 bunun bir örneğini gösteriyor.

Listeden içeriğini görmek istediğiniz queue'ya tıklayın (meselâ `kitapQueue`), ve `java.util.List listMessage()` yazan bölüme giderek `Invoke` düğmesine tıklayın. Yeni gelen sayfada queue'nuzun içeriğini göreceksiniz. Test için, üzerinde hiçbir dinleyici olmayan bir queue'ya kalıcı mesajlar yollayıp JBoss'u kapatıp açın, ve JmxConsole üzerinden mesajları halâ orada olup olmadığını kontrol edin. Aynı işlemi, uçucu mesajlar ıcin de yapın.

Öncelik

Metot `send`'in parametre listesinde gördüğümüz parametrelerden biri, öncelik parametresidir (priority). Bu parametre, hangi mesajın ne kadar önce hedefine ulaşacağını belirler. Kurumsal uygulamalarda, olağan öncelik değeri (default priority) yeterlidir.

Zamanlı Mesajlar

Eğer bir mesaj yerine ulaşmadan belli bir süre sonra zamanının geçmesini (expire) istiyorsak, `send`'e verilen son parametre olan `timeToLive` parametresinde bunu belirtebiliriz. Bu parametre, sadece belli bir zaman için geçerli veriler için uygun bir parametredir (meselâ borsada bir hissenin anlık değeri gibi -en son değer 10 saniye sonra geçersiz olmasını istiyorsak, vs-). Eğer `timeToLive` için `0` verilmiş ise, mesaj zamana bağlı geçersiz olmayacaktır.

4.6.7 Filtrelemek

Bir queue ya da topic'e gönderilen mesajların içinden sadece ilgilendiğimiz ve bir kıstasa uyan bazılarını seçmek istiyorsak, message selector üzerinden filtreleme tekniğinin kullanmamız gerekmektedir.

Filtreleme yapmak için, JMS mesajında bulunan, üzerinden filtre yapabileceğimiz bir parametre gerekmektedir. Bu parametreye 'mesaj parametresi' ismi veriyoruz. Mesaj parametresi set etmek için belli bazı JMS arayüzleri vardır. Her tip için, değişik bir parametre set çağrısı yapmak gerekiyor, tüm listeyi altta veriyoruz.

```
public void setStringProperty(String name, String value);
```

Şekil 4.5: JmxConsole'dan Queue İçeriğini Görmek

```
public void setIntProperty(String name, int value);
public void setBooleanProperty(String name, boolean value);
public void setDoubleProperty(String name, double value);
public void setFloatProperty(String name, float value);
public void setByteProperty(String name, byte value);
public void setLongProperty(String name, long value);
public void setShortProperty(String name, short value);
public void setObjectProperty(String name, Object value);
```

O zaman, gönderen taraf bir mesaj parametresi set etmek isterse, şu şekilde değiştirilmesi gerekecektir.

```
ComplexObject obj = ...
ObjectMessage objectMessage = queueSession.createObjectMessage();
objectMessage.setObject(obj);
objectMessage.setStringProperty("param1", ‘‘123456789’’);
queueSender.send(objectMessage);
```

Burada, diğer gönderme işlemlerine ek olarak, `setStringProperty` ile `String` tipinde bir mesaj parametresi set ettik. Artık `param1` üzerinden çalışacak bir filre yaratabiliriz. Filtreleri, hem listener hem de `receive` teknikleri üzerinden kullanmak mümkündür. Her şart için filtre yaratmayı altta görelim.

Listener ve Filtre

Meselâ `setStringProperty` ile set edilen, `param1` adlı parametre üzerinden işleyen, ve queue üzerinde bekleyen bir *listener* üzerinden filtre oluşturmak için

```
QueueReceiver qReceiver;
qReceiver = queueSession.createReceiver(queue,
                                        ‘‘param1 = ‘123456789’’’);
```

Aynı şekilde, ama bu sefer topic üzerinde beklemek için,

```
TopicSubscriber tSubscriber;
tSubscriber = topicSession.createSubscriber(topic,
                                            ‘‘param1 = ‘123456789’’’,
                                            false);
```

Bu çağrılar yapıldıktan sonra, `onMessage`'e gelen nesneler, artık sadece ve sadece filtre şartlarına uyan `javax.jms.Message` nesneleri olacaktır.

Receive ve Filtre

Bir queue üzerinden, blok eden `receive` üzerinden filtre ile mesaj almak için, şunları yapmak gerekir:

```
QueueReceiver qReceiver;
qReceiver = queueSession.createReceiver(queue,
                                        ‘‘param1 = ‘123456789’’’);
```

```
qc.start();
javax.jms.Message message = qReceiver.receive();
```

Topic için

```
TopicSubscriber tSubscriber;
tSubscriber = topicSession.createSubscriber(topic,
                                            ‘‘param1 = ‘123456789’’’,
                                            false);
qc.start();
javax.jms.Message message = tSubscriber.receive();
```

Yâni, bir receiver ya da subscriber yaratmak, dinleyici kodlaması ile neredeyse aynıdır, iki fark ile: Blok eden yöntemde `setMessageListener` ile dinleyici set edilmez, ikincisi, mesaj alma işlemi `receive` metotu çağırılarak, receiver/subscriber referansı üzerinden direk olarak yapılır.

4.7 JMS ve Command Mimarisi

Uzaktan Metot Çağrısı yöntemleri olan RMI ve EJB teknolojileri ile, Command mimarimizin ne kadar rahat gerçekleştirilebildiğini gördük. Command nesneleri `Serializable` nesneler oldukları için network üzerinden rahatlıkla gönderilebiliyor, ve ulaştıkları noktada `CommandHandler` tarafından işletilebiliyordu.

Eğer bu mimariyi asenkron bir yapıya çevirmek istersek, queue kavramını Command mimarimize bir şekilde dahil etmemiz gerekecek. Tahmin edilebileceği üzere, daha önce bir `execute` metot çağrısına parametre olan Command'ler, artık queue'lar üzerinden gidip gelen nesneler olacaklar.

Command işleyen CommandHandler ise, artık bir RMI ya da EJB nesnesi değil, bir JMS *dinleyicisi* olacaktır. Bu kavram değişikliğini iyice belirginleştirmek için, işleyici nesnenin ismini `CommandListener` olarak değiştireceğiz.

4.7.1 Fiziksel Yapı

JMS bazlı mimarilerde gündeme gelen ilk sorulardan biri, *kaç tane queue kullanalım* sorusudur. Bizce bu soruya cevap, basitlik, kısalık ve bakım rahatlığı bağlamında *mümkün olabildiğince az* olmalıdır. Nasıl olsa bir queue'ya birden fazla gönderici mesaj gönderebilir ve bir queue'dan birden fazla okuyucu mesaj okuyabileceği için, queue sayısında yapılan bir azaltma programın doğru işleyişi açısından bir yan etkiye sebebiyet vermez. Şekil 4.6 üzerinde çift queue ile kurulmuş bir Command yapısı görüyoruz.

Bu yapıda, istekleri göndermek için bir queue, cevapların geri gönderilmesi için başka bir queue ayırılmıştır. `CommandListener`, istek queue'su üzerinden dinleyici olarak set edilmiştir, ve bu queue'ya yazılan her Command'i anında alıp, üzerinde `execute` işlemini çağıracak ve cevabı (yâni aynı Command'i) ikinci queue'ya yazacaktır. Web tarafındaki JVM birden fazla kullanıcıyı idare

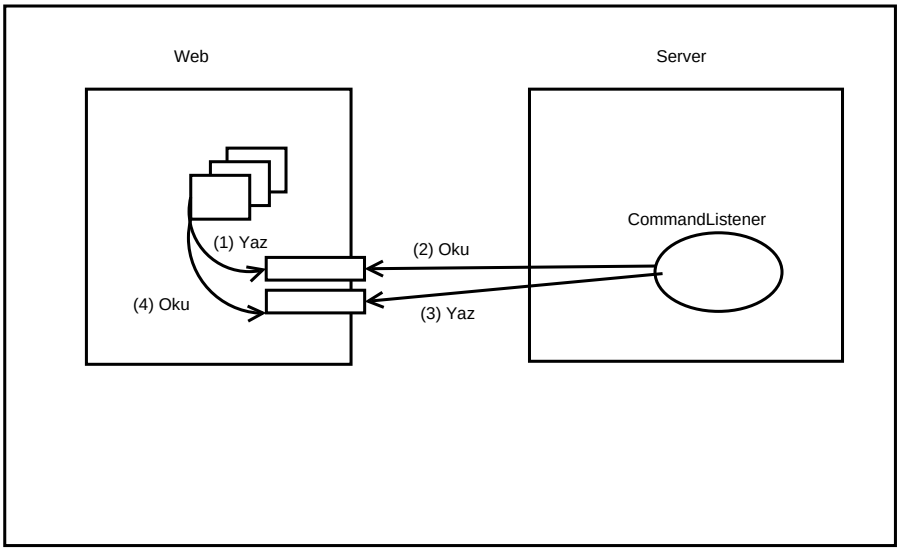
edebileceği için herhangi bir anda, eşzamanlı birçok kullanıcının istek Command'leri istek queue'su üzerinde dizilmiş olur. `CommandListener` bu istekleri teker teker, o queue'da olan sırasıyla işler.

Bu yapıya alternatif olarak, meselâ, her mesaj gönderilirken bir geçici queue yaratmak, bu cevap queue'sunu JMS mesajının içine "cevabı buraya yaz" babında eklemek, ve cevabı o geçici queue'ya almak düşünülebilir. Fakat bu seçeneğin performans olarak kötü yan etkileri olacaktır: Her mesaj için yeni queue yaratıp yoketmek, sistemin bütününü yavaşlatır.

Çift queue mimarisinin diğer bir faydalı tarafı, sistemin ne kadar yüklü olduğunu sadece iki queue'da *kaç tane mesaj birikmiş* olduğuna bakarak anlayabilecek olmamızdır. Eğer queue'larda, özellikle cevap queue'sunda fazla mesaj birikmiş ise, `CommandListener` tarafının yükü kaldıramadığını anlar, ve birden fazla işleyici JVM devreye sokabiliriz. Bunun için hiçbir ekstra ayarlama yapmamız gerekmeyecektir. Aynı queue üzerindeki her okuyucu bir mesajı okurken, diğerlerinin önüne geçmez. JMS okuyucu kurallarına göre bir mesajı alan onu *kapmış* addedilir, ve o mesajın işleyicisi o olur. Bu JMS özelliği, yük dağıtımı açısında biçilmiş kaftandır.

4.7.2 Kullanıcıları Ayırt Etmek

Şekil 4.6 ve 4.7 üzerinde gördüğümüz yapıya bir daha bakalım: Eğer hiçbir ek işlem yapmazsak, bu mimari çalışacak mıdır? Hayır, yapmamız gereken bir



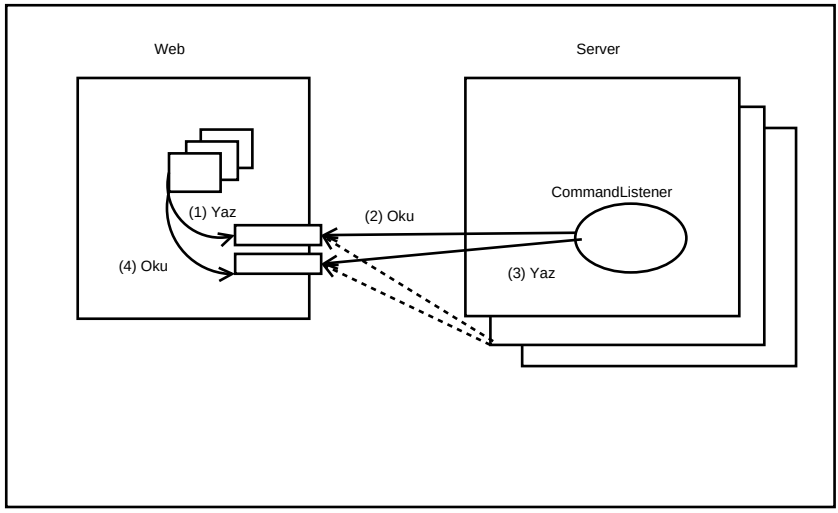
Şekil 4.6: JMS Üzerinden Command Mimarisi

ek işlem daha kaldı: Web tarafına gelen cevap mesajlarını, sadece ve sadece o isteği göndermiş olan kullanıcıya göndermek. Eğer bunu yapmazsak, birçok kullanıcıyı idare etmek için yazılmış Web tarafı, birçok kullanıcının mesajını aynı queue'ya koyacak, ve birçok kullanıcının cevaplarını başka bir queue'da birikmiş ve karışık olarak almaya çalışacaktır. İşlem doğruluğu olarak bu problemlere yolaçacaktır.

Peki her kullanıcıya sadece ilgilendiği mesajları nasıl verebiliriz? 4.6.7 bölümünde işlediğimiz filtreleme tekniğini hatırlayın: Eğer her kullanıcıya özel bir kimlik no'su bulabilirsek, bu kimlik değerini her mesaj üzerine *mesaj parametresi* olarak set edebiliriz, ve `CommandListener` geri gelecek cevap üzerine aynı parametreyi tekrar koyar. Web tarafındaki dinleyiciye de, bu mesaj parametresi üzerinden filtreleme yaparsa, böylece her kullanıcının sadece kendisi için olan cevap mesajlarını almasını sağlamış oluruz.

Web mimarisinde, her kullanıcıya özel bir kimlik bulmak çok basittir. Bu kimlik, `sessionId` değişkeninden başkası değildir. Bir Struts Action içinde `sessionId`'yi almak için `request.getSession().getId()` çağrısını yapmak, ve mesaj üzerine bu kimliği `setStringProperty` çağrısı ile koymak yeterli olacaktır.

Eğer çağıran taraf bir zengin kullanıcı ise (meselâ Swing), o zaman her bağlanan kullanıcının *ayrı bir JVM'i* var demektir. Böyle bir yapıda Message Broker da *ayrı ve tek* bir JVM içinde çalışıyor olacaktır. O zaman mesaj ayrıştırma problemini şöyle çözeriz: Zengin önyüz ortamında `sessionId` yoktur, ama her kullanıcı için bir rasgele kimlik üreterek mesaj üzerinde set etmek çok kolaydır. Rasgele kimlik üretmek için



Şekil 4.7: JMS Bazlı Mimariyi Ölçeklemek

```
java.util.Random generator = new java.util.Random();
int id = generator.nextInt(1000); // 1000'den küçük id'ler üret
```

çağrısı kullanılabilir.

4.7.3 Kodlar

JMS üzerinden Command mimarisinin kodlarına gelelim (bu kodların tamamlanmış hâlini, `CarJMS` projesi içinde bulabilirsiniz).

İlk önce iki queue'yu tanımlayalım. İstek Command'lerinin konduğu queue ismi, `kitapWebQueue`, cevap Command'lerinin okunduğu queue ise `kitapServerQueue` olsun.

Queue Tanımları

Liste 4.18: jboss-service.xml

```
<server>
  ...
  <mbean code="org.jboss.mq.server.jmx.Queue"
   name="jboss.mq.destination:service=Queue,name=kitapServerQueue">
    <depends optional-attribute-name="DestinationManager">
        jboss.mq:service=DestinationManager
    </depends>
    <attribute name="MessageCounterHistoryDayLimit">-1</attribute>
  </mbean>
  <mbean code="org.jboss.mq.server.jmx.Queue"
   name="jboss.mq.destination:service=Queue,name=kitapWebQueue">
    <depends optional-attribute-name="DestinationManager">
        jboss.mq:service=DestinationManager
    </depends>
    <attribute name="MessageCounterHistoryDayLimit">-1</attribute>
  </mbean>
</server>
```

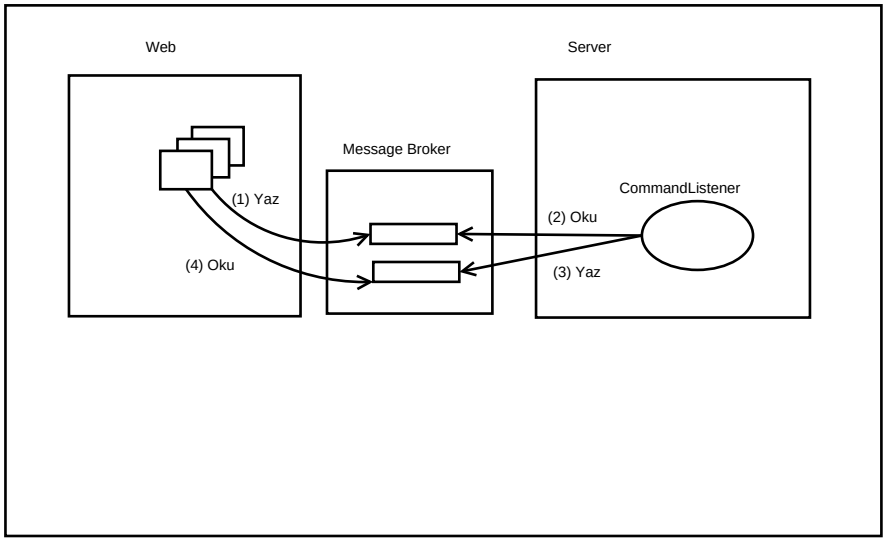
Bu queue'ları *Web tarafında* tanımladık (yâni üstteki `jboss-service.xml` dosyası `CarsJMS/WebClient` altında), çünkü, yük dağıtım amaçlı birden fazla işleyici (dinleyici) JVM başlatırsak, yeni dinleyicilerin (işleyicilerin) aynı queue'lara, yâni aynı JVM'e gitmesi lazımdır. Eğer queue'lar servis tarafında olsa idi, her başlatılan işleyici JVM, yeni queue'lar başlatmış olurdu ve bu uygun olmazdı.

Alternatif fiziksel yapı olarak, ayrı bir JVM başlatarak Message Broker'ı bu ayrı JVM'de tutmayı seçebilirsiniz. Biz basitlik ve idare edilebilirlik açısından queue'ları Web tarafına koymayı seçtik. Şekil 4.8 üzerinde bu alternatif fiziksel yapıyı görüyorsunuz.

Bağlantı, Oturum, Gönderici

Command'leri göndermekle yükümlü Web tarafında, gönderilen ve okunan iki queue için birer JMS bağlantısı, JMS oturumu, JMS gönderici ve JMS okuyucusu yaratmamız gerekiyor. Her kullanıcı için bu işlemi bir kez yapmamız yeterli, yâni her mesaj için bu işlemlerin tekrar tekrar yapılmasına gerek yok. Bu sebeple, gereken JMS referanslarını bir kez yaratıp, her kullancıya ait web oturumu üzerinde değişkenler olarak tutabiliriz. Böylece, gerekince bu nesnelere erişmek çok hızlı olacaktır. JMS referanslarına ilk ihtiyaç duyulduğu anda otomatik olarak yaratabilecek `ServiceReferencesFilter` adında bir filtre içinde bu işlemleri yapabiliriz.

```
public class ServiceReferencesFilter implements Filter {

  private Logger logger = Logger.getLogger("appLogger");

  public void init(FilterConfig filterConfig) throws ServletException {
  }

  public void doFilter(ServletRequest request,
                   ServletResponse response,
                   FilterChain chain) throws IOException,
                                             ServletException
  {
      HttpSession session = ((HttpServletRequest) request).getSession();
```



Şekil 4.8: Message Broker Ayrı JVM Üzerinde

```
try {
    if (session.getAttribute("senderSession") == null ||
        session.getAttribute("sender") == null) {

        QueueConnectionFactory qFactory = null;

        // ...
        // Context'i al
        // ..

        QueueConnectionFactory qcf =
            (QueueConnectionFactory)ctx.lookup("ConnectionFactory");
        QueueConnection qc = qcf.createQueueConnection();
        Queue queueServer =
            (Queue) ctx.lookup("queue/kitapServerQueue");
        QueueSession queueSession =
            qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        QueueSender sender = queueSession.createSender(queueServer);

        session.setAttribute("sender", sender);
        session.setAttribute("senderSession", queueSession);
        session.setAttribute("senderQueueConnection", qc);
        logger.info("Sender and Session Created");
    }


    if (session.getAttribute("receiverSession") == null ||
        session.getAttribute("receiver") == null) {

        // ...
        // Context'i al
        // ..

        QueueConnectionFactory qcf =
            (QueueConnectionFactory)ctx.lookup("ConnectionFactory");
        QueueConnection qc = qcf.createQueueConnection();
        Queue queue = (Queue) ctx.lookup("queue/kitapWebQueue");
        QueueSession queueSession =
            qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
        String filter =
              "sessionId = '" +
              ((HttpServletRequest)request).getSession().getId() +
              "'";
        QueueReceiver receiver = queueSession.createReceiver(queue,
                                                    filter);
```

```
            session.setAttribute("receiverSession", queueSession);
            session.setAttribute("receiver", receiver);
            session.setAttribute("receiverQueueConnection", qc);
        }

    } catch (Exception e) {
        e.printStackTrace();
        logger.error("",e);
    } // end of try-catch

    chain.doFilter(request, response);
  }

  public void destroy() { }
}
```

Her kullanıcının mesajlarını ayırtedebilmek için filtre kullanacağımızdan bahsetmiştik. Okuyucu nesneyi yaratırken, bu filtreyi de yukarıda yaratmış olduğumuzu görüyorsunuz.

JmsHelper

Web tarafında işimiz neredeyse bitti. Struts Action'lar içinde Command yaratıp queue üzerine koymak ve cevabı okuma işlemlerinin kodlamasını kolaylaştırmak için, bir yardımcı metot yazmaktan başka bir iş kalmadı. Bu metotu,`JmsHelper` adında bir nesne içine koyalım, ve çağrıyı `static` olarak tanımlayalım. Bu yardımcı çağrıya `sendCommand` ismi verilecek, ve parametre olarak bir command alıp, geriye cevap Command'ini geri getirmekten başka bir görevi olmayacak.

```
public class JmsHelper {

    private static Logger logger = Logger.getLogger("appLogger");

    public JmsHelper() { }

    public static Command sendCommand(HttpServletRequest request,
                                      Command command) {
        Command resCmd = null;

        try {
            HttpSession session = ((HttpServletRequest) request).getSession();
            QueueSender queueSender =
                (QueueSender)session.getAttribute("sender");
            QueueSession queueSession =
                (QueueSession)session.getAttribute("senderSession");

            QueueReceiver queueReceiver =
                (QueueReceiver)session.getAttribute("receiver");
```

```
            QueueSession receiverSession =
                (QueueSession)session.getAttribute("receiverSession");
            QueueConnection receiverQueueConnection =
                (QueueConnection)session.getAttribute("receiverQueueConnection");

            ObjectMessage objectMessage = queueSession.createObjectMessage();
            objectMessage.setObject(command);

            objectMessage.setStringProperty("sessionId",
            request.getSession().getId());
            queueSender.send(objectMessage, DeliveryMode.NON_PERSISTENT, 4, 0);

            receiverQueueConnection.start();
            ObjectMessage resObj = (ObjectMessage)queueReceiver.receive();
            resCmd = (Command)resObj.getObject();

        } catch (Exception e) {
            logger.error("",e);
        } // end of try-catch

        return resCmd;
    }
}
```

Yardımcı metotun içeriğine bakarsak, parametreden olarak gelen Command nesnesinin `setObject` ile bir `ObjectMessage` üzerinde set edildiğini görüyoruz. Daha sonra JMS `send` ile Command gönderilmekte, ve hemen arkasından cevap senkron olarak `receive` çağrısı üzerinden beklenmektedir. Gönderilen mesaj üzerine bir `sessionId` parametresi konulmasını tabii ki unutmadık, ve zaten okuyucu nesnemizde aynı `sessionId` üzerinden filtreleyecek şekilde yazılmıştı.

Hatalar

JMS Command mimarisi mesaj gönderen ve gönderilen tarafları birbirinden koparıp asenkron hâline getirdiği için, servis tarafında exception atıldığı zaman bu hatanın bir şekilde mesaj gönderen tarafına aktarılması gerekir. Bu aktarım, EJB ya da RMI örneğinde olduğu gibi metot çağrısının bir parçası olamaz, çünkü JMS Command mimarisinde *metot çağrısı yoktur*. O zaman bir hatayı nasıl bildireceğiz?

Exception'ı Command nesnesinin içine gömebiliriz. `Command` üst sınıfından miras alan bir `ExceptionCommand` sınıfı yazarız, ve tüm Command class'ları `Command` yerine `ExceptionCommand` sınıfından miras alır.

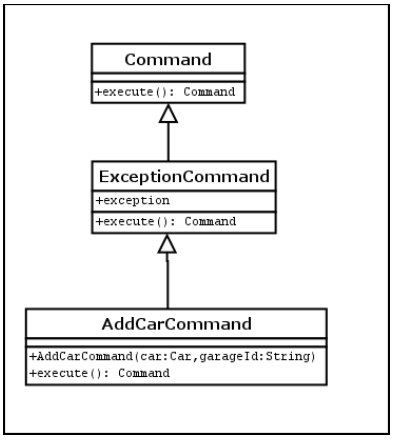
```
public abstract class ExceptionCommand implements Command {

    Throwable exception;
```

```
    public Throwable getException() {
        return exception;
    }

    public void setException(Throwable newException) {
        this.exception = newException;
    }
}

public class AddCarCommand extends ExceptionCommand { .. }
```

`ExceptionCommand` üzerinde bir `Throwable` referansı olduğunu görüyoruz (her `Exception` aynı zamanda bir `Throwable` tipidir). Artık servis tarafında bir hata alındığı zaman, `CommandListener` bu hatayı cevap Command'i içine gömerek cevap queue'suna yazarak gönderen tarafa geri verebilecektir.

Bağlanan taraf eğer bir hatanın varlığını kontrol etmek istiyorsa, cevap queue'sundan aldığı Command nesnesinin içine `getException` ile bakarak, hata tipini `instanceof` ile kontrol eder, ve `try catch` mantığının bir benzerini gerçekleştirebilir.

```
UpdateCarCommand cmd = new UpdateCarCommand(car, garageId);
UpdateCarCommand resCmd =
     (UpdateCarCommand)JmsHelper.sendCommand(request, cmd);

if (resCmd.getException()
    instanceof
    org.hibernate.StaleObjectStateException)
{
    String id = (String)daf.get("licensePlate");
```



Şekil 4.9: Hata Taşıyıcı Command

```
    ...
}
```

CommandListener

`CommandListener` bir JMS dinleyicisi ve Command işleyicisidir. İstek queue'su (`kitapWebQueue`) üzerine gelen her mesaj `CommandListener`'ın `onMessage` metotuna düşer. Bu mesaj içinde bir Command nesnesi olacaktır. Bu nesne alındıktan sonra, üzerinde `execute` çağırılır. Eğer bir hata olmuş ise, exception yakalanarak command'in üzerine yazılır. Cevap için yeni bir `ObjectMessage` oluşturulur, ama istek için gelen aynı Command, cevap olarak `ObjectMessage` üzerinde konur. Bu Command, cevap queue'su üzerinden `send` ile gönderilir.

Bitmiş kodları, `CarJMS/Server` projesi altında bulabilirsiniz.

Liste 4.19: CommandListener.java

```
public class CommandListener implements MessageListener {

    private Logger logger = Logger.getLogger("appLogger");

    QueueSender sender = null;
    QueueSession queueSessionS = null;
    Queue queueS = null;;

    public CommandListener() throws Exception {

        QueueConnectionFactory qFactory = null;

        // ...
        // Context'i al
        // ..

        QueueConnectionFactory qcf =
            (QueueConnectionFactory)ctx.lookup("ConnectionFactory");
        QueueConnection qc = qcf.createQueueConnection();
        Queue queue = (Queue) ctx.lookup("queue/kitapServerQueue");
        QueueSession queueSession = qc.createQueueSession(false,
                                                Session.AUTO_ACKNOWLEDGE);

        QueueReceiver qReceiver = queueSession.createReceiver(queue);
        qReceiver.setMessageListener(this);
        System.out.println("Listening on command queue .........");
        qc.start();

        // bir de cevapların yazılacağı queue'ya session aç
        queueS = (Queue) ctx.lookup("queue/kitapWebQueue");
        queueSessionS = qc.createQueueSession(false,
```

```
                                        Session.AUTO_ACKNOWLEDGE);
        this.sender = queueSessionS.createSender(queueS);
        System.out.println("Ready to send ..........");
    }

    public void onMessage(javax.jms.Message message) {

        ExceptionCommand command = null;
        ObjectMessage sendObjectMessage = null;
        try {
            ObjectMessage objectMessage = (ObjectMessage)message;
            command = (ExceptionCommand) objectMessage.getObject();

            sendObjectMessage = queueSessionS.createObjectMessage();
            sendObjectMessage.setStringProperty
                                ("sessionId",
                                 message.getStringProperty("sessionId"));
            command.execute();
            HibernateSession.commitTransaction();
        } catch (HibernateException ex) {
            HibernateSession.rollbackTransaction();
            if (logger.isDebugEnabled()) logger.debug("rolled back");
            command.setException(ex);
            logger.error("hibernate exception",ex);
        } catch (Exception ex) {
            HibernateSession.rollbackTransaction();
            command.setException(ex);
            logger.error("exception",ex);
        } finally {
            HibernateSession.closeSession();
            if (logger.isDebugEnabled()) logger.debug("session closed");
        }

        try {
            // command'i mesaj üzerine koy
            sendObjectMessage.setObject(command);

            // cevap queue'su üzerinden geriye yolla
            sender.send(sendObjectMessage,
                 DeliveryMode.NON_PERSISTENT,
                      Message.DEFAULT_PRIORITY,
                      0);
        } catch (JMSException ex) {
            logger.error("",ex);
        }
    }
}
```

Bölüm 5

Performans, Ölçeklemek

Bu Bölümdekiler

- Yük testleri ve JMeter
- Bir uygulamanın tıkanma noktalarını bulmak
- Performans iyileştirmeleri
- Ölçeklemek

Güzel işleyen bir uygulamayı tanımlamak için, o "programın hatasız çalışması" artık yeterli bir kıstas olmamaktadır. Günümüzde doğru ama yavaş çalışan bir sistem kullanıcılarını soğutacağı için, aynen yanlış çalışan bir sistem gibi istenmeyen bir sonuç ürünüdür.

Tarihsel olarak kurumsal programcılık (IT) dünyasında uzun süredir *birikmiş* bir yazılım ihtiyacı (application backlog) durumu mevcut idi. Bu durum, programcıların acele yazdıkları sistemlerini sadece doğruluk açısından test ederek, performans konularını sonraya bırakmalarına sebebiyet vermiştir. Fakat müşteri ile direk iletişimde olan ve birebir satma amaçlı olan e-ticaret sistemlerinin yaygınlaşması ile bir paradigma değişikliği meydana geldi. Artık bir Web sistemi müşterisini yavaş bir sistem ile bekletmek istemeyecekti, çünkü rakibi "bir tıklama mesafesi kadar" yakındaydı. Böylece günümüzdeki hem doğruluk hem performans amacı güden senteze gelmiş oluyoruz. Yeni yaklaşıma göre performans analizi/iyileştirmesi, sadece sonda yapılan, ve gerektiği anda yemek tarifi gibi bir araya konan bazı ufak "numaralar" toplamı olmamalıdır. Performans konusu, aynen bakımı rahat kod yazmak için yazılım mühendisliği şemsiyesinin altılnda oluşturulduğu gibi, bir yöntem, bir süreç ve düşünüş şekli hâline gelmelidir.

5.1 Kavramlar

Bir sistemin performans durumu ve gereklilikleri hakkında konuşurken beş çok basit ana kavramı kullanmak yeterlidir [7, sf. 42]. Bunlar:

- Yük (workload)
- Cevap zamanı (response time)
- Üretilen iş (throughput)
- Kaynak kullanımı (resource utilization)
- Kaynak servisleme zamanı (resource service times)

Hesap **kaynaklarından** (computing resources) oluşan bir bilgisayar sistemi, bir **yük** üzerinde çalışmaya başladığı zaman dışarıdan görülebilen ölçümler **cevap zamanı** ve **üretilen iş** olacaktır, ve bu ölçüler sistemin dışa dönük performans ölçütleridir. Sistemin iç performansını târif eden ölçümler **kaynak kullanımı** ve **kaynak servis zamanı**, dışarıdan görülen performansa bir teknik açıklama sağlarlar.

Cevap zamanı en basit hâliyle, tek bir işi yapmak için harcanan zamandır. Her iş tipinin değişik cevap zamanları olabilir. **Üretilen iş** ise bir sistemin toplam, ya da işlem *bölü* zaman (sayı/saniye -rate-) olarak ne kadar iş yaptığının göstergesidir. Cevap zamanı ve yapılan iş ölçümleri bir sistem üzerindeki yük ile doğrudan bağlantılıdır. Belli bir kırılma noktasından sonra

sistemimiz çok fazla yükü kaldırmayıp, cevap zamanı ve üretilen iş ölçümlerinde düşüş yaşayabilir. Ya da, belli bir cevap zamanını kaldırmak için yazılmış bir sistem, üzerindeki yük arttıkça üretilen iş ölçümünde artış görüp, cevap zamanında hiç artış görmeyebilir (bu, performansı iyi olan bir sisteme örnektir). Yâni cevap zamanı ve üretilen iş ölçümleri her zaman bir yük çerçevesi (context) içinde telâfuz edilmelidir.

Kaynaklar arasında mikroişlemci (CPU), sabit diskler (hard drive), bellek (memory) ve ağ (network) bağlantısı sayılabilir.

Mikroişlemci sistemimizin en hızlı birimidir ve bir bilgisayarın kalbidir. Her türlü veri alısverişi (I/O), program işletme ve hesap yapma gibi hayati işleri halleder. Bu şekilde bir hayati görevi olması sebebiyle, genellikle yavaş işleyen sistemlerde ilk suçlanan birim mikroişlemci olur. Fakat bir final analize atlamadan önce, tüm faktörlere bakmamız gerekir: Belki mikroişlemci çok fazla ama aynı türden işlemi aynı anda işlemeye mecbur bırakılmıştır, yâni mikroişlemci zamanı için bir *yarış* sözkonusudur. Ya da, mikroişlemci veri alışverişi ile de sorumlu olduğu için ve kurumsal sistemler genellikle IO bağımlı (IO bound) oldukları için, belki de mikroişlemci bir IO aracının boşalmasını bekliyor ve genellikle boş (idle) duruyor olabilir. Bu yüzden, mikroişlemci temelli olduğunu sandığımız bu problemi çözmek için yeni bir mikroişlemci eklersek, sorunumuzu çözülmeyecektir. Bu sefer yeni mikroişlemci de boş beklemeye başlayacaktır.

Bellek CPU'nun hatırlamak istediği verileri, işlemek istediği işlemleri (instructions) tuttuğu bir depodur. Uçucu bellek (RAM) kalıcı bellekten (disk) her zaman daha hızlıdır fakat bellek, diğer tüm kaynaklar gibi, sınırlıdır. Eğer uygulamamız elde mevcut olandan fazla bellek istiyorsa, modern işletim sistemleri disk sistemini bellek gibi kullanabilme, sayfalama (paging) yeteneğine sahiptir. Fakat uygulamamızın çok fazla sayfalama yapmaya başlaması da istenilen bir şey değildir, çünkü çok hızlı olduğunu bildiğimiz bellek için yazdığımız kodlar, artık diske giderek daha yavaş bir ortamda çalışmaya başlamıştır. Her modern işletim sistemi, yaptığı sayfalama istatistiklerini paylaşma yeteneğine sahiptir. Bu istatistikleri kullanarak, eğer çok yüksek sayfalama görürsek, o zaman uçucu belleğin gereğinden fazla kullanım (utilized) gördüğünü anlayabiliriz. Çözüm ya bellek eklemek, ya bellek kullanımını azaltmaktır (yükü yeni makinalara aktararak, ya da kodu değiştirerek).

Diskler, kalıcı olmasını istediğimiz iletişimin (IO) hedefidir. Bir disk, manyetik *disklerden* oluşur (ismi de buradan gelir) ve program çökmesi, elektrik kesintisi gibi kazasal durumlarda bilgimizi koruyacak olan ortam olma görevini yapar. Kaybolmasını istemediğimiz verilerin diskte tutulması, bu yüzden program doğruluğu açısından büyük önem taşır. Birden fazla diski birarada kullanmak istiyorsanız, RAID adı verilen disk yöntemi ihtiyacınızı karşılıyabilir. RAID, **R**edundant **A**rray of **I**nexpensive **D**isks (Gereğinden Fazla Alınmış Ucuz Disk Kümesi) kısaltmasından gelir, ve depolanmak istenen veriyi birden fazla parçaya bölerek (striping), bu parçaları her ucuz disk üzerinde parça parça tutarak veriye erişimi hızlandırmayı amaçlar. RAID bir önbellek (cache) ile kul-

lanırsa daha ideal bir ortam hâline gelir, çünkü önbelleksiz olarak kullanımda ufak ve noktasal erişimlerin (random access) sürekli tüm disklere giderek paralel işlemenin avantajlarını yoketmesi mümkündür. Bu sebeple RAID çözümünü aldığınız firmanın teknik altyapısını iyi tanımanız gerekmektedir.

Büyük miktardaki veri transferleri doğal olarak birkaç diske dağılacağından, bu tür kullanım için RAID'in faydası olacaktır.

Ağ birden fazla bilgisayarın birbiri ile haberleşmesi için gereken altyapıdır, ve Internet + servis tarafında bir küme mimarisi durumunda performansı etkileyecek önemli faktörlerden biridir. Ağ servisleme zamanı üzerinde, ağda kullanılan iletişim protokolünün (TCP/IP, UDP) büyük etkileri vardır, bu sebeple mimari tasarımımızda bilinmesi gereken önemli bir faktördür.

5.2 Yaklaşım

İyi perform eden bir sistemin performans kriteri, her zaman müşteriye bağlı olan bir detaydır. Bu yüzden projeye başlamadan önce, sistem hakkında "istenilen performans ölçülerinin" müşteriden alınması gerekecektir. Bu ölçülerin alınmasını aynen uygulamanın özellik listesini aldığımız ciddiyette almalıyız çünkü aynen bir programın doğruluğunu özellik listesinin ne olduğu belirlediği gibi, ne kadar hızlı olduğunu da yine müşterinin belirttiği performans kriterleri belirleyecektir.

İyi perform eden bir sistemin teknik gerçekleştirimi için, düşünce şeklimizin iyi performans kavramı üzerine kurulmasından bahsettik. O zaman, böyle bir sistemi yazarken dikkat etmemiz gereken kuralları ve aklımızda tutmamız gereken prensipleri sıralayalım: Bu kurallar ve prensipler, daha ilk kodu yazdığımız andan itibaren dikkat edilecek kurallardır.

- Veri tabanına ne zaman gidersek, orada yapılan işlemi (transaction) ufak tutmalıyız.
- Veri tabanları *kümeler* ile çalısmayı severler. Örnek olarak 100 tane SQL `UPDATE` yapan sorgular yerine, 100 satırı tek bir SQL ile güncelleyen bir sorgu veri tabanları tarafından daha çabuk işletilir.
- Mimarimizdeki fiziksel katmanlar, gereğinden fazla olmamalıdır. Network'e her çıkışımızda belli bir performans bedeli ödüyoruz.
- Sık erişilen ve az değişen birimler, önbelleğe konmalıdır (bir veri nesnesi, veri tabanı bağlantısı gibi).
- Kod bakımı açısından çok kullanıcılı sistemler, tek bir makina için yazılıyormuş gibi yazılabilmelidir. Ölçekleme, uygulama servis paketinin ayarlarını değiştirmek suretiyle yapılan, programlama dışı bir işlem olmalıdır. Ölçekleme zamanı gelince, ek bir makina (donanım) koyarak sistemin işlem gücünü arttırabilmeliyiz. Bu durum, kağıttan insan zinciri (Şekil 5.1) yapmaya benzer.

Tüm bunlara rağmen, eğer uygulamamızın yavaş işlediğini gözlemliyorsak, o zaman detaylı analiz yapma zamanımız gelmiştir.

5.2.1 Analiz

Cevap zamanı ve üretilen iş ölçüleri her zaman bir yük çerçevesinde anlamlı olduğu için, programımızın performansını analiz etmek için yapay bir yük yaratmalıyız. Bu tür bir teste yük testi adı verilmektedir. Yük testi, tek ya da daha fazla makinanın yüzlerce hattâ binlerce kullanıcıyı taklit (simüle) etmesi ile servis tarafındaki kodlarınız üzerinde bir kullanıcı trafiği yaratması, ve bizim de bu reaksiyonları ve performans ölçütlerini toplamamıza verilen isimdir. Apache JMeter bu tür bir taklit yükü oluşturacak araçlardan biridir, bu ürünü 5.3.1 bölümünde daha yakından tanıyacağız.

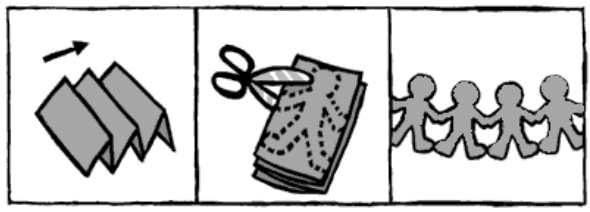
Yük testine başlamadan önce, aklımızda, projenin başında aldığımız "beklenen performans ölçülerinin" olmalıdır. Çünkü yük testinin sonuçlarını topladıktan sonra, işimizin bitip bitmediğini belli edecek ölçüler bunlar olacaktır. Bu beklenen ölçüler projenin başında müşteri tarafında belirtilmiş ölçüler olacaktır. Meselâ, "sistemin xx kadar eşzamanlı aktif kullanıcıya hizmet verebilmesini istiyorum, ve her sayfanın cevap zamanı (response time) yy saniye altında olmalı" gibi.

Yük testi sırasında aynen "sonuç ortamında (production) kullanılacağı miktarda" test verisi kullanmak çok önemlidir. Veri miktarı ve çeşidi her zaman gerçek sonuç ortamını yansıtacak şekilde olmalıdır. Aynı şekilde test *donanım* ortamı da, sonuç ortamını mümkünse bire bir, değilse çok yakın bir şekilde yansıtmalıdır.

Artık tıkanma noktalarını bulmaya başlayabiliriz. Tıkanma noktaları, uygulamamızın bir yerinde *kaynaklar üzerinde* şunlardan birinin göstergesidir:

- Kaynak kullanım fazlalığı (high utilization)
- Kaynağa erişim yarışı (fazlalılığı) sırasında kaynağın boğulması

O zaman, uygulamamız işlerken ilk önce kaynakları takip edip, kullanım durumu ve bekleme zamanlarını takip etmeliyiz. Her kaynağın kendine has bir takip aracı vardır. CPU için `vmstat`, disk için `vmstat` ve `iostat` gibi.



Şekil 5.1: Tek Makina Gibi Yaz, Otomatik Çoğalt

İyi işleyen bir sistemi (uygulama artı işletim sistemi) analiz etmek için, şu şekilde bir irdeleme yapabiliriz: Sistem optimal sayıda kullanıcının isteklerini optimal bir hızda karşılıyor iken, mikroişlemci kullanımı %90 üzerinde olmalıdır. Bu, işlemcinin *meşgul olduğunu* gösterir ve bu iyiye bir işarettir. Demek ki uygulamamız, sistemin mümkün olan tüm gücünü kullanmaktadır.

Bu durumun tersi de olabilir: CPU kullanımı %100'de, fakat desteklenebilen kullanıcı sayısı bekleneden az ise, demek ki uygulamamızın bazı bölümleri gereksiz bir şekilde CPU zamanı yemektedir. Hangi kod parçasının bu zamanı yediğini anlamak için Profiler ve benzeri araçlarla daha detaylı analize devam etmeliyiz.

Peki, ne kadar eşzamanlı kullanıcı/işlem için ne kadar bellek kullanmak gerekir? Bunun cevabını **Kurumsal Web'in Altın Kanunu** veriyor. Şimdiye kadar kurduğumuz Kurumsal Web uygulamalarının işleyişine ve kapasitesine bakarak geliştirdiğimiz bu oran, şöyledir:

> "1 Ghz hızındaki mikroişlemci üzerinde, 1 GB JVM belleği ile çalışan uygulamamız, aynı anda 500 oturumu (session) idare edebilir".

Bu ayarları yaptıysak ama sanal bellek (virtual memory) çok fazla kullanılıyor ise, Java Sanal Makinası için ayırdığımız bellek, işletim sisteminin elinde olduğundan daha fazla bir değere ayarlanmış olabilir; Bu yanlış ayarı telâfi etmek için işletim sistemi sürekli sanal belleğe giderek, diskte ve diskten fazla okuma/yazmaya sebebiyet vermektedir. Bildiğimiz gibi işletim sistemleri ellerindeki mevcut gerçek bellek kapasistesinden çok daha fazlasını, disk'i sanal bir bellek gibi kullanarak yaratabilirler. Tabii bunun yan etkisi, daha yavaş çalışan bir uygulama olacaktır, çünkü disk'e yazmak ve oradan okumak, gerçek bellekten okuyup yazmaktan daha yavaştır.

Bir kurumsal uygulamanın disk'e gidişi her zaman veri tabanı üzerinden olur, bu sebeple veri tabanının olduğu makinadaki kaynakları ayrı olarak takibe almanız gerekebilir. Eğer bu makinada gereğinden fazla diskten okuma ve yazma var ise, veri tabanının önbelleğini arttırma gündeme gelebilir, ya da uygulamanız tarafından veri tabanına gönderilen SQL komutlarında azaltma ya da düzeltme yapılmalıdır.

Uygulamamızın üzerinde çalıştığı makinaları takip etmeden önce, bilmemiz gereken diğer bir bilgi, aynı sistemin *üzerinde yük yok iken* nasıl çalıştığıdır. Çünkü, belki de suç uygulamada (ve uygulama servisinde) değil, uygulamanın üzerinde çalıştığı makinanın ayarlarındadır. Bir işletim sisteminin sıfır yük altında nasıl çalıştığını bilmiyorsak, yük altında nasıl çalıştığına dair bir karşılaştırma yapmamız zaten zor olacaktır, bu sebeple hiç yük yok iken sistemimizin nasıl çalıştığını not etmemiz gerekmektedir.

5.3 Analiz Araçları

5.3.1 JMeter

Yük testlerimiz için JMeter[1] aracı kullanabiliriz. JMeter, bir Web uygulamasının üzerinde değişik türden yükler oluşturmak için gerekli tüm özelliklere sahiptir.

JMeter ile yük testi yapabilmek için, web uygulamamızı JMeter'a *aynen bir kullanıcı kullanıyormuş gibi* kullandırtmamız lâzımdır. Yâni JMeter, uygulamamıza bağlanıp, bir kullanıcı gibi sayfaları yüklemeli, biraz veri girmeli, birşeyler silmeli, ve uygulamadan bazı listeler almalıdır. Yâni JMeter, uygulamayı bir *sanal kullanıcı* gibi kullanmalıdır.

JMeter, web uygulamanıza sanal bir kullanıcı olarak gözükmek için, HTTP protokolunu kullanır. Bir JMeter'a verilen *test planı*, hangi sayfanın ne zaman, ve hangi bilgiyle çağırıldığını belirler. Bu bilgilerin ışığında test planını işleme koyduğunuz zaman, JMeter HTTP protokolu ile uygulanıza bağlanacak ve istediğiniz veriler ile sayfalarınızı gezmeye başlayacaktır. Eğer test planımızı dinamik veriler girebilecek şekilde ayarlamışsak, eşzamanlı kullanıcı sayısını istediğimiz kadar arttırabiliriz, ve böylece gerçek dünya şartlarında uygulamamızın nasıl davranacağını, nihai ortamda işleme konmadan önce, görebilmiş oluruz.

Test planı hazırlamak için, iki yöntemi takip edebilirsiniz. İlki, JMeter'a gerekli çağrı komutlarını elle eklemektir. İkincisi, *siz* web uygulamasını kullanırken yapılan tüm hareketlerinizi JMeter'a *kaydettirmenizdir*. İkinci yöntem, doğal olarak daha basittir, ve takip edilmesini tavsiye ettiğimiz yöntem budur. JMeter'in kaydettiği test planını kullanmadan önce biraz değiştirmeniz gerekecektir, fakat bu değişiklik çok büyük ölçekte olmayacaktır, ve herşeyin elle yapıldığı şartlardan çok daha basit olacaktır.

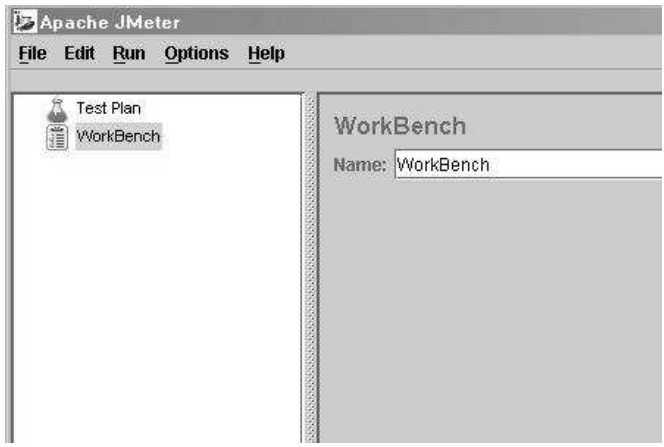
JMeter Kullanımı

JMeter programını kurmak için JMeter sitesinden indireceğiniz zip dosyasını herhangi bir dizinde açın. JMeter ana dosyasının `JMETER_DIR` altında olduğunu farz edersek, progamı başlatmak için `JMETER_DIR/bin/jmeter.bat` ya da `JMETER_DIR/bin/jmeterw.bat` dosyasına tıklamamız gerekiyor. JMeter ilk başladığında Şekil 5.2 üzerindeki gibi bir ekran göreceksiniz.

JMeter'in görsel kullanımı ilk başta değişik gelebilir. JMeter da bir görsel birimi sürükle/bırak (drag/drop) ile hareket ettirip yerine bıraktığınızda, size Şekil 5.3 üzerinde gösterildiği gibi bir menü ile bir seçim sunulacaktır. Gösterilen örnekte, Counter adlı JMeter görsel birimini `HTTP Request Defaults` adlı birimin üzerine bırakmışız.

Sorulan sorulardan eğer `Insert Before` seçilirse, sürüklediğiniz birim, üzerine bıraktığınız birim ile *aynı* seviyede, ama önce gelmek üzere pozisyonlanır.

[1] `jakarta.apache.org/jmeter`



Şekil 5.2: JMeter Açılış Ekranı

Şekil 5.3: Sürükle ve Bırak

`Insert After` seçilirse, yine aynı seviyede, ama *sonra* gelecek şekilde pozisyonlanma yapılır. `Add as Child` seçilir ise, bir birim öteki birimin çocuğu olacak şekilde düzenlenir. Her birimi her başka birim altına çocuk olarak eklemek legal değildir; JMeter yeni birimi sadece çocuk kabul eden diğer birimlerin altına atmanıza izin verecektir.

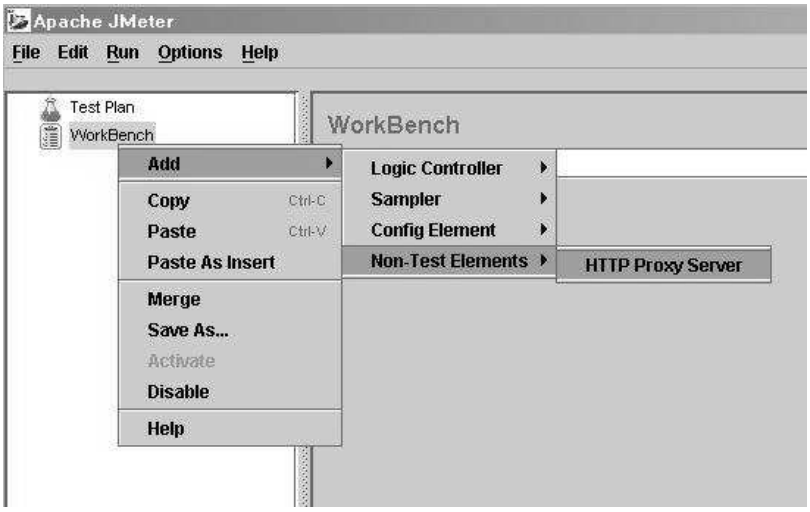
Senaryo Kaydetmek

Elle ya da üretilerek girilen test planları "Test Plan" altına gitmesini isteriz. Test planını üretebilmemiz için, *üzerinden* uygulamamıza bağlanacağımız bir proxy servisi yaratmamız gerekiyor. Proxy servisi, Web tarayıcılarımızdan bildiğimiz bir kavramdır. Internet'te bir siteye bağlanırken, genelde o siteye direk olarak değil, bir proxy üzerinden çıkarız, yâni web isteğini bizim için bir proxy (yer tutucu) gerçekleştirmiş olur. Bir yerel ağın Internet'e bağlantısı genellikle bir proxy üzerinden yapılır, meselâ bir şirketin yerel ağından dışarı çıkan herkes,

aynı proxy'i kullanır. Ağ güvenliği için bu şekilde bir kullanıma ihtiyaç vardır [8, sf. 85].

JMeter da, aynen yerel ağ için hazırlanan bir proxy gibi, bir proxy başlatabilir. Bunu yapma amacı, o proxy üzerinden web uygulaması test edilirken, yapılan tüm hareketleri, GET, POST işlemlerini gözlemleyebileceği bir geçiş noktasına ihtiyacının olmasıdır. Yâni birazdan hazırlamasını öğreneceğimiz JMeter Proxy Server, biz testimizi yaparken tüm web isteklerinin ve cevaplarının üzerinden yönlenlendirileceği bir geçiş noktası olacak.

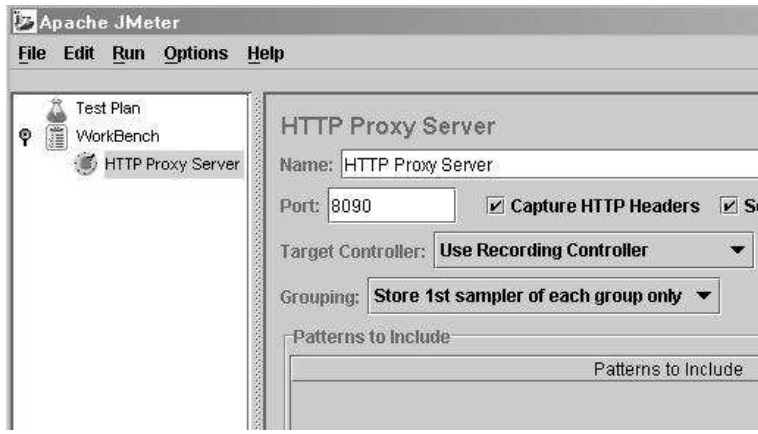
Proxy Server eklemek için, Workbench üzerinden sağ tıklama yaparak menüden `Add | Non-Test Elements | HTTP Proxy Server` seçeneğini seçin.



Şekil 5.4: JMeter İçin Proxy Server Yaratmak

Proxy'nin ayarlarını değiştirmek için, eklenen HTTP Proxy Server biriminin üzerinde tıklayarak, ayarlarını yapabiliriz. Meselâ Proxy Server'in hangi port üzerinden servis vereceğini tanımlamak istiyorsak, o port numarasını `Port` kutusundan girmeliyiz. Genelde, yerel bir JBoss'ta çalışan uygulamanın port numarası `8080` olacağı için, Proxy Server için verilen olağan değeri `8080`'i değiştirmemiz gerekiyor. Bu değer için `8090` kullanalım. Şekil 5.5 üzerinde bu ayarları görebiliriz.

Değişecek diğer ayar, `Target Controller` için "Use Recording Controller" ve `Grouping` altında "Store 1st sample of each group only" seçenekleridir. Bu seçeneklerden `Target Controller`, kaydedilen kullanıcı işlemlerinin "nereye kaydedileceğini" belirler. Biz bu kaydedilme işleminin `Test Plan` altında bir dinleyici tarafından dinlenip kaydedilmesini istiyoruz. Böylece test planımız, dinleme ve kaydetme bittikten sonra bizim için işletilmeye hazır bir test planını aynı JMeter penceresi içinde bekletiyor olacaktır.



Şekil 5.5: Proxy Ayarları

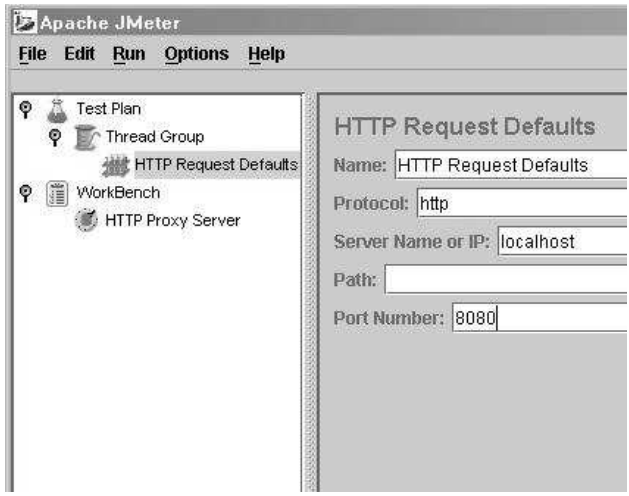
Önce test planının kullanıma hazır bir hâle getirmemiz gerekiyor. Ayarları öyle yapalım ki, test plan üretimi bittikten sonra, üzerinde ufak değişikliklerden sonra "İşlet" komutunu verebileceğimiz bir yapı hazır olsun. Bunun için öncelikle eklememiz gereken, `Thread Group` birimidir. Bu birim, yük testlerinin kaç Thread ve kaç kere işletileceğini kontrol eden bir birimdir. `Thread Group` eklemek için, `Test Plan` üzerinden sağ tıklama ile `Add | Thread Group` seçimini yapın.

Test planının hangi makinaya ve porta bağlanacağını kontrol edebilmek için, `Http Request Defaults` adında "her sayfa bağlantısı için aynı" olacak olağan değerleri belirleyen bir birim eklemeliyiz. Bu eklemeyi, biraz önce eklediğimiz `Thread Group` üzerinden `Add | Config Element | HTTP Request Defaults` seçimi ile yapabiliriz. Ekleme bittikten sonra JMeter ekranı Şekil 5.6 gibi gözükecektir.

`HTTP Request Defaults` değerleri şimdilik `localhost`, `8080` ve `http` olabilir. Kullanıcıyı kayıtlamayı `localhost:8080` üzerindeki bir uygulama üzerinden yapacağımız için, testi geri çalarken (replay) aynı makina ve port değerini kullandık. Eğer kaydedilmiş testleri başka bir makina, port'a doğru yönlendirmek istersek, bunun için test edilecek makinayı değiştirmek için `HTTP Request Defaults` üzerinden çok basit bir işlem olacaktır. Zaten klasik JMeter kullanma kalıbı budur: Yerel bir JBoss üzerinde test hareketleri kaydedilir, daha sonra, *daha güçlü* bir test makinası üzerinde aynı testler geri çalınır.

Şimdi kayıt edici birimi ekleyelim: `Thread Group` altında `Add | Logic Controller | Recording Controller` ile bir kayıt edici eklemiş oluruz. Kayıt edici için hiçbir ayar yapmamız gerekmiyor. JMeter için tek önemli olan kayıt edicinin nerede olduğudur çünkü kayıt edilecek sayfa istekleri o seviyeye yazılacaktır.

Kayıt etmeye başlamadan önce yapmamız gereken son işlem, Internet tarayıcımızın



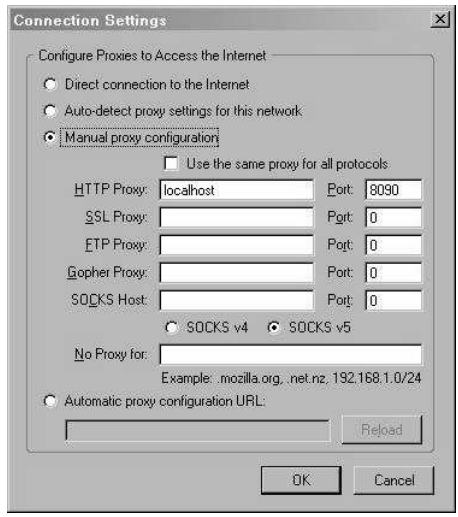
Şekil 5.6: HTTP Request Defaults Ayarları

üzerinden geçiş yapacağı Proxy Server bilgisini vermektir. Bu Proxy Server, JMeter'da tanımladığımız Proxy Server olacak. Mozilla Firefox üzerinde proxy değiştirmek için `Tools | Options | Connection Settings` ekranına gidin, ve `HTTP Proxy` değeri için `localhost` ve `8090` girin. Bu değerler, biraz önce JMeter Proxy Server için tanımladığımız değerlerin aynısıdır. Ve dikkat edin ki `No Proxy For` kutusunda `localhost` olmasın. Internet Explorer için ise, `Tools | Internet Options | Connections | LAN Settings` altındaki `Proxy Server` ekranında "Use a proxy server" kutusunu seçin ve, `Address` ve `Port` kutularına gerekli değerleri girin. Şekil 5.7 üzerinde Mozilla proxy ayarlarını görüyoruz.

Güzel. Artık kaydetmeye hazırız. Şimdi kayıt için şunları yapalım:

- Web uygulamamızı başlatalım
- JMeter'da tanımladığımız `HTTP Proxy Server` ekranındaki `Start` düğmesine basarak, proxy server'ı başlatalım.
- Tarayıcımızı `localhost:8080`'a yönelterek, uygulamımızı test etmeye başlayalım.

Test uygulaması olarak örnek kodlar içindeki `StrutsHibAdv` projesini kullandık. Bu uygulamanın tam adresi `http://localhost:8080/kitapdemo/¬ main.do` adresidir. Bu adrese ilk gittiğimiz zaman, ilk ekran yüklemesi ile arka planda birçok işlemin yapıldığını göreceksiniz. Web isteği, tarayıcı proxy server'ı üzerinden Web uygulamamıza gidecek ve gözlemi yapan proxy server JMeter olduğu için, yaptığımız istek JMeter tarafından yakalanacaktır. Web isteği, JBoss üzerindeki uygulamaya yönlendirilir, ama ondan önce JMeter bu

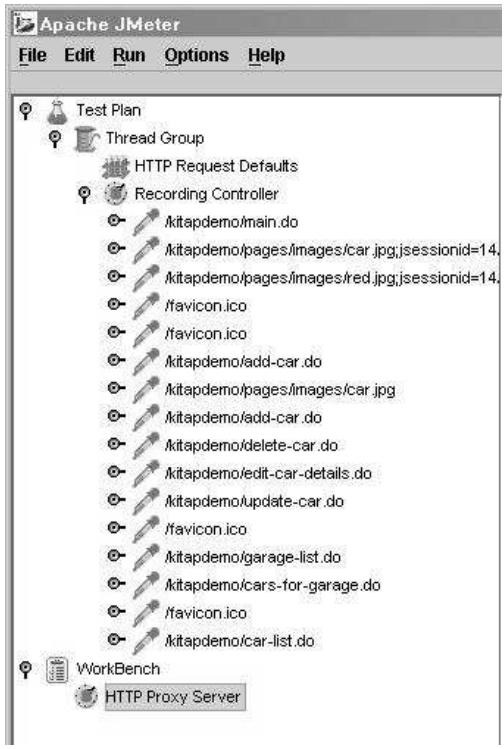


Şekil 5.7: Mozilla Firefox için Proxy Ayarları

isteğin ne olduğunu `Test Plan` altında kaydedecektir (çünkü `Recording Controller`'ı orada tanımladık). Web isteği işini bitirip geri gelir gelmez, `Thread Group | Recording Controller` altında yeni birimlerin *otomatik olarak* eklendiğini göreceksiniz. Şekil 5.8 üzerinde kaydedilen işlemleri görüyoruz. Bu işlemler, şu web isteklerinden üretilmiştir:

- Ana ekrana git
- Yeni bir araba ekle (araba #1)
- Bir yeni araba daha ekle (araba #2)
- Araba #1'i sil
- Araba #2'yi yeni bilgilerle güncelle
- Garage listesi al
- Garage #2 üzerine tıklayarak altındaki arabalara bak
- Tüm araba listesini "Liste" seçeneğine tıklayarak göster

Bir senaryoyu kaydettikten sonra onu `File | Save Test Plan as` menü seçeneği ile disk'e yazabilirsiniz.



Şekil 5.8: Kaydedilen İşlemler

Kaydedilen Testleri İşletmek

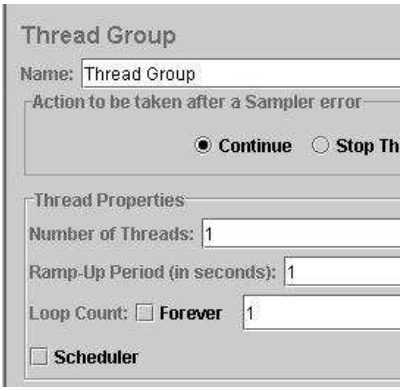
Kayıt edilen web isteklerinin detaylarını, o isteğin üzerine tıklayarak görebilirsiniz. Meselâ `/kitapdemo/add-car.do` isteğine tıklarsak, bu istek içinde hangi parametrelerin gönderilmiş olduğu sağdaki detay ekranında listenecektir. `add-car.do` için `Send Parameters With the Request` başlığının altında, `licensePlate`, `available` ve `size` gibi, bir araba yaratmak için gerekli parametrelerin FORM içinde gönderilmiş olduğunu görüyoruz. Herşey düzgün gözüküyor.

> **Not**: Bir tek detay haricinde her şey düzgün; Kayıt işlemindeki tek eksik, `/kitapdemo/update-car.do` içindeki `version` bilgisinin kayıt edilmemesidir. Bu bilgiyi, o isteğin detayına girip, `Add` düğmesine tıklayarak elle ekleyebilirsiniz. `Version` değeri için `0` değerini girin (version konusunun detayları için 3.8.3 bölümüne bakabilirsiniz).

Kayıt işlemi tamamlandığına göre, senaryomuzu kurumsal uygulamamız üzerinde işletebiliriz. Veri tabanındaki tüm `car` satırlarını temizledikten sonra

(yoksa aynı `licensePlate` kimlikli eklenen ikinci `car` problem çıkartırdı) test senaryosunu JMeter ana menüsü `Run | Start` ile işletelim.

Senaryo, `Thread Group` içinde tanımlanan işletme koşullarına göre işletilecektir. Bu koşullar tek bir thread'in sadece bir kez işletilmesini öngörüyor. Bu ayarlar, `Thread Group` detay ekranında gözükebilir (Şekil 5.9).



Şekil 5.9: Thread Group Detayı

`Thread Group` ekranındaki seçenekleri daha detaylı tarif etmemiz gerekirse:

- **Number of Threads**: Yük testinin eşzamanlı kaç thread ile işletilmesi gerektiğini kontrol eder

- **Ramp-up Period**: `Number of Threads` seçeneğinde belirtilen kadar thread'in, ne kadar süre içinde başlatılması gerektiğini buradan ayarlayabilirsiniz. Meselâ eğer thread sayısı 10 ve `Ramp-Up Period` 100 ise, 100 tane thread, 10'ar saniye aralıklarla başlatılacaktır.

  Eklemek gerekir ki biz yük test senaryolarımızda bu seçeneği pek kullanmıyoruz. Bunun sebebi, eşzamanlı *aktif* sanal kullanıcı sayısının sürekli belli bir sayıda tutacak türden testlere ihtiyaç duymamızdır. Bu bilinen sayı kadar bir yük oluşturulması, uygulamamızın eşzamanlı kaç tane kullanıcıya dayanabileceği hakkında bize bir fikir vermektedir.

- **Loop Count**: `Number of Threads`'de belirtilen kadar thread'in kaç kere arka arkaya işletileceği buradan ayarlıyoruz. Meselâ 10 thread'e `Loop Count` 5 verdiysek, uygulamamız üzerinde toplam 50 thread işlemiş olacaktır (tabii herhangi bir anda, uygulama üzerinde eşzamanlı sadece 10 thread olacaktır).

Değişken (Dinamik) Test Değerleri

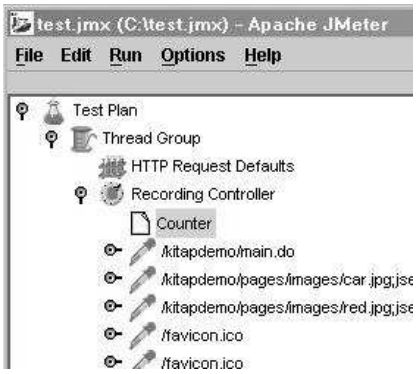
Test planımızı bir kere işlettiğimize göre, aynı planı, eşzamanlı birçok kullanıcı (thread) tarafından ve üstüste işletmeyi düşünebiliriz. Fakat bunun için `Thread Group` altındaki muhtelif alanları değiştirip tekrar işleterek doğru sonuç alacağımızı düşünüyorsak, yanılırız. Eğer senaryoyu bu şekilde işletirsek, Hibernate ve veri tabanı tarafından hatalar geldiğini göreceğiz.

Bu hataların sebebi, birden fazla `Car` nesnesinin aynı kimlik ile birkaç kere eklenmeye çalışılıyor olmasıdır. Bu da, test senaryosunun statik yapısı gözönüne alınırsa, çok normâldir. Test senaryomuz, test içinde gömülü (hard coded) bir `licensePlate` verisini kullanarak yazıldı. Eğer aynı testi birkaç thread'den işletirsek, aynı `licensePlate`'li arabalar, birkaç kez eklenmeye çalışılacaktır, ve bu da `license_plate` kolonu tekil olması gereken `car` tablosu için hatalı bir durumdur!

O zaman, eşzamanlı birçok kullanıcıyı aynı test senaryosu ile kullanmak istiyorsak, senaryomuza dinamik bir `licensePlate` kullandırabilmemiz gerekiyor. Öyle ki, senaryoyu işleten her thread, otomatik olarak yeni bir `licensePlate` kullanıyor olsun.

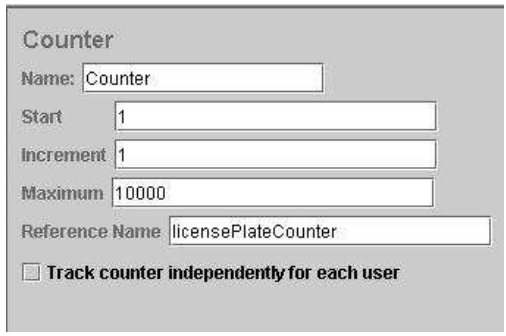
Counter

Senaryomuza bu şekilde bir dinamikliği JMeter programının Counter tekniğini kullanarak ekleyebiliriz. Bir Counter, birim olarak eklendiği `Thread Group` altıda, her değişik thread'in her döngüsünde yeni artacak şekilde tanımlanabilen bir JMeter birimidir. Bu birimi, `Recording Controller` üzerinden `Add | Pre Processors | Counter` seçeneği ile ekleyebiliriz.



Şekil 5.10: Counter

Counter'ın hangi sayıdan başlayıp kaçar kaçar artacağını ayarlamak için, Counter üzerinde tıklayarak detaylarına girebiliriz. Örnek için bizim girdiğimiz değerler, Şekil 5.11 üzerinde gösterilmiştir.



Şekil 5.11: Counter ile Artan Plaka Değişkeni

Bu ayarlara göre Counter, 1 değerinden başlayarak birer birer artacak, ve 10000 üst değerine kadar böyle artacaktır.

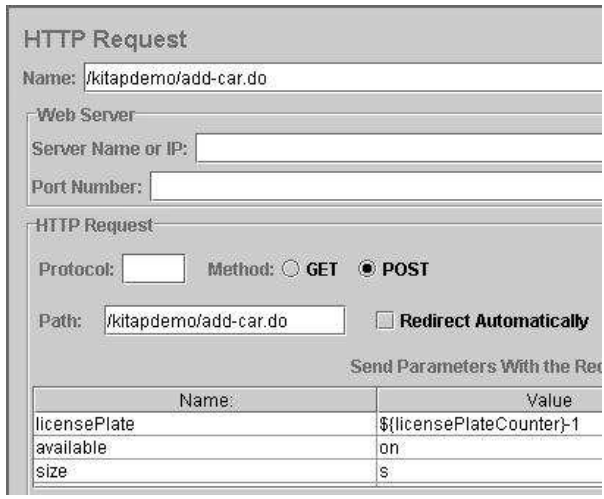
Artık elimizde dinamik, değişken bir `licensePlate` değişkeni olduğuna göre, bu değişkeni gereken web istekleri içinde kullanabiliriz. Meselâ, elimizde iki tane `add-car.do` isteği var, bir istek için `\${licensePlateCounter}-1`, öteki için `\${licensePlateCounter}-2` tanımını kullanırsak, bu istekleri arka arkaya işletebilmiş olacağız (`licensePlateCounter` dinamik bir değişken olmasına rağmen, bir tur bitmeden değişmez, bu sebeple aynı tur içinde iki farklı `Car` için aynı değişkeni kullanmak, yine bir hataya sebep olurdu. O yüzden aynı tur içinde `licensePlateCounter`'ı her araba için ufak bir ek ile değiştiriyoruz).

Dinamik `licensePlate` kullanan diğer yerler şunlardır (ve bunların hepsini `licensePlateCounter` kullanacak şekilde değiştirmemiz gerekiyor).

- Test silme işlemi için biraz önce eklenen `licensePlateCounter` değişkenlerinden birini seçip, silme detayına girip FORM içinde gönderilen parametrelerden `licensePlate` için bu değişkeni (meselâ `\${licensePlateCounter}-1`) girebiliriz. Böylece eklenen arabalardan biri silinmiş olacaktır.

- `edit-car.do`: Yükleme testi içindeki `licensePlate`'i değiştirin (silinmemiş olan `Car` için)

- `update-car.do`: Biraz önce yüklenen `Car`'ın güncellenmesi için, `licensePlate`'i değiştirin. Ayrıca, `description` öğesinin, yine `licensePlateCounter`'ı kullanarak, dinamik bir değer göndermesini sağlayın. Bunun sebebi, eğer bir nesne üzerinde hiçbir değişiklik olmazsa, Hibernate'in bu durumu anlayıp (dirty check), veri tabanında `UPDATE` yapmayacak olmasıdır (Hibernate ne kadar akıllı değil mi?). Biz testimizde bir veri tabanı güncellemesi olmasını istediğimiz için, herhangi bir değerin "değişmesini" zorlamak istiyoruz. Bu yüzden `description` öğesi için "description changed for `\${licensePlateCounter}`" gibi bir ibare kullanabilirsiniz.



Şekil 5.12: Dinamik Plaka Değeri ile add-car.do

Bu değişikliklerden sonra, test senaryomuzu artık birden fazla thread ve birden fazla *kere* işletmemiz mümkün olacaktır. Tam bir yük testi!

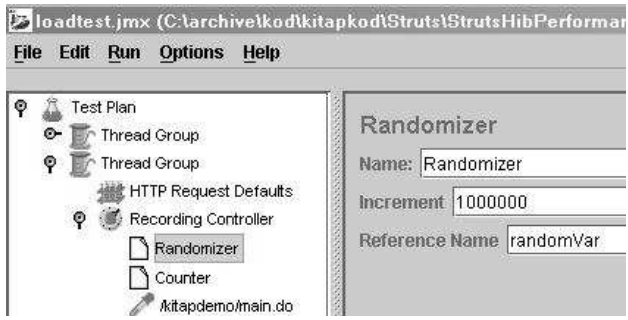
Randomizer

`Counter` dinamik değerleri, her JMeter senaryosunu işlettiğimizde, bizim tanımladığımız başlangıç değerinden başlar. Eğer başlangıç değeri olarak 1 kullandıysak, senaryomuzu her işlettiğimizde `\${licensePlateCounter}` değeri de 1 değerinden başlayacaktır. Fakat yük testlerimize, bizim senaryomuza özel bir şarttan dolayı, her başlangıçta aynı olan bir sayaç değeri yerine, rasgele (random) olan değerler gerekebilir.

Bu tür durumlarda, `Randomizer` adlı `Pre-Processor` birimini kullanabiliriz[2].

`Randomizer` birimine, `Add | Pre-Processors | Randomizer` menüsünden erişilebilir. `Randomizer` eklendikten sonra, aynı `Counter` örneğinde olduğu gibi, tanımlanan bir değişken üzerinden rasgele sayılara erişilebilecektir. Şekil 5.13 üzerindeki örnek bir kullanımda rasgele değerlerin, `\${randomVar}` adlı bir değişkene atanması belirtilmiştir. Bu değişken, artık herhangi bir HTTP istek biriminde rasgele sayı olarak kullanılabilecektir.

[2]Randomizer,tarafımızdan geliştirilmiş bir JMeter birimidir. Bu özelliği JMeter'a eklemek için, kitap örnek programları dizini altındaki randomizer.patch yamasını JMeter sürümü üzerinde uygulamanız gerekmektedir. Yama uygulaması için A.14 bölümüne bakınız.



Şekil 5.13: Randomizer

Veri Hazırlık İşlemleri

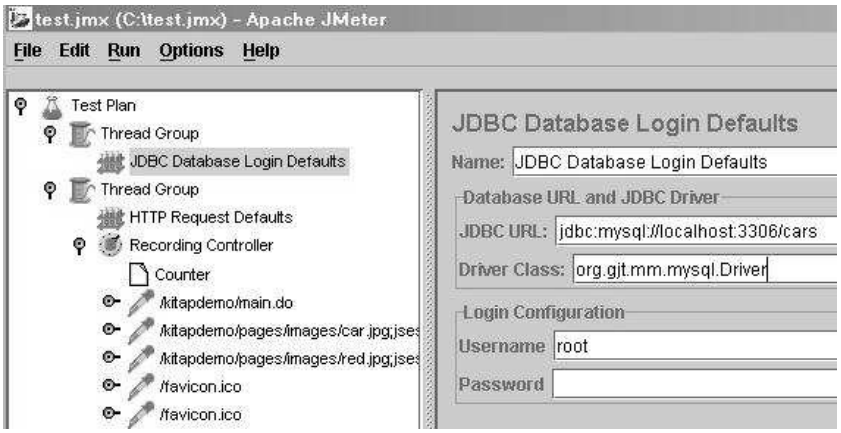
Test senaryomuzu işletmeden önce, her seferinde, veri tabanını elle temizlemek gerekiyor. Bu çok külfetli bir işlemdir (Kural #7). Evet, Counter kullanımı ile her işleyiş *içinde* değerler değişiyor, ya da Randomizer kullanımı ile işleyişten işleyişe bile değişik değerler almamız mümkün, fakat senaryo sonunda her `Car`'ı silen bir web isteğimiz olmadığı için, veri tabanında bazı `car` satırları artık kalmış olacaktır. Eğer her senaryonun temiz bir veri tabanı ile başlamasını istiyorsak, ve eğer silme işlemini elle yapmak istemiyorsak, bu işi JMeter'a yaptırabiliriz. JMeter'ın programlama birimlerinin arasında, bir JDBC sorgu işletme birimi de vardır. Bu birime verilen tanıdık, bildik JDBC bağlantı ayarları üzerinden işletilecek bir sorgu sayesinde her test senaryosunun başında `truncate table car` ile tabanın temizlenmesini sağlayabiliriz.

JDBC birimini yeni bir `Thread Group` altına koymamız gerekiyor, çünkü senaryomuzu işleten esas `Thread Group`'u, eşzamanlı ve birden fazla işleyebilecek şekilde hazırlanmış bir gurup idi. Her thread için veri tabanı temizliği yapılmasını istemiyoruz, temizliğin sadece bir kez, ve test başında yapılmasını istiyoruz. O zaman başa yeni bir `Thread Group` ekleyebiliriz, ve "thread sayısı" ve "kaç kere" parametrelerini 1 değerinde tutarız.

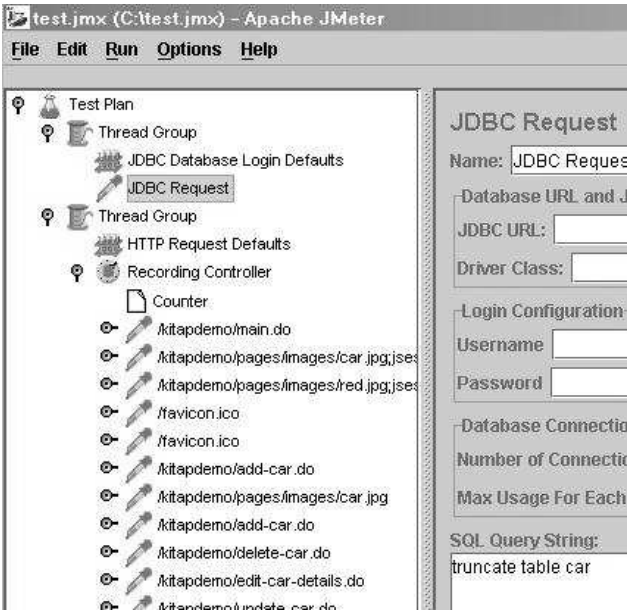
Bu yeni gurup üzerinde `Add | Config Element | JDBC Database Login Defaults` adında yeni bir birim ekleyelim. Veri taban bağlantı bilgilerini buradan gireceğiz. Örnek bir ekranı Şekil 5.14 üzerinde görebilirsiniz.

Bağlantı ayarları tamamlandıktan sonra, işletmek istediğimiz sorguyu girebiliriz. Bunun için ikinci bir JDBC birimi olan `JDBC Request` birimini kullanacağız. Aynı `Thread Group` altında `Add | Sampler | JDBC Request` ile bu birimi ekleyebiliriz. Temizlik sorgumuzu işletmek için de, birimin detaylarına girip `SQL Query String` parametresi için `truncate table car` değerini girebiliriz. JDBC ayarlarının bitmiş hâlini Şekil 5.15 üzerinde görebilirsiniz.

> **Not**: JMeter'in JDBC ile veri tabanınıza erişebilmesi için, gerekli JDBC sürücü jar dosyasına ihtiyacı vardır. Bu dosyayı geliştirme



Şekil 5.14: JDBC Bağlantı Ayarları



Şekil 5.15: JDBC Sorgu Tanımı

dizininden alıp (meselâ MySQL için `mysql-connector-java-3.0.16-ga-bin.jar` dosyası) direk `JMETER_DIR/lib` altına atabilirsiniz.

Tüm bunlar yapıldıktan sonra, artık `Run | Start` ile senaryomuzu istediğimiz kadar thread ile ve istediğimiz kere üst üste çalıştırmamız mümkün olacaktır.

Komut Satırından Çalıştırmak

Eğer JMeter programını görsel arayüzü yerine komut satırından çalıştırmak istiyorsanız, önceden disk'e kaydettiğiniz senaryoyu `File | Open` ile tekrar yüklemek yerine, bir senaryoyu kaydedildiği şekliyle direk komut satırından `jmeter` komutunu `-n` seçeneği üzerinden kullanarak işletebilirsiniz. `JMETER/bin` dizini `PATH` içinde ise şu komutu kullanmak yeterlidir;

```
jmeter -n -t myscenario.jmx
```

Performans Verisini Görmek

Yük testinin işlerken ne kadar zaman aldığını görmek için, görsel bir rapor birimi de eklemek mümkündür. `Thread Group` üzerinden `Add | Listener | Aggregate Report` seçilirse, raporlama birimi eklenmiş olacaktır. Şekil 5.16 üzerinde görülen sonuçlar, yerel bir makina üzerinde 10 eşzamanlı thread'in 5 kere arka arkaya işletilmesi sonucunda elde edilmiştir. Test makinasının yerel olması, loglama seviyesinin DEBUG olması ve `StrutsHibAdv` projesinin üzerinde optimizasyon yapılmamış (şimdilik) bir proje olması sonucunda, 1 saniyede 2.6 işlem gerçekleştirilebilmiştir.

`StrutsHibAdv` projesini nasıl hızlandıracağımızı 5.4 bölümünde göreceğiz.

5.3.2 Unix Üzerinde İstatistik Toplamak

Uygulamamızın, üzerinde çalıştığı işletim sistemini ne kadar verimli kullandığını anlamamız için, işletim sistemi komut satırında bazı programlar kullanarak sistemden istatistikler toplayabiliriz. Bu istatistikler, mikroişlemci, diskler, işletim sistemi önbelleği ve virtual memory (sanal bellek) birimleri hakkında bilgiler verecektir. Bu bilgileri kullanarak işletim sisteminin ne kadar iyi kullanıldığını (utilized) anlayabiliriz.

Bu bölümün geri kalanında, Unix işletim sisteminden performans bilgisi toplamanın yöntemlerini tanıyacağız. Linux işletim sistemi, süreçlerinizin (process) performansını takip edebilmeniz için çok yararlı araçlar sağlar.

Komutlar

Altta gösterilen komutlar, izledikleri süreçlere ve işletim sistemin bütününe başlangıçta ufak bir ek yük getirirler, o yüzden canlı bir sistem üzerinde, özellikle tekrar eden bir şekilde kullanılmaları uygun olmayabilir. Tavsiyemiz, bu araçları yük testi yaptığınız makinada kullamanızdır.

Aggregate Report

Name: Aggregate Report

Write All Data to a File

Filename Browse... Log Errors Only

URL	Count	Average	Min	Max	Error%	Rate
/kitapdemo/main.do	50	489	40	1812	0.00%	10.6/min
/kitapdemo/pages/images/car.jp...	50	103	0	1943	0.00%	10.6/min
/kitapdemo/pages/images/red.jp...	50	78	0	911	0.00%	10.6/min
/favicon.ico	200	60	0	1382	100.00%	39.2/min
/kitapdemo/add-car.do	100	3965	120	104038	0.00%	19.6/min
/kitapdemo/pages/images/car.jpg	50	27394	20519	133843	0.00%	9.9/min
/kitapdemo/delete-car.do	50	4710	100	102988	0.00%	10.5/min
/kitapdemo/edit-car-details.do	50	373	40	2272	0.00%	10.5/min
/kitapdemo/update-car.do	50	700	120	3405	0.00%	10.5/min
/kitapdemo/garage-list.do	50	513	10	7651	0.00%	10.6/min
/kitapdemo/cars-for-garage.do	50	4846	30	127624	0.00%	10.6/min
/kitapdemo/car-list.do	50	445	20	2184	0.00%	10.6/min
TOTAL	800	2989	0	133843	25.00%	2.6/sec

Şekil 5.16: Yük Testi Sonuçlarını Görmek

Süreçler (Processes)

Her sürecin CPU kullanım yüzdesine (utilization) göre ps çıktısı almak (en cok kullanıma sahip olan en altta olmak üzere) için kullanılan `ps` komutu şöyledir:

```
ps -eo pid,pcpu,args | sort +1n
```

Bu komut bizi ayrı ayrı `ps` ve `top` kullanmaktan kurtaracaktır. Bildiğimiz gibi `top`, her süreç no'sunun (PID) ne kadar CPU zamanı yediğini gösterir. Fakat PID, bize bir sürec ismi vermez, bu sebeple testçiler ayrı bir `ps` ve `grep` komutu kullanarak PID'den script ismi almak zorunda kalıyorlardı. Gösterdiğimiz `ps` sayesinde iki komut yerine bir tane yeterli olmaktadır.

vmstat

İşletim sistemin CPU kullanımı, sanal belleği, disk istatistikleri ve boş gerçek hafıza gibi birçok değeri `vmstat` programından öğrenebilirsiniz. Eğer `vmstat <saniye>` gibi bir komut kullanırsanız, `vmstat` bir sonsuz döngü içinde verdiğiniz her saniye değeri kadar bekleyecek, ve sonra uyanarak tüm istatistikleri tekrar toplayacaktır. Altta örnek bir `vmstat` çıktısı görüyoruz.

```
procs --------memory-------- ---swap-- -----io---- --system-- ----cpu----
 r  b  swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa
 2  0     0 760628 44344 155940    0    0    14   122  291 3063 49 2 49 0
```

CPU altında listelenen, `us`, `sy`, `id` kalemleri, sırasıyla kullanıcı, sistem ve boş zaman (idle time) için harcanan zaman yüzdesini gösterir. Bir Java

uygulamasının kullandığı CPU yüzdesi, `us` altında çıkacaktır. İşletim sisteminin çekirdek (kernel) içinde harcadığı zaman `sy` altında çıkar, ve CPU'nun yüzde kaç zamanda boş durduğu ise `id` altında gösterilir.

IO başlığı altındaki `bi` ve `bo`, sırasıyla, kaç bloğun tüm blok araçlarına gönderildiği, ve kaç bloğun araçlardan okunduğunu gösterir. Bu değerler, bir saniyede kaç bloğun araçlara yazılıp okunduğunu belirtirler. Linux üzerinde bir blok, 1 kilobayt'tır (1024 bayt). Diğer Unix'ler üzerinde bir bloğun ne kadar olduğuna `man vmstat` üzerinden bakabilirsiniz.

Memory altında `free` değeri, işletim sisteminin kullanmadığı ve boş duran bellek kapasitesini gösterir. Bu boş alan, herhangi bir uygulama için kullanıma hazır bellek ölçüsüdür. `swpd`, o an kullanımda olan sanal bellek (virtual memory) büyüklüğünü gösterir.

/usr/proc/bin/pstack

Bu komut size Java sürecinizin içindeki lwp yığıtını gösterir. Her Java Thread'i koşmaya başlamadan önce bir lwp'ye (hafif süreç/lightweight process) bağlanmak zorundadır, ve bu komutu ile hangi Thread'in hangi sistem çağrısı üzerinde beklediğini raporda görebilirsiniz. Bütün Thread'leri bir soketten okumada, ya da bir yerel çağırım üzerinden JDBC sürücünüzü beklediğini görebilirsiniz. Bâzen bu gibi bir bilgi yeterli olmayabilir, çünkü hangi Thread'in hangi soket no'sunu beklediğini bu araç göstermez.

/usr/proc/bin/pldd

Bu komut hangi yerel kodun (native code) JVM'inize yüklenmiş olduğunu göstermesi açısından yararlıdır. Komut, hangi `.so` dosyasının JVM sürecine dinamik link edilmiş olduğunu göstererek çalışır.

/usr/proc/bin/pflags

Bu komut, her süreç içindeki her lwp'nin işaretlerini (flags) ve bekleme durumunu (wait state) rapor eder. Bu komutu kaç tane thread context değtirimi (switch) olduğunu anlamak için kullanabilirsiniz. Eğer her JVM için birden fazla CPU var ise, context değiştirimi büyük bir ihtimalle yüksek sayıda olacaktır.

/usr/proc/bin/pfiles

Pfiles, herhangi bir sürecin açık tuttuğu dosyaları gösterir, böylece dosyaların kapanmaması ile alakalı olan problemleri takip etmenize yarar.

lsof

Bir Unix sürecinin açık tuttuğu dosyaları görmek için `lsof`'u[3] kullanabilirsiniz. `lsof`, `pfiles`'a benzer, fakat daha yararlı bir çıktı verir.

truss -c -p

Bu komut, bir sürece karşı başlatılması ve kesilmesi (interrupt) arasında süreç tarafından yapılmış olan sistem çâğrılarını rapor eder. Örnek:

```
\$ ps

 PID TTY     TIME CMD
 11218 pts/5   0:00 ps
 11211 pts/5   0:00 bash

\$ truss -c -p 11211

^C  (Control-C ile programı durdurduk)

syscall             seconds  calls errors
                   -------- ------  ----
sys totals:            .000      0     0
usr time:              .000
elapsed:              3.750
```

5.3.3 Detaylı Performans İstatistiği Toplamak (Profiling)

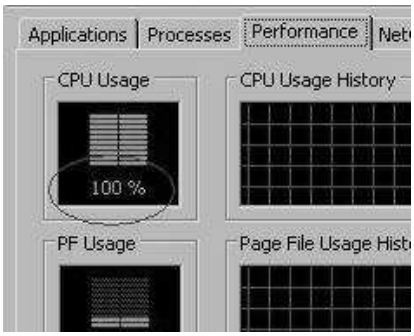
Diyelim ki elinizdeki program çok yavaş çalışıyor. Unix'de `vmstat` ile (ya da Windows üzerinden Task Manager üzerinden) bakıyoruz, ve görüyoruz ki makinadaki CPU kullanımı (CPU utilization) 100%'e gelmiş! Fakat (gene varsayalım) programımız belirli aralıklarla `Thread.currentThread.sleep()` metodunu işleten türden bir program, ve CPU'yu bu kadar zorlamaması gerekiyor.

Acaba programın hangi bölümü bu kadar CPU zamanı çalıyor? Ya da, daha genel bir soru: Hangi Java metotu işlemekte en uzun zaman alıyor?

Genel İhtiyaç

Yukarıdaki sorunun cevabı, Profiler adlı araçlar ile verilir. Elimizdeki problem için bir Profiler'dan beklediğimiz, en acısız şekilde JVM'e eklemlenebilmesi, "kaydet" şeklinde bir komut ile bütün metot çağrımlarını JVM içinde gözetlemeye başlaması, "dur" diyince ham çıktıyı bir dosyaya yazması, ve bu ham veriden, daha sonra görsel bir şekilde bütün metot çağırımlarını ne kadar zaman aldığı bilgisi ile birlikte gösterebilmesidir. Analiz evresinde bir görsel araç

[3] ftp://vic.cc.purdue.edu/pub/tools/unix/lsof/



Şekil 5.17: CPU Kullanımı

ile kullanıcı bütün metotları işleyişte harcadığı zamana göre sıralayarak en yavaş metotu hemen görebilir. Bu metot da dertlerimizin kaynaklandığı metot olacaktır.

Java için revaçta birçok Profiler seçeneği mevcuttur. Şu anda piyasadaki biri bedava biri ticari iki kullanışlı Profiler'dan bahsedeceğiz. Bedava seçenek olarak JDK'nin parçası olan HProf, ticari olarak ta YourKit Java Profiler [4] adlı Profiler üzerinde karar kıldık. Tabii bu iki paket üzerinden anlatacağımız kavramlar **her Profiler** üzerinde geçerli olacaktır. YJP ürünü ticari olsa da, test etme lisansı çerçevesinde 15 gün bedava kullanılabilir.

HProf

HProf bağlamında CPU, bellek kullanım, vs türden istatistikleri toplamak için, incelenen program işletilirken Java komut satırına `-Xrunhprof` eklemek gerekiyor.

Degisik secenekler eklenerek toplanan istatistik tanımlanabilir. Tüm liste için aşağıdaki seçeneği kullanınız.

```
java -Xrunhprof:help
```

Meselâ, CPU bazında istatistik toplamak için

```
java .... -Xrunhprof:cpu=samples,depth=6 com.sirket.XYZ
```

Yukarıdaki komutu işlettikten sonra, ölçmek istediğimiz kod bölümlerini egzersiz eden testlerimizi işletiriz.

Test bittikten sonra, artık sonuçları görmek için, programı bitirmek (JVM'i durdurmak) gerekiyor (Control-C, `kill -9`, vs). Ancak o zaman Profiler topladığı sonuçları bir dosyaya yazacaktır. Dosya ismi tanımlanmaz ise, çıktı dosya ismi `java.hprof.txt` olacaktır. Bu dosya içinde (en altta), en cok zamanın harcandığı metotlar, en fazlası üstte olmak üzere listelenmektedir.

[4] http://www.yourkit.com

```
CPU SAMPLES BEGIN (total = 203235) Fri Jan 21 18:39:44 2005
rank   self  accum   count trace method
   1 14.96% 14.96% 30394   553 java.net.PlainSocketImpl.socketAccept
   2 12.96% 14.10% 1002    123 com.sirket.vs..vs..
   3 11.96% 14.00% 435     343 com.sirket.vs..vs..
   4 10.00% 14.00% 554     564 com.sirket.vs..vs..
```

Buna göre en çok zaman `java.net.PlainSocketImpl.socketAccept` içinde harcanmış. socketAccept'in stack trace'ini görmek için, listedeki TRACE kolonunda verilen numarayı alıp, `java.hprof.txt` içinde baştaki bölümde aramak gerekiyor. Burada aranacak kelime "TRACE 553". Bu da şöyle gözükebilir:

```
TRACE 553:

java.net.PlainSocketImpl.socketAccept(PlainSocketImpl.java:Native method
)
        java.net.PlainSocketImpl.accept(PlainSocketImpl.java:353)
        java.net.ServerSocket.implAccept(ServerSocket.java:448)
        java.net.ServerSocket.accept(ServerSocket.java:419)
        sun.rmi.transport.tcp.TCPTransport.run(TCPTransport.java:334)
        java.lang.Thread.run(Thread.java:534)
        com.sirket.paket.FilancaClass.metot(FilancaClass.java:222)
```

En çok zaman harcanan çağırım zinciri bu olarak gözüküyor. Artık kodumuza dalıp sorunlu olan bölümü optimize etmeye başlayabiliriz.

Call Stack listesinin derinliğini (kaç metot gösterileceğini) `depth=xx` kontrol ediyor. `Cpu=samples`, örnekleme yöntemi ile (her metotun gözlenmediği) türden ölçüm yapmak için kullanılıyor.

YJP

YJP'nin JVM'e eklemlenebilmesi için, komut satırından işletilen `java` komutunun, *YJP üzerinden* işletilmesi lazım. YJP'yi indirip bir dizine kurduktan sonra, programımızı işlettiğimiz `.sh` ya da `.bat` dosyası içine şu komutları eklememiz lazım.

```
set YJP_JAVA_HOME=c:\j2sdk1.4.2\_04
set JAVA_HOME=c:\j2sdk1.4.2\_04

set PATH=C:\yjp-2.5-build310\bin;%PATH%

java -Xrunyjpagent -cp . Test
```

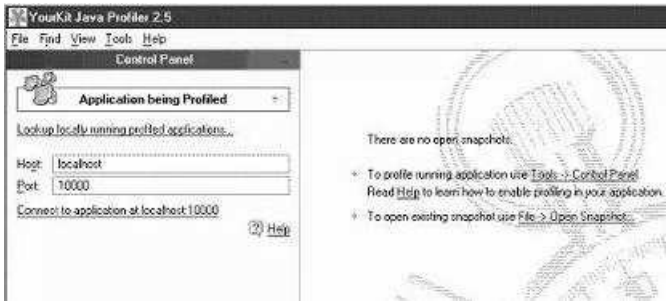
Java komutundan sonra eklenen -Xrunyjpagent seçeneği, kontrolü YJP'ye vermektedir. Şimdi, YJP görsel programını işletmek için (kaydet, dur, göster komutlarını vereceğimiz program bu olacaktır) `YJP_DIZINI/bin/yjp.bat` içine

```
set JAVA_HOME=C:\j2sdk1.4.2\_04
set YJP_JAVA\_HOME=C:\j2sdk1.4.2_04
```

komutlarını ekleyin. Ölçtüğümüz programa dönelim: YJP'nin gözetimi altında işletmek için, kostur.bat'ı koşturalım.

```
> ./kostur.bat
...
c:\\temp>java -Xrunyjpagent -cp . Test
[YourKit Java Profiler 2.5] Listening on port 10000...
```

Bu en son "listening" mesajını görünce, demektir ki YJP gözetimi altında programımız işliyor. YJP hem gözetliyor, hem de port 10000 üzerinden bağlanabilecek bir programa bilgi vermeye hazırlanıyor. Meselâ YJP'nin önyüz programı bu port'a bağlanacaktır. Şimdi o öteki kısmı `YJP_DIZINI/bin/yjp.sh` (ya da `yjp.bat` ile) çalıştıralım.

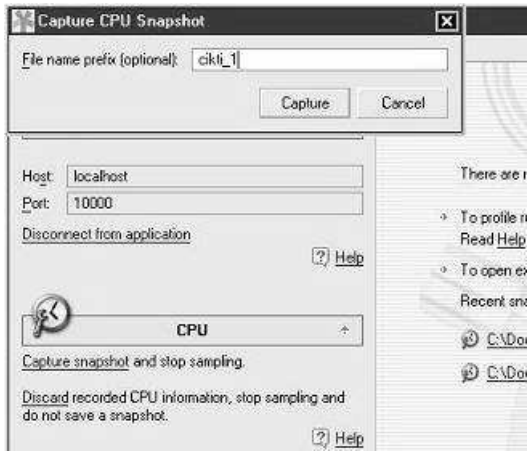


Şekil 5.18: YJP Bağlantı Ekranı

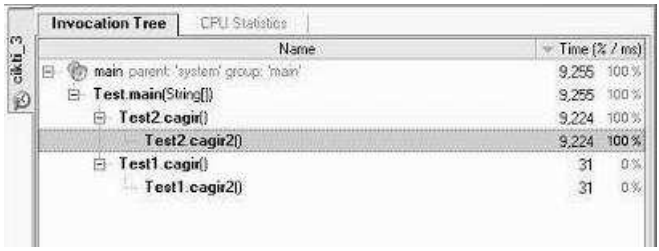
Makina ve port bilgilerini bu programa girdikten sonra, gönderelim. Bağlandıktan sonra, "start sampling" adlı bir seçenek göreceksiniz (sol tarafta). Bunu seçtikten sonra, YJP numune veri toplamaya başlamıştır. Bu veri toplamayı istediğiniz zaman "stop sampling" komutu ile durdurabilirsiniz. Bu durdurmadan sonra, aşağıdaki gibi bir soru sorulacaktır: Ham veriyi hangi dosya üzerine yazmak istiyorsunuz?

Yukarıda `cikti_1` gibi bir isim verdikten sonra, akış sizi direk olarak raporlama ve analiz ekranına alacaktır. Artık bu ekranda metotların işleyiş zamanını görebilirsiniz. `ProfileSample` örnek kodumuzda, en çok zamanın `Test.cagir2()` metotunda harcandığını görüyoruz 9,224 rakamı kodun geri kalan tarafına göre çok büyük bir rakamdır, ve böylece hain kodu bulmuş olduk.

`ProfilerSample` projesinin basitliği bizi aldatmamalı. Bu kadar ufak bir kod içinde, problemli kısmı çıplak gözle de görebilirdik. Fakat düşünün ki 1000 metot çağrımının yapıldığı bir programımız var ve bu programda en yavaş işleyen noktayı bulmammız bekleniyor. Doğal olarak nereden bakmaya başlayacağımızı bile bilemezdik. Profiler araçları hem bu başlangıcı, hem de daha detaylı analizler için gereken özellikleri aynı pakette bize sunmaktadır.



Şekil 5.19: YJP ile Sonuçları Kaydetmek



Şekil 5.20: YJP ile Sonuçlara Bakmak

StopWatch

Detaylı performans ölçümü için ne kadar görsel ticari, açık yazılım ürününü sorunun üstüne atsak ta, bazen kendi yazdığımız ve kodun istediğimiz bölümüne koyabileceğimiz bir araç daha faydalı olabilir. Kodun bir kısmını ölçülebilecek hâle getirmeye, *instrumentation* adı verilir; O zaman denebilir ki, hazır araçlar üzerinden yaptığımız performans verisi toplama işlemi, *tüm kodu* instrument etmektedir. Bu bölümde, sadece istediğimiz kısımları, kendi yazdığımız bir instrumentation aracı ile yapmayı öğreneceğiz.

Bu amaç için `StopWatch` adlı bir yardımcı class'ı geliştirdik. StopWatch kelimesinin İngilizce anlamı, spor müsabakalarında kullanılan türden bir "ölçüm saatidir". Bu saatin bir "başla" bir de "dur" düğmesi vardır, ve başla düğmesine basılınca ölçüm (genellikle milisaniye bazında) alınmaya başlanır. "Dur" komut verilince saat durur, ve o anki ölçülmüş değeri gösterir.

Kodlarımızdan istatistik toplamak için bize gereken saat de, bu şekilde bir

saattir. Tabii bizim `StopWatch` class'ımız, normal saatlere göre bazı ek özellikler de içerecektir:

- Tek bir `StopWatch` ana class'ı üzerinde, farklı kod blokları üzerinde ölçüm alan farklı saatler aktif olabilmelidir.
- Ölçülen bir bloğa *birkaç kere* girilince, yeni alınan değerler eskisine eklenerek, toplamsal (cumulative) ölçüm tutulmalıdır.
- Belli aralıklarla, bir thread üzerinden, tüm saatlerin ölçüm değerleri ekrana basılmalıdır.

Bu özelliklere sahip olan bir `StopWatch` class'ı, şu şekilde kodlanmıştır.

```
public class StopWatch {

    private static Logger logger = Logger.getLogger("appLogger");

    private Calendar startCal;
    private Calendar endCal;
    private TimeZone tz = TimeZone.getTimeZone("CST");

    private static Thread stopWatchThread = null;

    public static final String BLOK_1 = "BLOK_1";
    public static final String BLOK_2 = "BLOK_2";
    ...

    public StopWatch() { }

    public class ShowThread extends Thread {
        public ShowThread() { }
        public void run() {
            while (true) {
                try {
                    Thread.sleep(10000L);
                } catch (Exception e) {
                }
                StopWatch.showAll();
            }
        }
    }

    public StopWatch(String tzoneStr) {
        tz = TimeZone.getTimeZone(tzoneStr);
    }

    public void start() {
```

```
    startCal = Calendar.getInstance(tz);
}

public void stop() {
    endCal = Calendar.getInstance(tz);
}

public double elapsedSeconds() {
    return (endCal.getTimeInMillis() -
            startCal.getTimeInMillis())/1000.0;
}

public long elapsedMillis() {
    return endCal.getTimeInMillis() -
           startCal.getTimeInMillis();
}

public double elapsedMinutes() {
    return (endCal.getTimeInMillis() -
            startCal.getTimeInMillis())/(1000.0 * 60.0);
}

// ---------------- STATIC SECTION ------------------------

public void startThread() {
    stopWatchThread = new ShowThread();
    stopWatchThread.start();
}

private static HashMap watches = new HashMap();

private static HashMap values = new HashMap();

private static HashMap counts = new HashMap();

public static synchronized void start(String key) {
    StopWatch stopWatch = new StopWatch();
    if (!values.containsKey(key)) {
        values.put(key, new Double(0));
        counts.put(key, new Integer(0));
    }

    watches.put(key, stopWatch);

    stopWatch.start();
}
```

```
    public static synchronized void stop(String key) {
        StopWatch stopWatch = (StopWatch)watches.get(key);
        stopWatch.stop();

        double old = ((Double)values.get(key)).doubleValue();
        int oldCount = ((Integer)counts.get(key)).intValue();

        values.put(key, new Double(old + stopWatch.elapsedSeconds()));
        counts.put(key, new Integer(oldCount + 1));
    }
    public static synchronized void showAll() {
        logger.debug(values);
        logger.debug(counts);
    }
}
```

`StopWatch`'ı kullanmak için, ölçmek istediğiniz her kod parçası için bir isim vermemiz gerekiyor. Bu ismi, bir sabit olarak `StopWatch` içinde tanımlayabiliriz. Meselâ,

```
public class StopWatch {
    ...
    public static final String GET_CARS_ACTION = "GET_CARS_ACTION";
    public static final String VIEW_CAR_ACTION = "VIEW_CAR_ACTION";
    public static final String GET_GARAGES_ACTION = "GET_GARAGES_ACTION";
    ...
}
```

Bu tanımlar ile, üç tane ölçüm bloğu tanımlamış olduk. Bloklarda ne kadar zaman geçirildiğini ölçmek için, gerekli `.java` dosyalarına girerek, ölçümün başlaması ve bitmesi gereken yerlerde `start` ve `stop` çağrılarını koymamız gerekiyor; Meselâ `ShowCarDetailAction` içine bir ölçüm bloğu yerleştirelim. Ölçümü başlatıp durdurabilmek için `start` ve `stop` çağrılarını şöyle kullanacağız:

```
public class ShowCarDetailAction extends Action
{
    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response)
        throws Exception {

        StopWatch.start(StopWatch.VIEW_CAR_ACTION);
        ...
        //
        // işlem kodları
        //
        ...
        StopWatch.stop(StopWatch.VIEW_CAR_ACTION);
```

```
        return mapping.findForward("success");
    }
}
```

Ölçümleri, Action'ın `execute` metotunun girişinde başlamak ve dönmeden hemen önce bitecek şekilde koyduk. Daha dar bir alanı da ölçmeyi seçebilirdik.

Toplanan ölçümleri loglarda ve ya `STDOUT` ekranında göstermek için, uygulama bazında tek ayar yapmamız gerekiyor: Uygulama başladığında `StopWatch` içindeki "ekrana istatistikleri basan" thread'i başlatmamız lâzım. Bunun için, `ShowThread` ismindeki `StopWatch` içinde tanımlı bir iç class'ı (inner class) kullanacağız. Bu thread, her 10 saniyede bir uyanıp, o ana kadar toplanan istatistikleri ekrana basmak üzere kodlanmıştır. Ekrana basılan istatikler, her ölçüm bloğu için, o bloğun toplam olarak kaç kere çağırıldığı, ve o blokta toplam olarak (saniye bazında) ne kadar zaman harcandığıdır.

`ShowThread`'i tetiklemek için, meselâ `StrutsHibAdv` projesinde, `AppStartup¬ .start` metotunda şu çağrıyı yapabiliriz.

```
new StopWatch().startThread();
```

Artık uygulamamız başladığında ekrana (hem `STDOUT` hem de log dosyasına) aşağıdaki gibi mesajlar her 10 saniyede bir basılacaktır. Bazı ölçüm değerleri görmek için uygulama üzerinde çok basit bir test yaptık (araba yüklemek, detay görmek, garajlardan araba almak, vs gibi).

```
17:02:36,920 INFO [STDOUT] {GET_CARS_ACTION=5.258, GET_CARS_FOR_GARAGE_AC
TION=0.09, GET_GARAGES_ACTION=0.03, VIEW_CAR_ACTION=0.02}
17:02:36,920 INFO [STDOUT] {GET_CARS_ACTION=2, GET_CARS_FOR_GARAGE_ACTION
=1, GET_GARAGES_ACTION=1, VIEW_CAR_ACTION=1}
```

Üstteki mesajlara göre, `VIEW_CAR_ACTION` bloğu bir kere çağırılmış ve 0.02 saniye zaman harcanmış, `GET_CARS_ACTION` 2 kere çağırılıp içinde 5.28 saniye harcanmış ve `GET_CARS_FOR_GARAGE_ACTION` 1 kere çağırılıp 0.09 saniye harcanmıştır.

Bu sonuçlar üzerinde hemen bir analiz yapacak olursak, `GET_CARS_ACTION` 2 kez çağrılmış olsa bile, bu blokta 5.28 gibi çok uzun bir zaman harcandığını farkederiz. Bunun sebebi, hiç optimize edilmemiş `StrutsHibAdv` projesinin, taban bağlantı havuzlarını (connection pool) anca *kullanıcı ilk isteği yaptıktan* sonra kurmaya başlıyor olmasıdır (ilk ekran da arabaları yüklediği için `GET¬ _CARS_ACTION` bloğuna girilir). Bu yüzden bu blokta 5.28 gibi çok uzun bir zaman geçirilmiştir. 2.2.4 bölümünde yaptığımız tavsiyenin geçerli olduğunu böylece görmüş oluyoruz: Taban bağlantıları, önbellekleme gibi başta hazır olması daha uygun olan ilk yüklemeleri, uygulama başında bir şekilde zorlayıp, ilk gelen kullanıcıya bu performans bedelini ödetmememiz gerekmektedir.

5.4 Performans İyileştirmeleri ve Ölçeklemek

Bu bölüme kadar, bir kurumsal uygulama üzerinde yük yaratmak ve kod üzerindeki tıkanma noktalarını (bottleneck) bulmak için kullanılabilecek araçları tanıdık. Eğer elimizde beklenen bir performans kriteri var ise, bu kritere ulaşıp ulaşmadığımızı yük testleri sonuçlarına bakarak anlayabiliyor, ve problem yaratabilecek alanlarda harcanan zamanı profiler araçları daha detaylı bir şekilde alabiliyorduk.

Bu ölçümler sonucunda, yapılabilecek bazı genel optimizasyon numaralarını bu bölümde paylaşmak istiyoruz. Bahsedeceğimiz türden genel optimizasyonlar, JBoss, Hibernate, veri tabanı ve log sistemi odaklı olacaktır. Optimizasyonlar, şu başlıklar altında işlenecek:

1. JVM'e yeterli bellek ayırmak
2. Log seviyesini değiştirmek (üçüncü parti ürünler ve kendi kodlarımız için)
3. JBoss Web işleyicisinin kullandığı thread sayısını arttırmak
4. Hibernate açılış ve kritik bazı veri yükleme işlemlerini uygulama başında yapmak
5. Hibernate POJO nesneleri ve listelerini, veri tabanı yerine önbellekten almak
6. Hibernate POJO'lar arası ilişkilerde, tembel yükleme seçeneğini kullanmak
7. Hibernate'in ilişkileri için ürettiği SQL'i iyileştirmek için, `fetch=‘‘join¬ ’’` seçeneğini kullanmak
8. Web uygulamamızı küme ortamında çalıştırmak
9. Session Bean'lerimizi dağıtık yapıda kullanmak
10. Veri tabanını hızlandırmak için, gereken yerlerde kolon indekslerini eklemek

Genel kural olarak, tıkanma noktalarını tamir süreci şöyle olmalıdır:

> **Performans İyileştirme Metadolojisi**: İlk önce en ciddi tıkanma noktasını bulun ve tamir edin. Sonra bu işlemi tekrar edin. Performans kabul edilir seviyeye gelince, durun.

5.4.1 JVM

Java uygulamalarını işletmek için kullanılan komut satır programı `java` (`JAVA_HOME/bin` altında), en az ve en çok ne kadar hafıza kullanacağını seçenek olarak alabilir. Meselâ en az 100 Megabayt ve en fazla 200 Megabayt isteyen bir uygulama şöyle başlatılabilir:

```
java -Xms100m -Xmx200m -classpath ... MyApplication
```

JBoss ortamında, uygulama servisini biz direk `java` komutu ile değil, başka bir script içinden çalıştırdığımız için, `java` komutunu kullanan bu script'i değiştirmemiz gerekiyor. Bu script `JBOSS_HOME/bin` altında olan `run.conf` dosyasıdır; İçeriği, paketten çıktığı şekliyle şöyledir:

```
if [ "x\$JAVA_OPTS" = "x" ]; then
   JAVA_OPTS="-server -Xms128m -Xmx128m"
fi
```

Değiştirmemiz gereken `-server` seçeneğinden sonra gelen kısımdır. Uygulamamızı istediğimiz ölçekte çalıştırmak için, JVM'e işletim sisteminde mevcut gerçek bellek değerinin izin verdiği kadar bellek yeri vermeliyiz. Meselâ, 1 GB belleği olan bir Linux servis makinasında, aşağı yukarı 200 MB kadar yeri işletim sisteminin kendisi kullanır. Geri kalan yerin tamamı JVM'e verilebilir. En az ve en çok değerleri aynı olmasında hiç bir sakınca yoktur (biz böyle yapıyoruz).

"Ne kadar eşzamanlı kullanıcı/işlem için ne kadar bellek gerektiği" sorusuna cevabı 5.2.1 bölümündeki Kurumsal Web'in Altın Kanunu vermişti: Bu oran başlangıç olarak iyi bir kriterdir. Tabii bu oranın oturum kısmını zorlayarak 600, 700 rakamlarına çıkabilirsiniz, fakat belli bir noktadan sonrası uygulamanızın optimal işleyişi kötü yönde etkilenecektir. Dikkat edilmesi gereken diğer bir nokta, 500 sayısının eşzamanlı aktif olan *oturum* sayısını göstermesidir. Bir sistemde 500 tane oturum olması demek, 500 kişinin *sürekli, aynı anda* bir sayfaya tıklayıp sistem üzerinde bir istek yaratması demek değildir. Kullanıcının düşünme zamanı (think-time) olduğunu gözönünde bulundurursak, 500 session, belki 100 eşzamanlı sayfa isteğine tekabül eder. Bu sayı çok önemlidir, çünkü JMeter ile yük testi yaparken `Thread Group` altında vermeniz gereken eşzamanlı thread sayısı, 500 değil, 100 olacaktır.

5.4.2 Log4J

3.3.3 bölümünde, JBoss ortamında Log4J ayarlarını nasıl ve nerede yapacağımızı tarif etmiştik. Performans arttırımı için, bu ayar dosyasının her tür bilgilendirme mesajına (özellikle `DEBUG` seviyesindeki mesajlara) izin vermemesi gerekir. Çünkü `DEBUG` seviyesindeki bilgilendirme mesajları, kod içinden *en fazla* üretilen mesajlardır. Genelde hata bulmak için programcı tarafından koda eklenen bu mesajlar, nihai ortamda işe yaramayacak bir ton mesajı log dosyasına basarlar. O sebeple, nihai ortamda kapatılmaları gerekir.

Bildiğimiz gibi Log4J ortamında bilgilendirme mesajları `logger.debug(¬ ‘‘mesaj’’)` kullanımıyla basılır. Bir üst seviyede, yâni bilgilendirmeden daha nadir ve daha ciddi olan mesajlar `INFO` üzerinden, `logger.info` kullanarak basılacaktır. Hatalar ise, tipik olarak `logger.error` ile loglanır.

Loglama işlemini hızlandırmak için bizim kullanacağımız özellik ise, Log4J'in loglama seviyesini Java paket bazında kontrol edebilme özelliğidir. Bu özellik sayesinde, her Java paketinin hangi seviyede loglaması gerektiğini tanımlayabiliyoruz. Nihai ortamda çalışan bir servisin, tüm paketler için loglama seviyesini `INFO`'ya çekmesi gerekiyor. Bu şekilde ayarlanmış örnek bir `log4j.xml` dosyası, aşağıda görülebilir (bu dosyanın bir örneğini `StrutsHibPerformance` projesinde de bulabilirsiniz).

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/"
                     debug="false">
   <appender name="appAppender"
             class="org.jboss.logging.appender.DailyRollingFileAppender">
      <param name="File"
             value="\${jboss.server.home.dir}/log/kitapdemo.log"/>
      <param name="Append" value="false"/>
      <param name="DatePattern" value="'.'yyyy-MM-dd"/>
      <layout class="org.apache.log4j.PatternLayout">
         <param name="ConversionPattern"
                value="[%d] %5p [%t] (%F:%L) - %m%n"/>
      </layout>
   </appender>

   <category name="org.apache">
      <priority value="INFO"/>
   </category>

   <category name="org.jgroups">
      <priority value="WARN"/>
   </category>

   <category name="org.hibernate">
      <priority value="INFO"/>
   </category>

   <category name="org.hibernate.cache">
      <priority value="INFO"/>
   </category>

   <category name="org.apache.axis">
      <priority value="INFO"/>
   </category>

   <category name="com.opensymphony.oscache">
```

```
    <priority value="INFO"/>
  </category>

  <category name="com.mchange.v2.resourcepool">
    <priority value="INFO"/>
  </category>

  ....

  <logger name="appLogger" additivity="false">
    <level value="INFO"/>
    <appender-ref ref="appAppender"/>
  </logger>

</log4j:configuration>
```

Görüldüğü gibi `<category name>` ibaresinden sonra, seviyesi set edilmek istenen *paket* ismi veriliyor. Her paketin hangi seviyede loglanacağı `<level value>` etiketi ile bildiriliyor.

Peki üstteki listeye hangi paketleri ekleyeceğimizi nereden bildik? Çok basit: Uygulamayı birkaç kez işletip, `kitapdemo.log` dosyasında yazılan `DEBUG` mesajlarının hangi paketten geldiğine baktık (Log4J bir mesajın hangi paketteki hangi class'tan geldiğini gösterebilir) ve mesajlarını görmek istemediğimiz paketleri, üstteki listeye ekledik. Bir paketin seviyesini değiştirdiğimiz zaman, o paketin altındaki tüm class'ların seviyesi o seviyeye set edilmiş olacaktır.

Nihai ortamda yanlış log seviyesinde (`DEBUG`) çalışıyor olmak, başlangıçta yapılan en yaygın performans hatalarından biridir. Ne zaman 10 saniyede işlemesini beklediğimiz kurumsal uygulamamızın ilk deploy edildikten sonra 3 dakikada çalıştığını görürsek, ilk kontrol etmemiz gereken yerin log seviyeleri olduğunu hemen hatırlamalıyız. Meselâ eğer seviyelerini `INFO`'ya çekmemişseniz, EHCache, OSCache ve Hibernate paketlerinin müthiş miktarda `DEBUG` log üretip uygulamanıza diz çöktüreceğine emin olabilirsiniz. Bu paketlerin log seviyesini `INFO`'ya getirdiğiniz anda, uygulamanız kanatlanıp uçacaktır.

5.4.3 JBoss Thread'lerini Arttırmak

Yük testlerinizi gerçekleştirirken, JMeter `Thread Group` sayımız 150 üzerine çıktığınızda, JBoss'tan "maksimum thread sayısına erişildiği" ve "servisin tıkandığı" hakkında bir mesaj alırsınız. Bunun sebebi, paketten çıktığı hâliyle JBoss'un HTTP isteklerini karşılayan thread sayısının da 150 olmasıdır. Tek JVM üzerinde 150'den daha yüksek ölçekte testler gerçekleştirebilmek için, HTTP istek karşılayıcı thread sayısını arttırmamız gerekiyor. Bu değişikliği yapacağımız dosya, `JBOSS¬_HOME/server/default/deploy/jbossweb-tomcat50.sar` altındaki `server¬.xml` dosyasıdır. Burada gördüğünüz

```
<Connector port="8080" address="\${jboss.bind.address}"
```

```
maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
```

olarak verilen satırdaki, `maxThreads` için tanımlanmış değeri 150'den daha yüksek bir değere çekersek, daha yüksek ölçekte yük testlerimizi servis üzerinde kullanabiliriz. Değişiklikten sonra JBoss servisinizi kapatıp açmamız gerekecektir.

5.4.4 Açılış Hibernate İşlemleri

Uygulamamız başlarken eğer `HibernateSession` nesnesi hiç çağrılmaz ise, Hibernate hiçbir hazırlık işlemi yapmayacaktır. Bu hazırlıklar işlemleri arasında, gereken tüm veri taban bağlantılarının açılıp havuza konması, nesne eşleme dosyalarının okunması ve POJO'ların bir önişlemden geçirilerek kalıcılık işlemleri için hazır edilmesi gibi işlemler vardır.

Hazırlık işlemlerini tetiklemek için, `HibernateSession` içindeki `static` kodların bir şekilde çağırılması gerekmektedir. Bunu yapmak için en basit yol `HibernateSession` nesnesinin bir `static` metotunu çağırmaktır, çünkü `HibernateSession` hafızaya alındığı an Java kurallarına göre içindeki `static` blok bir defaya mahsus olmak üzere işleme konur.

`HibernateSession`'i kullanmak için üzerinde zararsız bir işlem bulmamız lazım: Bu da `openSession` ve hemen arkasından `closeSession` çağrısı olabilir. Bu iki işlem bir Hibernate oturumunu hemen açıp kapatır, ve toplam olarak hiçbir şey değişmemiş olur, fakat `static` blok işleme konduğu için tüm Hibernate hazırlıklarının yapılması zorlanır. Bu iki çağrıyı, `AppStartup` nesnesindeki `start` metotunun içine koyabiliriz. Örnek kodu `StrutsHibPerformance` projesinde bulabilirsiniz.

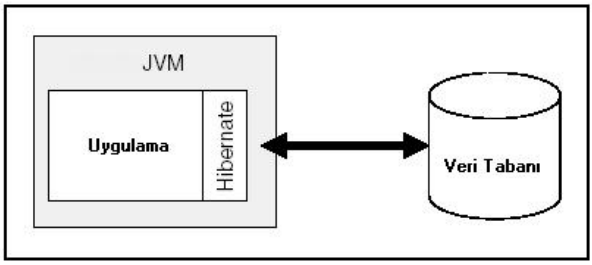
```
public class AppStartup implements AppStartupMBean {
    ..
    public void start() throws Exception {
        HibernateSession.openSession();
        HibernateSession.closeSession();
        ..
    }
}
```

Başlangıçta bağlantı havuzlarının hazırlanmasını zorladığımız gibi, bazı verilerin Hibernate önbelleğine alınmasını da zorlayabiliriz. Bunu yapmak için, uygulama başında uygulamanın sıkça kullanacağını *bildiğimiz* nesneleri `get` ve HQL sorguları ile hafızaya getiririz; Böylece bu nesneler aynı anda önbelleğe alınmış olurlar.

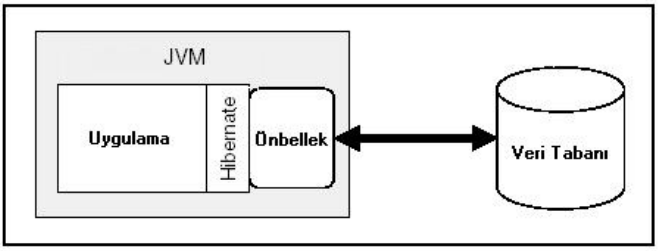
Bu işlemleri de `AppStartup.start` metotunda gerçekleştirebiliriz (nasıl olsa bir Hibernate oturumunu açmış bulunmaktayız, oturumu hemen altındaki satırlarda veri yüklemesi için kullanabiliriz).

5.4.5 Hibernate Önbellek Kullanımı

Hibernate kullanan kodların JDBC kullanan kodlardan daha hızlı olmasının iki sebebi, Hibernate'in ürettiği SQL'in elle yazılan SQL'den daha optimal olması ve Hibernate'in sık değişmeyen POJO'ları veri tabanı yerine önbellekten alabilmesidir. Hibernate, kontrolünde olan POJO'ları, gene kendi kontrolünde olan `get`, `saveOrUpdate` ve `delete` komutları üzerinden ne zaman önbelleğe yazacağını ve ne zaman oradan silmesi gerektiğini çok iyi bildiği için, Hibernate *üzerinden* önbellek kullanımı programcı üzerinde çok az külfet getiren bir işlem hâline gelmektedir.



Şekil 5.21: Önbelleksiz Hibernate



Şekil 5.22: Önbellekli Hibernate

Hibernate ile beraber kullanılabilecek birçok açık yazılım ve ticari önbellek paketi mevcuttur. Biz bu bölümde EhCache ve OsCache önbellek paketlerini işleyeceğiz. EhCache, tek JVM'li ortamlarda, OsCache ise, dağıtık mimarilerde kullanılmasını tavsiye ettiğimiz paketlerdir.

Genel Ayarlar

Önbellek kullanmak için, `hibernate.cfg.xml` dosyasında hangi nesnelerin ve ilişkilerin önbellekleceğini belirtmelisiniz.

```
<class-cache
    class="org.mycompany.kitapdemo.pojo.Car"
    region="Simple"
    usage="read-write"/>

<class-cache
    class="org.mycompany.kitapdemo.pojo.Garage"
    region="Simple"
    usage="read-write"/>

<collection-cache
    collection="org.mycompany.kitapdemo.pojo.Garage.cars"
    region="Simple"
    usage="read-write"/>
```

Üstteki tanımlara göre `Garage`, `Car` nesneleri, ayrıca `Garage` üzerindeki `cars` ilişkisinin önbelleğe alınacağı belirtilmiştir. Hibernate, sorgulardan gelen sonuçları da önbellekleyebilir. Bunun için, Hibernate `Query` nesnesinin üzerinde `setCacheable(true)` çağrısını yapmalısınız, ve `Query` üzerinde sorgu sonuçlarının hangi önbellek bölgesini (region) kullanacağını belirtmelisiniz.

```
Query query = s.createQuery("from Car");
query.setCacheable(true);
query.setCacheRegion("KitapDemoQueries");
```

Buradaki kullanım, sorgu sonuçlarının `KitapDemoQueries` adlı bir bölgede kullanılması gerektiğini belirtiyor. Sorgu sonuçlarının önbellek bölgesi, aynen POJO ve nesne ilişkilerin bölgeleri gibi tanımlanmıştır.

Sorgu sonuçları önbelleklemesinin perde arkası şöyledir: Hibernate, bir sorgudan gelen sonuçların sadece kimlik değerlerini önbellekte tutar. Gerçek objeler, ikinci seviye önbellekte tutulacaktır. Bir sorgu sonucunun önbellekten atılması (eviction) POJO nesnelerinden farklı olarak, o sorgu içinde referans edilen herhangi bir nesnenin değişmesi (`delete`, `saveOrUpdate`, `merge`) durumunda olur. Bu aslında oldukça kısıtlayıcı bir kıstastır (tabii ki sistem doğruluğu için böyle yapılması gerekir) ama sorguları önbelleğe alma/almama kritlerimizde aklımızda tutmamız gereken önemli bir etkendir.

Önbellek Bölgeleri

Önbellek paketinin önbelleklenen bir nesneyi ne kadar uzun süre içeride tutacağı, ne kadar kullanılmadan kalmasına (idle time) izin vereceği, o nesneden kaç tane nesnenin olmasına (bölge büyüklüğü bağlamında) izin vereceği gibi *fiziki ayarlar*, hep bir bölge baz alınarak yapılmaktadır. Eğer fiziki ayarlar, her

nesne, her ilişki için aynı olsaydı, Hibernate birbirinden değişik olması gereken türden önbellek kullanımlarına hitap edemezdi.

EhCache

Hangi nesne ve ilişkileri önbelleğe alınmasını belirttik. Artık kullanacağımız önbellekleme paketini belirtmeli, ve bu önbellek paketine bölge tanımları vermeliyiz. Paket olarak EhCache kullanmak için, `hibernate.cfg.xml` dosyasında, EhCache ile köprü kurabilen bir class'ı belirtmemiz gerekiyor.

```
<property name="hibernate.cache.provider_class">
  org.hibernate.cache.EhCacheProvider
</property>
```

`EhCacheProvider`, Hibernate ile EhCache arasında bağlantı kuran köprü kodudur. Bu tanımdan sonra, `CLASSPATH`'te bulunması gereken ayrı bir dosya `ehcache.xml` içinde, her *bölgenin* fiziki ayarlarını yapmamız gerekiyor.

```
<ehcache>
  <diskStore path="java.io.tmpdir"/>
  <cache name="org.mycompany.kitapdemo.pojo.Car"
         maxElementsInMemory="1000"
         eternal="true"
         timeToIdleSeconds="0"
         timeToLiveSeconds="0"
         overflowToDisk="false"
         />
  <cache name="org.mycompany.kitapdemo.pojo.Garage"
         maxElementsInMemory="1000"
         eternal="true"
         timeToIdleSeconds="0"
         timeToLiveSeconds="0"
         overflowToDisk="false"
         />
  <cache name="org.mycompany.kitapdemo.pojo.Garage.cars"
         maxElementsInMemory="1000"
         eternal="true"
         timeToIdleSeconds="0"
         timeToLiveSeconds="0"
         overflowToDisk="false"
         />
  <cache name="KitapDemoQueries"
         maxElementsInMemory="1000"
         eternal="true"
         timeToIdleSeconds="0"
         timeToLiveSeconds="1000"
         overflowToDisk="false"
         />
</ehcache>
```

Her önbellek bölgesini tanımlarken kullandığımız parametrelerin açıklaması şöyledir:

- **maxElementsInMemory**, bir bölge içinde en fazla kaç tane birimin tutulağını belirtir.

- **eternal**: Bölge içindeki birimin sonsuza kadar önbellekte kalıp kalmayacağını `eternal` parametresi için `true` ya da `false` vererek ayarlayabiliriz. Bu değer `true` olarak belirlenmişse, o bölgedeki birimlerde değişiklik yapılmadığı sürece önbellekte tutulurlar. Ayrıca bu durumda `timeToIdleSeconds` ve `timeToLiveSeconds` değerlerine bakılmayacaktır. Eğer `eternal false` ise, `timeToIdleSeconds` ve `timeToLiveSeconds` parametrelerinin ayarları baz alınır.

- **timeToIdleSeconds**: Bir nesnenin önbellekte ne kadar kullanılmadan kalabileceğini belirler. Eğer bir nesne, önbellekten x saniye kadar alınmamışsa ve `timeToIdleSeconds` değeri x saniye ise, o obje önbellekten atılacaktır.

- **timeToLiveSeconds**: Bir nesnenin, *kullanılsa da kullanılmasa da* ne kadar süre önbellek bölgesinde tutulacağını belirtir.

OsCache

Performansı EhCache kadar iyi olan diğer bir önbellek paketi, OsCache'dir. OsCache kullanmak için, `hibernate.cfg.xml`'de OsCache ile iletişim kuracak bağlantı kodunu tanımlamak gerekiyor.

```
<property name="hibernate.cache.provider_class">
     org.mycompany.kitapdemo.util.OSCacheProvider
</property>
```

Dikkat: OsCache ile bağlantı kurmak için `org.mycompany.kitapdemo.util.¬ OSCacheProvider` class'ını tanımladık[5]. Önbellek bölge ayarlarını ise, `CLASSPATH`'te tutulacak `oscache.properties` adlı bir dosyada yapıyoruz.

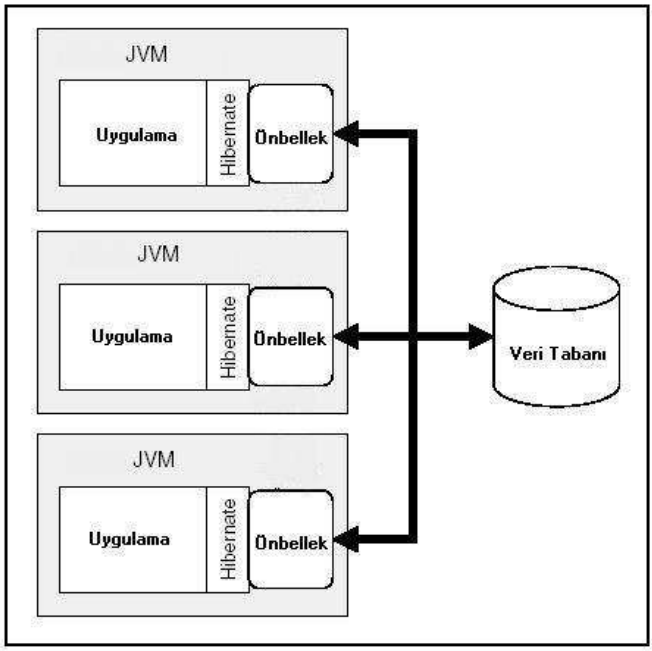
```
cache.capacity=100000
cache.timeout=-1
```

OsCache bölge ayarları EhCache'e kıyasla daha basittir. Tüm nesneler aynı bölge içinde olacaktır. Fakat OsCache'de, EhCache'de olmayan bir özellik vardır, o da dağıtık bir yapıda çalışabilme özelliğidir.

[5]Hibernate paketinin içinden çıkan OsCacheProvider kodu, dağıtık mod'da calışmaya hazır değildir. Köprü kodunu, OsCache programcıları değiştirerek dağıtık şekilde çalışmaya hazır hâle getirdiler. Bu kodu, http://wiki.opensymphony.com/display/CACHE/Hibernate adresinden, ya da StrutsHibPerformance projesinde bulabilirsiniz

Dağıtık Mimari ve OsCache

Dağıtık mimarilerde önbellek kullanımı, eğer kendi hâline bırakılırsa problem doğuracak bir durumdur. Meselâ dağıtık mimariye hazır olmayan EhCache'i dağıtık yapıda kullandığımızı varsayalım. Şekil 5.23 üzerinde bu mimariyi görüyoruz.



Şekil 5.23: Yanlış Dağıtık Mimari ve Önbellek Kullanımı

Böyle bir yapıda, meselâ, hem JVM #1'in hem JVM #2'nin *aynı anda* okuduğu bir Car #1 nesnesi olduğunu farz edelim. Böyle bir yapıda, eğer Car #1 nesnesi JVM #1 tarafından değiştirilirse, bu değişiklik JVM #1'in önbelleğine ve veri tabanına doğru şekilde yansır. Fakat, güncellemeden önce Car #1'i kendi önbelleğine almış olan **JVM #2**, bu durumdan tamamen habersiz kalacaktır. Bu durum, uygulamanın doğruluğu açısından tam bir hezamet olur, çünkü önbelleğinde tuttuğu nesnenin doğru olduğunu zanneden JVM #2, veri tabanına gidip en son değeri alması gerektiğinden habersizdir. Böylece JVM #2'i kullanan bir kullanıcı, eski ve yanlış değerler görecektir.

Bu durumu düzeltmek için, ağ üzerinden *birbirlerine mesaj gönderebilen* türden dağıtık önbellek paketleri kullanmak gerekmektedir. Şekil 5.24 üzerinde böyle bir yapı görüyoruz.

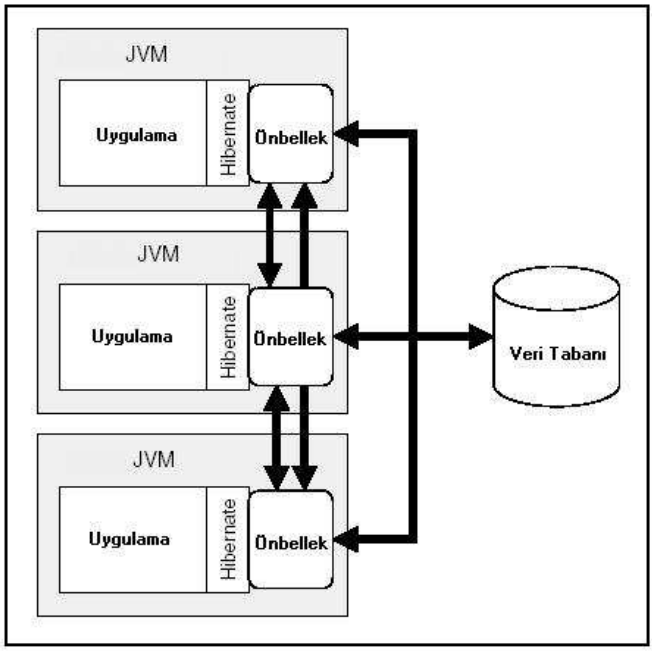
OsCache, dağıtık yapıda çalışabilen türden bir önbellektir. Önbellekler arasında haberleşme, OsCache içinde JGroups adı verilen bir multicast protokol ürünü ile

halledilir. Multicast, bildiğimiz gibi, yayın bazlı bir protokoldür ve yerel ağlarda sıkça kullanılan Ethernet donanımının da yayın bazlı olduğunu düşünürsek, yayın bazlı donanım üzerinde yayın bazlı protokolün optimal bir şekilde çalışabileceği sonucuna varırız.

OsCache'i dağıtık bir yapıda çalıştırmak için tek yapmamız gereken, `oscache.properties` içinde gerekli tanımları yapmaktır.

```
cache.cluster.multicast.ip=231.12.21.132
cache.event.listeners=
   com.opensymphony.oscache.plugins.clustersupport.
   JavaGroupsBroadcastingListener
cache.blocking=false
```

Bu ayarlar ile, OsCache önbelleği içinde değişen her nesne için yerel ağ üzerindeki *her diğer OsCache'e* multicast ile bir mesaj gönderir. Bu mesaj, değişen her nesnenin diğer önbelleklerden silinmesi için gönderilen bir mesajdır, böylece kendi önbelleğinden bir nesneyi kaybeden diğer JVM'ler, bu nesnenin gerçek değerini almak için veri tabanına gitmeye mecbur kalırlar. Bu da istediğimizi bir davranıştır, çünkü o nesnenin en son *doğru* hâli, veri tabanında olacaktır.



Şekil 5.24: Doğru Dağıtık Mimari ve Önbellek Kullanımı

5.4.6 Hibernate ve Tembel Yükleme (Lazy Loading)

Tembel yükleme, içinde veri taşıyan bir nesnenin (tek nesne ya da nesneler içeren bir `List`) verilerini o veriye ihtiyaç oluncaya kadar yüklemeyip beklemesi anlamına gelir. Hibernate, 3. versiyondan itibaren, olağan davranış olarak tembel yükleme yapmak üzere ayarlanmıştır. Hibernate `get()` ya da `load()` ile bir ana nesneyi yüklerseniz, bu nesnenin öğeleri veri tabanından yüklenecek, fakat bu nesnenin bire bir, bire çok, ya da çoka çok ilişkisinin olduğu diğer nesnelerin içerikleri yüklenmeyecektir.

Bunun yapılmasının sebebi, uygulamaların kullanım düzenine (pattern) bakılarak, genelde bir nesneden, o nesnenin ilişkinin olduğu *her diğer* nesneye atlanmadığının bilinmesidir. Eğer kullanıcı, çoğunlukla, A nesnesinin detaylarına bakıyor fakat A'nın ilişkide olduğu B'nin detaylarına (öğelerine) hiç bakmadan A nesnesini atıyorsa, o zaman "A yüklenince B'nin otomatik yüklenmesi" gibi bir önsezinin hiçbir değeri olmayacaktır. Hibernate, bu genel kullanım düzenine göre bir optimal ayarı desteklemiştir.

Fakat, her tür kullanım şekli ve uygulama çeşidi için bu doğru olmayabilir. A yüklenince B'nin detaylarına ihtiyaç olan şartlar için, tembel yüklemeyi kapatabilirsiniz. Hibernate, tembel yükleme ayarını class ve ilişki seviyesinde yapmanıza izin verir. Meselâ her `Garage` yüklenince onunla ilişkide olan `Car` listesi yüklensin istiyorsanız, `Garage.hbm.xml` eşleme dosyasında şu ayarı yababilirsiniz.

```
<set name="cars"
     inverse="true"
     lazy="false"
     >
  <key>
    <column name="garage_id"/>
  </key>
  <one-to-many class="Car"/>
</set>
```

Eğer bir class'ın ilişkide olduğu tüm ilişkiler için tembel yüklemenin iptal edilmesini istiyorsanız, class seviyesinde şu ayarı yapmanız gerekir:

```
<hibernate-mapping package="org.mycompany.kitapdemo.pojo">
  <class name="Garage" table="garage" lazy="false">
    ...
  </class>
</hibernate-mapping>
```

Eğer `Car` için 2. seviye önbelleği işleme koymuşsak, `lazy="false"` kullanımı aradığı değerleri önbellekten alacaktır. Nesne önbellekte bulunamaz ise, veri tabanından ek bir `SELECT` işletilecektir.

5.4.7 Hibernate Yükleme (Fetching) Stratejileri

Tembel yükleme işaret edilen nesnelerin "ne zaman" yükleneceğini belirtiyorsa, yükleme stratejisi nesnelerin "nasıl" yükleneceğini belirler. Tembel yükleme şartlarında, işaret edilen bir nesne önbellekte yok ise, ana nesnenin bir `SELECT` ile yüklenmesinden her ilişki için ek bir `SELECT` gerekecektir. Bundan kaçış yoktur. Bu da her durumda kötü değildir, fakat uygulamamızın kullanım düzeni her ana nesnenin detayından sonra her ilişkide olunan nesnenin detayını almayı gerektiriyor ise, ek `SELECT` yöntemi veri tabanına erişim açısından optimal olmayacaktır. 9. bölümde belirttiğimiz gibi veri tabanları *kümelerle* çalışmayı severler; 10 tane satırı işleyen 10 SQL komutu yerine, tek SQL ile 10 satırı işlemek veri tabanları için daha tercih edilir kullanım şeklidir.

Hibernate, ek SELECT problemine [6] çözüm olarak, `fetch="join"` seçeneğini getirmiştir. Bu ayar, eğer bir Hibernate ilişkisi üzerinde tanımlanırsa, ana nesneyi yükleyen `SELECT`'e otomatik olarak bir `JOIN` ibaresi eklenecektir. Bu JOIN, ikinci nesnenin ana nesne ile ilişkide olan *tüm satırlarını* tek bir kerede yükler.

```
<set name="cars"
     inverse="true"
     fetch="join"
     >
  <key>
    <column name="garage_id"/>
  </key>
  <one-to-many class="Car"/>
</set>
```

Bu kullanımı test etmek için `HibernateRelFetchSelect` projesine bakabilirsiniz. Testimiz şudur:

```
Session s = HibernateSession.openSession();
HibernateSession.beginTransaction();
Garage garage = (Garage) s.get(Garage.class, new Integer(1));
for (Iterator it = garage.getCars().iterator(); it.hasNext();) {
    Car car = (Car)it.next();
    System.out.println(car.getLicensePlate());
}
```

Test programını işletirseniz, `fetch="join"` kullanılmadığı zaman (Hibernate olağan `fetch` değeri `fetch="select"` değeridir) şu çıktıyı görürsünüz:

```
...
Hibernate: select garage0_.id as id0_, garage0_.description as
descript2_1_0_ from garage garage0_ where garage0_.id=?

Hibernate: select cars0_.garage_id as garage3___, cars0_.license_plate as
license1___, cars0_.license_plate as license1_0_, cars0_.description as
```

[6] Bu probleme literatürde "N+1 problemi" ismi de veriliyor

```
descript2_0_0_, cars0_.garage_id as garage3_0_0_ from car cars0_ where
cars0_.garage_id=?
...
```

Görüldüğü gibi, Garage ile tek Car ilişkidedir, ve ek `SELECT` problemi yüzünden iki tane `SELECT` üretilmiştir. Eğer bu `SELECT` sorgularını teke indirmek istiyorsak, `fetch="join"` ayarını verdikten sonra testi tekrar işletiriz, ve o zaman şu sonuç gelir:

```
..
Hibernate: select garage0_.id as id1_, garage0_.description as
descript2_1_1_,cars1_.garage_id as garage3___, cars1_.license_plate as
license1___,cars1_.license_plate as license1_0_, cars1_.description as
descript2_0_0_,cars1_.garage_id as garage3_0_0_ from garage garage0_ left
outer join car cars1_ on garage0_.id=cars1_.garage_id where garage0_.id=?
..
```

Burada tek bir SQL üretildiğini görüyoruz, demek ki ek `SELECT` problemi halledilmiştir. `Car` nesneleri, bir `left outer join` komutu ile artık `Garage` ile birlikte *tek sorgu ile* yüklenebilir durumdadır.

5.4.8 JBoss Kümesi (Web Uygulamaları İçin)

Bir küme (cluster), birden fazla servis noktasının (node) birarada çalışmasından meydana gelen kütledir. Bu servis noktalarının genelde ortak bir amacı vardır. Bir servis noktası bir bilgisayar da olabilir, aynı bilgisayarda çalışan birden fazla süreç (process) de olabilir. JBoss dünyasında bir küme iki şeyi yapar: Çökme Toleransı (Fault Tolerance) ve Yük Dağıtımı (Load Balancing).

Çökme Toleransı bir bilgisayarda çökme olduğu anda, kullanıcının bağlı olduğu servis noktasının değiştirilebilmesi ve bu değişimin kullanıcıya fark ettirilmeden yapılabilmesidir. Web dünyasında bunun tercümesi kullanıcının oturum (session) bilgisinin başka bir makinada her zaman kopyasının olması, ve çökme anında kullanıcının kopyaya doğru yönlendirilmesi işlemidir.

Yük Dağıtımı ölçekleme amaçlı olarak eşzamanlı kullanıcıların yükünü birden fazla makinaya dağıtarak tek servis noktaları üzerindeki yükü (bağlantıyı) azaltma çabasına denir. Çökme Toleransı ve Yük Dağıtımı terimlerini birbiri ile karıştırmamak gerekir.

Kümeleme hakkında konuşurken telâfuz edilen bir diğer özellik de, Yüksek Mevcudiyet (High Availability) kavramıdır. Bu kavram bir kullanıcının bağlanacak bir servis bulma şansını ve bu servisten gelecek cevap süresinin (response time) her zaman mâkul çerçevede olması gerekliliğine/garantisine verilen genel addır. Çökme Toleransı desteği, her zaman Yüksek Mevcudiyet anlamına gelir. Ama tersi her zaman doğru olmayabilir çünkü çökme toleransı, *veri doğruluğu* ile çok yakın alakalıdır; Kullanıcı, arka planda çökme olsa da hem çökmeyi hissetmemeli, hem de o anda girmekte olabileceği *verisinde* bir yanlışlık görmemelidir. Finans dünyasında yapılan elektronik işlemler bunun tipik örneğidir. Fakat

kıyasla Yüksek Mevcudiyet, her zaman veri doğruluğunu gerektirmez, belli yüzdelerde veri yanlışlığına rağmen ayakta olan bir bilgisayarları bulmamız ihtimali yüksek ise, Yüksek Mevcudiyete sahibiz demektir. Telco dünyası her zaman veri doğruluğu gerektirmeyen yüksek mevcudiyet için güzel bir örnektir.

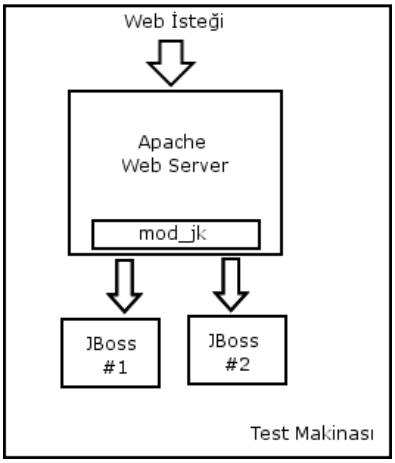
Bu bölümde, Web uygulamamızı tek bir JBoss üzerinden değil, iki JBoss servisinden oluşan bir kümede nasıl çalıştırabileceğimizi öğreneceğiz.

Apache ve JBoss Kümesi

Aynı uygulamayı birden fazla JBoss üzerinde çalıştırınca, şu sorulabilir: Dışarıdan gelen Web isteklerinin hangi JBoss'a gideceğine kim karar verecektir? Bu sorunun cevabı, "Web isteğini ilk karşılayan servis noktasından" başkası değildir. Apache Web Server (port 80 üzerinde işlem yapan) Web isteklerini ilk karşılayan yer olarak, gerekli JBoss servisine aktarma görevini yapabilir.

Apache'nin bu görevi yerine getirmesi için, Apache üzerinde mod_jk adında bir eklenti kurmamız gerekiyor. Mod_jk kurulduktan sonra, üzerinde elimizdeki tüm JBoss servislerini kayıt ederiz, ve böylece Apache'nin JBoss servislerinden haberi olur.

Mod_jk, JBoss servisleri ile AJP port'u ve AJP protokolü üzerinden iletişim kurar. AJP protokolu, HTTP protokolu gibi Web bilgisi servis etmek için yazılmıştır, fakat AJP tek başına tarayıcınıza HTML servis etmek için değil, Web Server programı ile beraber çalışarak Servlet teknolojisi ile Apache arasında bağlantı kurulabilmesi için kullanılır. Şekil 5.25 üzerinde böyle bir kurulumu görüyoruz.



Şekil 5.25: Aynı Makinada Küme: Bir Apache ve İki JBoss

Optimal olan Web fiziksel mimarisi, statik içeriğin (jpg, gif dosyaları gibi) Apache tarafından servis edilmesi, sadece Servlet işlemlerinin JBoss'a bırakılmasıdır.

Bunun için tüm statik içerik Apache dizinleri içinde olmalıdır. Bunun nasıl yapılacağını bu bölümde göreceğiz. Özet olarak ölçekleme amaçlarımız için bir JBoss kümesi, hızlandırma amaçlarımız için statik içerikleri Apache'ye işletme tekniklerini uygulayacağız.

HttpSession

Bir Web uygulamasının `HttpSession` üzerine bilgi koyması muhtemel olduğu için, küme ortamında bile, oturum bağlılığı (session affinity) kavramını küme ortamına taşımamız gerekecek. Yâni bir kullanıcı ilk web isteğinin karşılanması için JBoss #1'e gittiyse, aynı oturum altındaki ikinci isteği de aynı JBoss'a yönlendirilmelidir, çünkü hafızada (`HttpSession`) tutulan bilgiler, o JVM üzerindedir.

Fakat oturum bağlılığı kullanımı, JVM'in çökmesi durumunda bir zorluk yaratır, çünkü eğer JVM belleğinde (`HttpSession`) önemli bazı bilgiler tutuyorsak ve onu kaybedersek, yeni isteklerin mod_jk tarafından öteki JBoss'a yönlendirilmesi yeterli değildir: Bellekteki bilgilerin öteki JVM'ler üzerinde de tutuluyor olması gerekir. Yoksa yönlendirmeden sonra yeni JVM üzerinde beklenen değerler bulunamayacaktır. JBoss, bu problemi, her servisi birbiriyle yayın bazlı bir protokol üzerinden konuşturarak çözmüştür. Her Servlet üzerinde tutulan `HttpSession` içeriği, aynı kümenin parçası olan *öteki bir JBoss'a* JGroups yayın kütüphanesi ile kopyalanır. Oturum Kopyalama (Session Replication) için gereken ayarları bu bölümde göreceğiz.

Kavramları anlattığımıza göre, Şekil 5.25 üzerindeki kümeyi kurmaya başlayabiliriz.

Kurmak

Apache'yi Linux üzerinde kurmak için, kaynaklarından derlememiz gerekiyor. Alttaki adres

```
http://godel.cs.bilgi.edu.tr/mirror/apache/httpd/binaries/linux/
```

üzerinden `httpd-2.0.50-i686-pc-linux-gnu.tar.gz` adlı kaynak dosyayı indirin. Bu dosyayı herhangi bir dizinde açın. Bu dizine `APACHE_SOURCE` diyelim. Şimdi aşağıdaki komutları komut satırından uygulayın.

```
cd APACHE_SOURCE

./configure --with-layout=Apache
           --prefix=/usr/local/apache
           --enable-rule=SHARED_CORE --enable-module=so

make

make install
```

Dizin `/usr/local/apache`, derleme ve install bittikten sonra Apache işler dosyalarının bulunacağı yer olacaktır. Bu dizine giderek, sonucu kontrol edebilirsiniz: `bin/`,

conf/ gibi dizinler görmeniz gerekiyor. Test olarak, Apache'yi hemen başlatıp durdurabilirsiniz.

```
/usr/local/apache/bin/apachectl start
```

```
/usr/local/apache/bin/apachectl stop
```

Mod_jk'yi de, aynen Apache gibi, kaynaklarından derleyeceğiz. Alttaki adres

```
http://www.apache.org/dist/jakarta/tomcat-connectors/jk/source
```

üzerinden jakarta-tomcat-connectors-1.2.10-src.tar.gz paketini indirin. Paketi açın (MODJK_SOURCE diyelim) ve bu dizinin altında, MODJK_SOURCE/jk¬ /native/ dizinine gidin. Derlemek için şu komutları kullanın:

```
./configure --with-apxs=/usr/local/apache/bin/apxs --enable-EAPI
```

```
make
```

Derleme bittikten sonra, MODJK_SOURCE/jk/native/apache-2.0/ altında mod_jk'nin kütüphanesi yaratılmış olmalıdır. Bu dosyayı alıp, Apache altına atmamız gerekiyor.

```
cp ./apache-2.0/mod_jk.so APACHE_HOME/modules
```

Artık, Apache tarafında ayarları yapmaya başlayabiliriz. Apache'nin kurulmuş olduğu /usr/local/apache dizinine gidelim, ve conf dizini altında şu dosyaları yaratalım.

Liste 5.1: httpd.conf

```
# Include mod_jk configuration file
Include conf/mod-jk.conf
```

Liste 5.2: mod-jk.conf

```
# Load mod_jk module
# Specify the filename of the mod_jk lib
LoadModule jk_module modules/mod_jk.so

# Where to find workers.properties
JkWorkersFile conf/workers.properties

# Where to put jk logs
JkLogFile logs/mod_jk.log

# Set the jk log level [debug/error/info]
JkLogLevel info

# Select the log format
JkLogStampFormat "[%a %b %d %H:%M:%S %Y]"
```

```
# JkOptions indicates to send SSK KEY SIZE
JkOptions +ForwardKeySize +ForwardURICompat -ForwardDirectories

# JkRequestLogFormat
JkRequestLogFormat "%w %V %T"

# Mount your applications
JkMount /kitapdemo/*.do loadbalancer

# You can use external file for mount points.
# It will be checked for updates each 60 seconds.
# The format of the file is: /url=worker
# /examples/*=loadbalancer
JkMountFile conf/uriworkermap.properties

# Add shared memory.
# This directive is present with 1.2.10 and
# later versions of mod_jk, and is needed for
# for load balancing to work properly
JkShmFile logs/jk.shm

# Add jkstatus for managing runtime data
<Location /jkstatus/>
    JkMount status
    Order deny,allow
    Deny from all
    Allow from 127.0.0.1
</Location>
```

Liste 5.3: workers.properties

```
# Define list of workers that will be used
# for mapping requests
worker.list=loadbalancer,status
# Define Node1
worker.node1.port=8109
worker.node1.host=localhost
worker.node1.type=ajp13
worker.node1.lbfactor=1
worker.node1.cachesize=10

# Define Node2
worker.node2.port=8209
worker.node2.host=localhost
worker.node2.type=ajp13
worker.node2.lbfactor=1
worker.node2.cachesize=10
```

```
# Load-balancing behaviour
worker.loadbalancer.type=lb
worker.loadbalancer.balance_workers=node1, node2
worker.loadbalancer.sticky_session=1
worker.loadbalancer.local_worker_only=1
worker.list=loadbalancer

# Status worker for managing load balancer
worker.status.type=status
```

`worker.node1.lbfactor` ve `worker.node2.lbfactor` parametrelerinin değerine göre mod_jk, hangi küme birimine daha fazla yük aktarması gerektiğini anlar. Eğer `node1.lbfactor=100` ve `node2.lbfactor=200` olsaydı, `node2`'ye `node1`'e nazaran iki kat daha fazla yük aktarılırdı.

`worker.loadbalancer.sticky_session=1` parametresi, oturum bağlılığı (session affinity) kavramını kullanacağımızı göstermektedir (`=1`)

Liste 5.4: uriworkermap.properties

```
# Simple worker configuration file
#
# Mount the Servlet context to the ajp13 worker
/kitapdemo/*.do=loadbalancer
```

Ayar dosyaları `mod-jk.conf` ve `uriworkermap.properties` içinde yük dağıtıcıya (`loadbalancer`) iletilmek üzere sadece `/kitapdemo/*.do` URL düzeninin seçilmiş olduğuna dikkat edelim. Böylece, başta belirttiğimiz gibi, sadece dinamik içeriği (`.do` ile biten URL'ler, yâni Struts Action'ları) JBoss'a iletmiş oluyoruz. Geri kalan her türlü Web isteğini Apache karşılayacaktır. Bunu, statik içeriği daha hızlı servis etmek için yapıyoruz. JBoss içindeki HTTP servisi, büyük statik içerik yüklerini kaldırmak için uygun değildir. Apache Web Server'ı bu konuda en hızlı servislerden biridir.

Sitemizi oluşturan statik içeriğin Apache tarafından karşılanabilmesi için de, Apache'nin o içeriğe sahip olması gerekmektedir. O zaman statik içeriğimizi, geliştirme dizini olan `src/pages` olarak alıp, olduğu gibi `/usr/local/apache¬ /htdocs/kitapdemo` altına kopyalamamız gerekiyor. `htdocs` dizini, Apache için en üst seviye içerik dizini olarak kabul edilir. Apache işleten `localhost` makinanızı tarayıcınızdan `http://localhost/page.html` ile ziyaret edersek, Apache, `pages.html` adlı dosyayı `htdocs` altında arayacaktır.

Artık JBoss servislerini (kümesini) hazırlayabiliriz. Örnek web projesi olarak `StrutsHibPerformance`'ı kullanacağız.

1. Projeyi önce derleyip (A.1 bölümünde anlatıldığı gibi) `JBOSS/server/¬ default/deploy` altına gönderelim.

2. `JBOSS/server/default/all` dizininin, `JBOSS/server/default/node1` ve `JBOSS/server/default/node2` olarak iki kopyasını çıkartalım.

3. Daha önceden `JBOSS/server/default/deploy` altına göndermiş olduğumuz (derleme ile) `kitapdemo.sar` projesini, `node1` ve `node2` dizinleri altına kopyalayalım.

4. Her iki dizinde de, `kitapdemo.sar/kitapdemo.war/WEB-INF/classes¬ /web.xml` dosyasına girelim ve `<distributable/>` etiketini şu şekilde ekleyelim:

```
<web-app>
   ...
   <distributable/>
</web-app>
```

5. A.4.3 bölümünde anlatıldığı gibi, `node1` ve `node1` altındaki JBoss kuruluşunu, değişik port'lar kullanması için ayarlayalım.

6. `JBOSS_HOME/server/node1/deploy/jbossweb-tomcat50.sar` ve `JBOSS¬ _HOME/server/node2/deploy/jbossweb-tomcat50.sar` altında olan `server¬ .xml` dosyalarındaki `Engine name="jboss.web"` diyen bloğu şöyle değiştirelim:

```
<Engine name="jboss.web" defaultHost="localhost" jvmRoute="node1">
    ..
</Engine>
```

   ve

```
<Engine name="jboss.web" defaultHost="localhost" jvmRoute="node2">
    ..
</Engine>
```

7. `JBOSS_HOME/server/node1/deploy/jbossweb-tomcat50.sar/META-INF` ve `JBOSS_HOME/server/node2/deploy/jbossweb-tomcat50.sar/META¬ -INF` altındaki `jboss-service.xml` dosyalarında `UseJK` komutunun verildiği satırı bulalım, ve onu şöyle değiştirelim:

```
<attribute name="UseJK">true</attribute>
```

Küme hazırlığımız tamamlandı. Apache, mod_jk üzerinden iki JBoss servisini tanıyor, ve onların arasında yük dağıtımı yapmak üzere hazır. JBoss servislerinin her biri de ayrı port'larda çalışmak üzere ayarlandılar (küme ortamında port değişikliği için A.4.3 bölümüne bakınız). Her iki JBoss da başlatıldıktan sonra, birbirleri ile iletişime geçerek `HttpSession` içeriklerini ötekine kopyalayacaklar, ve çökme durumunda Apache mod_jk, yeni istekleri *çökmemiş* makinaya yönlendirdiğinde, kullanıcı uygulamayı sanki hiç çökmemiş gibi kullanmaya devam edecektir.

Kümeyi başlatmak için `JBOSS_HOME` altından

```
bin/run.sh -c node1
```

```
bin/run.sh -c node1
```

komutlarını uygulamalıyız. Apache için ise `/usr/local/apache` altında

```
/usr/local/apache/bin/apachectl start
```

komutunu kullanırız. Üç servisin tamamı ayağa kalktıktan sonra, tarayıcınızı `http://makina-ismi/kitapdemo/main.do` adresine yönelttiğinizde Apache'nin Web isteğini JBoss'lardan birine yönelttiğini ve başlangıç sayfasını bastığını göreceksiniz. Yük dağıtımını test etmek için, ikinci bir tarayıcı açın, ve tekrar aynı adrese gidin. Bu sefer yönlendirme yapılan JBoss, ikincisi olacaktır. Çökmeye dayanıklılığı test etmek için, birinci JBoss'u `Control C` ile çökertin, ve her iki tarayıcıyı da günceleyin. Uygulamanın hiçbir şey olmamış gibi işlemeye devam ettiğini göreceksiniz.

Not: JBoss servislerini başlattığınızda eğer

```
java.net.SocketException: bad argument for IP_MULTICAST_IF: address
not bound to any interface
```

hatası gelirse, Linux servisiniz multicast paketlerini yönlendirmeye (route) hazır değil demektir [6, sf. 93] . Multicast desteği eklemek icin, komut satırında

```
route add -net 224.0.0.0 netmask 240.0.0.0 dev eth0
```

komutunu `root` olarak işletmelisiniz. Aynı komutu sürekli işletmeyi hatırlamaktan kurtulmak için bu kullanımı bir başlangıç script'inin içine koymanız isabetli olur. Ayrıca, JVM'in IPv6 (internet protokolü sürüm 6) kullanıyorsanız, yeni bir network hatası çıkacaktır. Bu hatadan kurtulmak için, Linux seviyesinde IPv6 yerine IPv4 kullanmanız yeterlidir[7]. `JBOSS/bin/run.conf` içinde `JAVA_OPTS` değişkenine şu değeri atamanız yeterli olacaktır

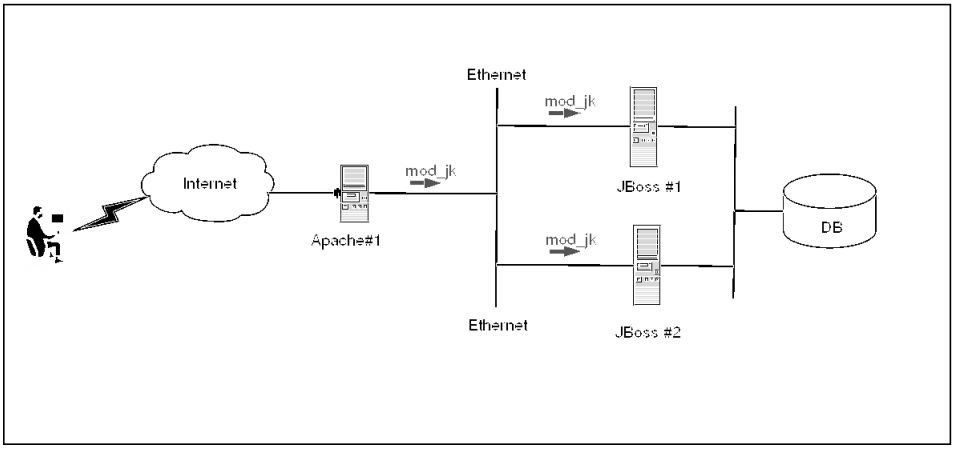
```
JAVA_OPTS="-server .. -Djava.net.preferIPv4Stack=true"
```

Not 2: Eğer JBoss kümesi başladığında ayrı makinalardaki JBoss'lar birbirini görmez ise, Linux üzerindeki birimleriniz `127.0.0.1` adresini bağlanıyor olabilir. Bu durumda JBoss'unuz network'teki diğer IP'leri görmeyecektir. Bunun önünce geçmek için `run.conf` içinde set edilen `JAVA_OPTS`'a meselâ `192.168.1.1` IP adresi taşıyan Linux makinası için `-Dbind.address=192.168.1.1` ibaresini eklemeniz gerekecektir.

[7] http://wiki.jboss.org/wiki/View.jsp?page=IPv6

Fiziksel Küme Yapıları

Üstte tarif ettiğimiz küme kuruluşu, hepsi aynı makinada çalışan bir Apache ve iki JBoss servisi üzerinden hazırlandı. Gerçek bir Web sitesinin fiziksel yapısı, daha fazla donanım birimi içerecektir, çünkü ölçekleme demek, problemin üzerine daha fazla donanım atmak, daha fazla beygir gücü ile eşzamanlı yapılan islem sayısını arttırmaktır. Kümenin donanım yapısı açısından çoğunlukla takip edilen yöntem, her JBoss servisinin ayrı bir Unix makinası üzerinde olmasıdır. Böyle bir yapıyı Şekil 5.26 üzerinde görüyoruz.



Şekil 5.26: Küme: Ayrı Makinalarda Bir Apache İki JBoss

Bu şekle göre, ayrı bir makinadaki Apache, ilk isteği alarak mod_jk üzerinden yine ayrı makinalarda olan JBoss servislerinden birine yönlendirmektedir. Bu da kullanılabilir, ve ölçeklenebilir bir mimaridir. Daha fazla eşzamanlı kullanıcıyı idare edebilmek için, JBoss kümesine yeni bir makina eklenir, ve mod_jk ile kayıt ettirilir.

Daha da ciddi bir mimari için, Şekil 5.27'deki mimariyi tavsiye ediyoruz. Bu mimaride, kümedeki JBoss'ların çökmesine hazır olunduğu gibi, Apache'lerden birinin de çökmesine bile hazırız. Görüldüğü gibi, tek Apache yerine iki Apache kullanılmıştır, ve her iki Apache de kümedeki tüm JBoss'lardan haberdardır.

İki Apache arasında seçim yapmak için, Cisco 11150 markasındaki bir switch donanımını kullanacağız. Bu switch, aynen Apache'nin Java `HttpSession` seviyesinde yapışkanlık sağladığı gibi, pür HTTP bazında bir yapışkanlık sağlar, yâni aynı müşteriyi (client) aynı Apache'ye yönlendirir. Bu switch, çok basit bir yönlendirme yaptığı için çökme şansı daha azdır (kıyasla Apache bir yandan statik HTML servislemesi de yapar). Switch'in Yüksek Mevcudiyet gereklilikleri network admin'lerinin sorumluluğu altında olacaktır. Biz eğer sistemimizi

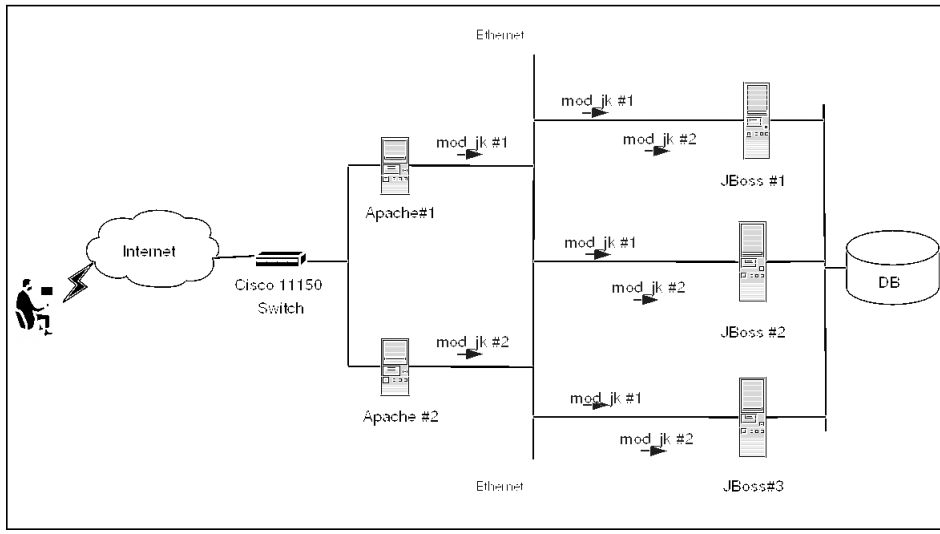
bu yazıda tarif edilen seviyeye getirdiysek, bir kurumsal teknik mimar olarak kontrolümüz altındaki tüm noktaları elimizden geldiğince sağlama almış durumdayız demektir.

5.4.9 EJB Kümesi Yaratmak

4.5 bölümünde EJB Session Bean teknolojisini işledik. Bu bölümde işlenen şekliyle EJB Session Bean nesneleri, kendilerini JNDI üzerinden bulan, bağlanan ve elinde referansını tutan dış müşterilere metotlarını sunabiliyordu. Gerçek bir projeye örnek olarak bir Swing uygulaması JNDI üzerinden erişebileceği EJB Session Bean nesnelerinde metot çağrıları yaparak (CommandHandler mimarisinde tek nesne ve `executeCommand` metotuna) servis tarafına gerekli işleri yaptırdığı bir yapıyı düşünebiliriz. Şekil 5.28 üzerinde böyle bir yapıyı görüyoruz.

Bu yapı, dikkatimizi çekebileceği gibi, 4.5 bölümünde tarif edilen yapıdan değişiktir. Yeni yapıda, Swing uygulaması EJB'lere ulaşırken bir "akıllı yer tutucu" (smart proxy) üzerinden bağlanmakta ve yerel (local) JNDI yerine, global JNDI servisi kullanmaktadır. Yerel JNDI yerine global JNDI kullanılmasının sebepleri şunlardır:

- 5.4.8 bölümündeki Web kümesi gerekliliklerine benzer bir şekilde, kodlama basitliği açısından, çağıran tarafta tek ve herkese ortak bir giriş noktasına ihtiyaç vardır.



Şekil 5.27: Küme: Final Fiziksel Mimari

- Dağıtıcı merkez nokta, merkezi bir yer olduğu için, kümedeki birimlerde olan yüke göre, yeni isteği yükü en az olan EJB servisine yönlendirme işlemini gerçekleştirmek için en uygun yerdir.

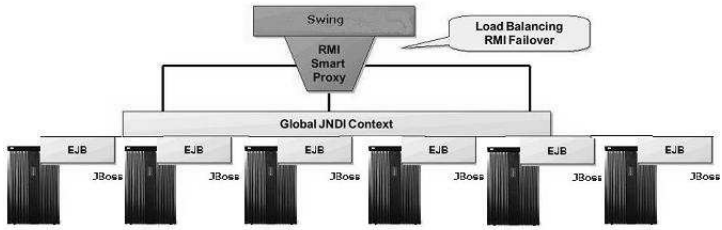
- Giriş noktası, küme birimlerinden ayrı olduğu için tek çökme noktası (single point of failure) yapısından bizi kurtarır.

Bir diğer değişiklik ise, yerel yerine global ve yüksek mevcudiyeti olan JNDI (HA-JNDI) kullanıldığı zaman, geriye gelen normal bir EJB proxy'nin değil, akıllı bir proxy'nin olmasıdır. Akıllı proxy'nin kullanılmasının sebebi ise, küme içindeki diğer EJB küme birimlerinden haberdar olan bir proxy'ye olan ihtiyaçtır. Çökmeden kurtulma tekniğini kullanmak için böyle bir proxy'ye ihtiyaç vardır. Senaryo olarak şunu düşünün: Meselâ Swing uygulamasının HA-JNDI'dan aldığı, makina #1 üzerinde EJB #1'e işaret eden bir eden bir proxy olsa ve o sırada makina #1 çökse, elimizdeki referansın bize hissettirilmeden *çökmemiş* olan makina #2'de olan bir EJB #1 kopyasına yönlendirilerek Swing uygulamasının hiçbir şey olmamamış gibi işine devam edebilmesini isteriz. Akıllı proxy, küme kavramından haberdar bir nesne olarak bu işi başarılı bir şekilde yapabilir.

JBoss Üzerinde EJB Kümesi Kullanmak

JBoss kullanarak bir EJB kümesi yaratmak için, üç şey yapılması gerekiyor.

1. Projenizde EJB ayarları yapan dosyalardan biri olan `dd/jboss.xml` içinde EJB'mizin küme hâlinde çalışacağını belirtmeliyiz. Bunun için bahsi geçen dosyada şu şekilde bir ayar değişikliği gerekir:



Şekil 5.28: Bir EJB Kümesi

```
<jboss>
   <enterprise-beans>
      <session>
         <ejb-name>test/MyTestSession</ejb-name>
         <jndi-name>ejb/test/MyTestSessionBean</jndi-name>
            <clustered>True</clustered>
      </session>
   </enterprise-beans>
   <resource-managers>
   </resource-managers>
   </resource-managers>
</jboss>
```

Bu listede, normâl EJB'ye göre olan tek değişiklik, `<clustered>` etiketi altında `True` değerinin verilmiş olmasıdır. Olağan (default) değer olarak bu etiket, `False` değerini taşıyacaktır.

2. Kümemize ait olan servisleri `run.sh` ya da `run.bat` ile başlatmadan önce, işler kodlarımızın `JBOSS_HOME/server/all` dizinine gönderilmiş olması gerekmektedir. Tabii servisimizi başlatırken de `sh run.sh -c all` komutunu kullanmalıyız. Eğer aynı makinada birden fazla küme birimi (JBoss servisi) kullanacaksak, A.4.3 bölümünde anlatılan port değişiklik tekniğini kullanmalıyız.

3. HA-JNDI, JBoss üzerinde JNDI servisinden değişik bir port'tan servis edilir. O yüzden HA-JNDI'ı bulan kodların bağlandığı port'un bilinmesi gerekiyor. Bu port değerinin ne olduğunu, JBoss başlangıç log mesajlarına bakarak bulabiliriz. Meselâ, aşağıdaki gibi bir açılış ekranında

```
....
17:50:16,237 INFO [SnmpAgentService] SNMP agent going active
17:50:17,739 INFO [DefaultPartition] Initializing
17:50:18,149 INFO [STDOUT]
-------------------------------------------------------
GMS: address is localhost:1032 (additional data: 14 bytes)
-------------------------------------------------------
17:50:20,262 INFO [DefaultPartition] Number of cluster members: 1
17:50:20,262 INFO [DefaultPartition] Other members: 0
17:50:20,272 INFO [DefaultPartition] Fetching state (will wait
for 30000 milliseconds):
17:50:20,923 INFO [HANamingService] Listening on /0.0.0.0:1100
...
```

Bu ekranda `HANamingService`'in `1100` numaralı port'ta dinlemeye başlamış olduğu bildirilmektedir. İşte bu port, HA-JNDI için kullanmamız gereken port'tur. Ayrıca küme ortamında birden fazla makina olabileceği için, ve

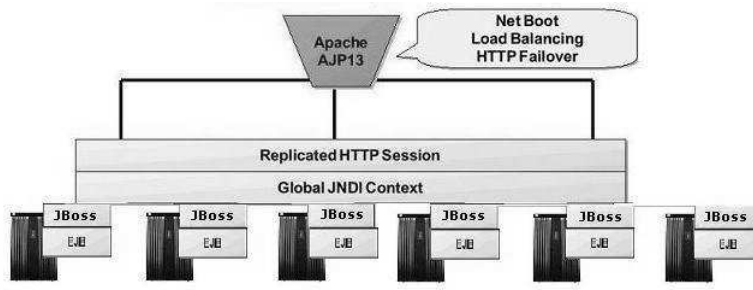
bu makinaların çökme ihtimaline karşı, JBoss HA-JNDI kullanımı `Context.PROVIDER_URL` için birden fazla makine ismi tanımlamamıza izin verir. O zaman bağlantı kodlarınız şu şekilde olmalıdır.

```
MyTestSession beanRemote;
InitialContext ic = null;

Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY,
     "org.jnp.interfaces.NamingContextFactory");
p.put(Context.URL_PKG_PREFIXES, "jboss.naming:org.jnp.interfaces");
p.put(Context.PROVIDER_URL, "host1:1100, host2:1100, host3:1100");
try {
    ic = new InitialContext(p);
    Object objref = ic.lookup("ejb/test/MyTestSessionBean");
    ...
```

Böylece `host1`,`host2` ve `host3` isimli makinelerde çalışmakta olan HA-JNDI servislerinden hangisi ayakta ise ondan bir EJB referansı talep edebiliyoruz. `lookup` yöntemi sırasıyla listedeki her makinadaki HA-JNDI'ya gidecek, eğer makinayı bulamazsa bir sonrakine devam edecektir.

Yanlış Mimariler

4.3.3 bölümünde, Web katmanı içeren mimarilerde EJB kullanmanın optimal olmadığından bahsetmiştik. Sebeplerini burada detaylı olarak açıklayalım;



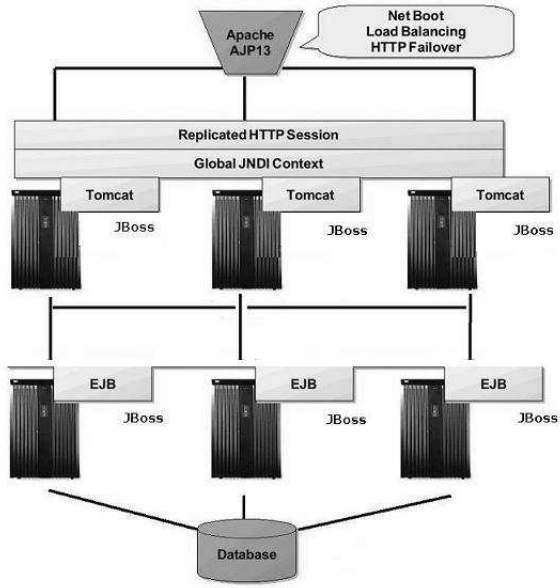
Şekil 5.29: Yanlış EJB Kümesi - Yerel Çağrı Yapılacaksa EJB'ye Ne Gerek Var?

- Web EJB'yi Yerel Çağırıyor: Şekil 5.29 üzerinde gösterilen yapıda, şu soru sorulabilir: Eğer Web kodlarından yerel çağrı yapılacaksa, dağıtık mimaride çalışmaya yarayan EJB teknolojisinin kullanılması gerekli midir? Bazı mimarlar tarafından "ileride ayırmamız rahat olsun diye" yerel bile çağırılsa EJB kullanılmış olması, bir seviyede belki geçerli olabilirdi, fakat

birbirinden ayrı olan Web + EJB mimarisinin başka sebeplerden ötürü optimal olmaması, bu tasarımın başarısına gölge düşürür.

- Web ile EJB Ayrı JVM'lerde: Şekil 5.30 üzerindeki fiziksel olarak ayrı Web ve EJB şartlarında, performans açısından network'e her çıkışımızda ek bir bedel öderiz. Bu yüzden Web ortamından işlem mantık kodlarımızı EJB içine koymak yerine, düz bir class üzerinden Java `import` ile kendi JVM'imize dahil ederek yerel çağrı yapmamız daha hızlı olacaktır. Kodlama açısından da düz class kullanmak, tek bir bileşen için Home, Remote ve Bean class'ları yazılmasını gerektiren EJB seçeneğinden daha az kodlama gerektirecektir. Kodlama külfeti argümanı, yeni JDK 5.0 annotation teknolojisi üzerinden EJB 3.0 üzerinden kullanılıyor olsa bile geçerlidir, çünkü bu sefer annotation'ların eklenmesi ve ayar dosyalarında gereken değişiklikler, yerel çağrı yapıldığı duruma göre ek kod külfeti anlamına gelecektir.

Fakat buna rağmen, Web + EJB mimarisi hakkında şu ek savunmayı da bazen işitebiliyoruz: "Web + EJB mimarisinde bazı kodlar bazı kodlardan fiziksel olarak ayrıldıkları için, bir parçanın çöküşü öteki parçayı etkilemez". Bu geçersiz bir savunmadır. Son kullanıcıdan başlayan bir Web isteğini bir zincir gibi düşünürsek, bir zincir, sadece en zayıf halkası kadar sağlamdır. A, B'yi çağırır, B C'yi çağırır ve B çökerse, A'nın da C'nin de çalışması hiçbir anlam



Şekil 5.30: Yanlış Bir EJB Kümesi - Web Katmanı ile EJB Konuşuyor

ifade etmeyecektir. Bu yüzden işlem mantığı kodlarının ayrı bir fiziksel makinada olması, çökmeden kurtulma açısından hiç bir yardımda bulunamaycak bir faktördür.

Son iddia, "Web + EJB durumunda bir kod öbeğinin (meselâ EJB) daha fazla işlem gücüne ihtiyacı varsa, onun fiziksel ölçeklenmesinin ayrı yapılabilmesi, meselâ ek bir EJB makinası ile sadece işlem gücü isteyen öbeklere işlem gücü verilebilme" iddiasıdır. Bu düşüncede, pür Web mimarisine kıyasla, ciddi eksiklikler vardır. Web + EJB fiziksel katmanları ayrı olan bir mimarinin pür Web fiziksel katmanı içeren bir mimariye nazaran yükünü optimal şansı sıfıra yakındır. Bunu matematiksel olarak şöyle ispatlayabiliriz.

Dağıtık Mimari Performans Teoremi 5.4.1. *Pür Web mimarisinde olan belli bir yük altındaki $n \in \mathbb{N}$ makina, kod içerisinde web için $w(w \in \mathbb{R}, 0 \leqslant w \leqslant 1)$, işlem mantığı için ise $e(e \in \mathbb{R}, 0 \leqslant w \leqslant 1)$ kadar zaman harcamaktadırlar. Ayrıca $e + w = 1$ olduğunu farz ediyoruz.*

Pür Web kodlarını Web için ayrı, işlem mantığı için ayrı EJB "fiziksel makinalara" geçirdiğimizde (ki Web makinalarının sayısı $fw \in \mathbb{N}$, EJB makinalarının sayısı $fe \in \mathbb{N}$ olsun) tam kullanım gören (fully utilized) tek dağılım, $fe = en$ olduğu şartlardadır. Aynı şekilde Web için tam kullanım $fw = wn$ şartlarında olur. Diğer tüm muhtemel dağılımlarda makinaların bir kısmı her zaman az yüklenmiş (underutilized) diğer kısmı da gereğinden fazla yüklenmiş (overutilized) olacaklardır.

Kanıt. Bir makina zamanına 1 değeri verelim. O zaman pür fiziksel Web şartlarında tüm makinalarda harcanan işlem mantığı makina zamanı, $e \cdot n$ olarak hesaplanabilir. Ayrı fiziksel Web ve EJB makinalarına geçilince, her EJB makinası başına düşen makina zamanı ise

$$E_{load} = \frac{e \cdot n}{fe} \tag{5.1}$$

olarak gösterebiliriz. E_{load} değerinin 1 çıkması için tek yol, görüldüğü üzere, $fe = e \cdot n$ olduğu şartlarda olacaktır. Diğer tüm muhtemel dağılımlarda, üstteki formülün böleninde yeralan fe değeri, bölümde olan $e \cdot n$'den ya büyük ya da küçük olacağı için sonuç 1'den ya büyük ya da küçük olacaktır. Ayrıca fe'nin $e \cdot n$'e eşit olması da düşük bir olasılıktır çünkü fe bir doğal sayı, $e \cdot n$ ise bir gerçek (real) sayıdır. O zaman EJB makinası ya az kullanılıyor olacak, ya da boğulacaktır.

E_{load}'un 1'den küçük çıkması şartı da iyi değildir, çünkü bu fe'nin $e \cdot n$'den büyük olduğu şartlarda olur ve bunun sonucu W_{load}'un 1'den büyük çıkmasıdır, bu da Web makinalarının boğulduğu anlamına gelir. Bunu şöyle ispatlayabiliriz;

$$
\begin{aligned}
f \cdot e &> e \cdot n && (5.2)\\
f \cdot e &> (1-w) \cdot n && (5.3)\\
f \cdot e &> n - w \cdot n && (5.4)\\
n - fe &< w \cdot n && (5.5)\\
fw &< w \cdot n \text{ çünkü } f \cdot e + f \cdot w = n && (5.6)\\
W_{load} &= \frac{w \cdot n}{fw} && (5.7)\\
W_{load} &\geqslant 1 && (5.8)
\end{aligned}
$$

□

Eğer bu teorinin, tersi yönünde ilerlesek, yâni tam kullanım (fully utilized) Web+EJB mimarisini pür Web mimarisine geçirirsek, o zaman dağılımın sancısız şekilde düzgün olacağını görürdük (bu teori, basitliği sebebinden ispatlanmayacaktır) çünkü ayrı fiziksel katmanlardan tek katmana geçiş yapıyoruz, ve Web ile işlem mantığı kodları aynı JVM içindedir. Bu durumda yük dağıtımı için tek gereken, Apache'nin kullanıcı yüküne göre gereken yönlendirmeyi yapmasıdır. O zaman şu sonuca varıyoruz: Kullanabilecek en az donanımla, Web + EJB mimarisi sadece çok küçük bir ihtimalle (neredeyse sıfır) pür Web mimarisiyle aynı performansı gösterebilecektir.

Web + EJB kullanımında pür Web kullanımına nazaran donanım ihtiyacı, bir makina zamanı kadar olsa bile, teorilerin ispatı sırasında yok farzettiğimiz bir gerçeklik daha vardır. Teoride network yavaşlatma faktörünü (latency) sıfır olarak farz ettik. Fakat, gerçek dünyada *ayrı* fiziksel EJB katmanına network üzerinden gitmek, uygulamamız üzerinde kesinlikle yavaşlatıcı bir faktör olacaktır. Sadece bu sebep için bile Web ortamında dağıtık nesneler kullanılması uygun değildir.

O zaman, örnek olarak 6 makinayı 3 Web 3 EJB olarak ayırmak yerine (Şekil 5.30), 6 makinanın hepsini Web (ve onun `import` ile dahil edip çağırdığı işlem mantığı kodlarına) ayırmak daha iyidir (Şekil 5.31 -ki bu mimari 5.27 üzerinde gösterilen ideal yapı ile aynıdır-).

Dışarıdan bağlananların *aynı anda* hem Web hem zengin önyüz (meselâ Swing) olabilmesi şartlarında bile tavsiyemiz, Web üzerinden EJB'ye bağlanılmamasıdır. Bu şartlarda işlem mantığı kodlarını pür Java kodlarına ayırıp hem Web'den hem EJB'lerden `import` ederek çağırmak, Web fiziksel katmanının EJB katmanına çağrı yapmasından daha optimal olacaktır.

5.4.10 Veri Tabanında İndeks Kullanımı

Bir kurumsal uygulamanın optimal çalışmasında, o uygulamanın veri tabanında gerçekleştirdiği sorguların performansı çok önemlidir. Eğer bu sorguların ne

kadar hızlı çâlıştığını anlamak istiyorsak, öncelikle sorguların ne olduğunu görmeliyiz. Bunun için `hibernate.cfg.xml` dosyasında

```
<property name="show_sql">true</property>
```

ayarını yapmalıyız. Bu ayar, Hibernate'in ürettiği tüm SQL sorgularını ekrana basacaktır. Böylece yavaş işlemesi muhtemel olan SQL'leri gözümüzle görebilmiş oluruz.

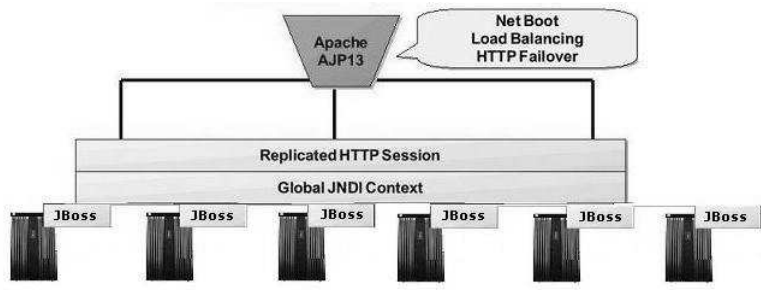
Fakat, eklememiz gerekir ki, şartların %99'unda Hibernate'in ürettiği SQL, bizim elle yazacağımız SQL'den daha optimal olacaktır. Ayrıca Hibernate arka planda, meselâ Oracle şartlarında "bind parameters" gibi hızlandırıcı teknolojileri kullandığı için, sorgu işletim hızı stored procedure hızına bile eşit hâle gelmektedir. Yâni, işin pür Hibernate tarafında bir problemimiz olmayacaktır; Hibernate'e has optimizasyon numaraları bu yüzden önbellekleme, fetching ayarları gibi başka bir seviyede yapılır.

Eğer tüm optimizasyonlara rağmen Hibernate veri erişimini yavaş olduğunu gözlemlersek, sebep, veri tabanında eklenmesi unutulmuş indeksler olabilir. İndekslerin yetersiz olması şartında zaten SQL ister üretilsin, ister elle yazılsın, yavaş çalışacaktır; En alt seviyedeki veri tabanı gerekli şekilde optimize edilmemiştir.

Veri tabanlarını optimize etmek için, Hibernate'in ürettiği SQL'in `WHERE` bölümünde kullanılan filtre şartlarına bakabiliriz. Bu filtre şartlarında kullanılan kolonların (meselâ bir SQL içinde kullanılan her kolonun kombinasyonu) için bir indeks var mıdır? Eğer

```
...
WHERE
  KOLON1 = '22'
  KOLON2 = 'VS VS'
  ;
```

gibi bir SQL üretilmiş ise, `KOLON1 + KOLON2` kombinasyonu üzerinde bir indeks olmalıdır. Eğer indeks var ise, bu indeks ile `WHERE` filtre şartları uyum gösterdiği anda veri tabanı bu indeksi işleme sokacaktır.



Şekil 5.31: Doğru Web Kümesi

Gözle kontrol yerine, gerekli yerde indeks olup olmadığını anlamak için mekanik bir yöntem de takip edebiliriz. Her veri taban ürününde bir SQL sorgu analiz programı mevcuttur, bu program bir sorgunun gerekli indekslerin kullanıp kullanmadığını gösterebilir. İndeks kavramını, analiz yöntemini, ve her taban ürünü için bunların nasıl kullanıldığını ayrıntılı şekilde 9.2.9 bölümünde okuyabilirsiniz.

İndeks eklerken dikkat edilmesi gereken bir husus, sadece ve sadece gereken kolon kombinasyonu için indeks eklememizin gerektiğidir. İndeksler, diğer pek çok şeyin aksine, "ne kadar fazla olursa o kadar iyi" olan katkılardan değildirler, çünkü veri tabanı bir tabloya her yeni satır eklediğinde o tablo üzerindeki olan indeksleri güncellemeye de mecburdur, ve bu güncelleme işleminin bir hız bedeli vardır. Bu sebeple eğer bir tabloda gereğinden fazla indeks var ise, `SELECT` işlemimizi hızlandırmış olsak bile, bu sefer `INSERT` işlemimizi yavaşlatmış oluruz. İndeksleme işlemini kararında yapmalıyız; O zaman indeksler performansımızı arttıran önemli bir faktör olabilirler.

Bölüm 6

Sonuç Ortamı

Bu Bölümdekiler

- Sonuç ortamına kod göndermek
- İşlemde olan bir kurumsal programı kontrol etmek

Kurumsal uygulamızı yazıp, test ettikten sonra, uygulamamızı işleme açmak için işler kodlarını sonuç ortamına göndermemiz gerekir. Kitabımızda tavsiye edilen sonuç ortamı, işletim sistemi olarak, Unix (Linux, Solaris) sistemleridir. O zaman kurumsal uygulamamızı Unix üzerindeki bir JBoss üzerine göndermek için, bazı yöntemler kullanmamız gerekecektir. Bu yöntemlere deployment başlığı altında bakacağız.

Kod gönderildikten sonra (ciddi sonuç ortamlarında) diğer bir problem ile karşı karşıya geliriz. Bir kurumsal uygulamayı sonuç ortamında gerçek kullanıcı yükü altında işletmek, test ortamında işletmekle aynı değildir. Komut satırından elle başlattığımız bir program, eğer çökerse ne olacaktır? Modern bir kurumsal uygulamanın üzerindeki gereklilikler düşünülürse (haftanın 7 günü ve her günün 24 saati ayakta kalmasının beklediğimiz sistemler) onlarca Unix makinasının oluşturduğu bir küme ortamında, sadece `run.sh` ile başlattığımız servislerin izlenemez, ve takip edilemez bir halde olduğu sonucuna varırız. Bu sorunu izleme (monitoring) başlığı altında çözeceğiz.

6.1 Kod Gönderimi

Bazı seminerlerimde kullandığım bir fiziksel mimari örneği vardır: "Bu sitenin kullandığı fiziksel mimari, küme hâlinde çalışan 20 tane Unix makinası içeriyor" diyerek bazı donanım istatistiklerini dinleyiciye sunarım. Bunu duyan katılımcılardan biri bazen şu soruyu sorar: "Biz 4 makinayı zor idare ediyoruz, bu adamlar 20 makinayı nasıl idare ediyorlar?".

Bahsedilen sayı hakikaten büyük bir sayıdır. Fakat örneği verilen türden bir yapıyı idare eden admin'in kullandığı o ufacık tekniği duyunca, sorunun cevabının aslında ne kadar basit olduğunu anlarsınız . Meselâ `kitapdemo.sar` adlı Web paketimizi, `host1`, `host2`, ... , `host100` olarak isimlendirilmiş 100 tane makinanın `/usr/local/jboss/server/all/deploy` dizini altına kopyalamamız gerektiğini düşünün (her makinada JBoss'un aynı yere kurulmuş olduğunu farz ediyoruz). Bu kopyalama işlemini, eğer düzeninizi iyi kurmuşsanız, şu şekilde yapabilirsiniz (`sh` scripting dili ile)

Liste 6.1: sendcode.sh

```
LIMIT=100
for ((a=1; a <= LIMIT ; a++))
do
  scp kitapdemo.sar "remoteuser@host\$a:/usr/local/jboss/server/all/deploy"
done
```

Ya da Windows BAT ortamında

Liste 6.2: sendcode.bat

```
@echo off
FOR /L %%n IN (1,1,100) DO call :copying %%n
```

```
goto end:
:copying
scp kitapdemo.sar remoteuser@host%1:/usr/local/jboss/server/all/deploy
:end
```

İste bu tek script ile paketimizi 100 makinayı tek seferde kopyalamış olduk. Peki bu nasıl oldu?

6.1.1 SSH ve SCP

Bir makinadan diğerine hem şifresiz hem de güvenli bir şekilde girip, orada komut işletmek, ya da oraya dosya kopyalamak için iki program vardır: `ssh` ve `scp`. Kullanmak için komut satırından (`ssh` için)

```
ssh host1 -l remoteuser
```

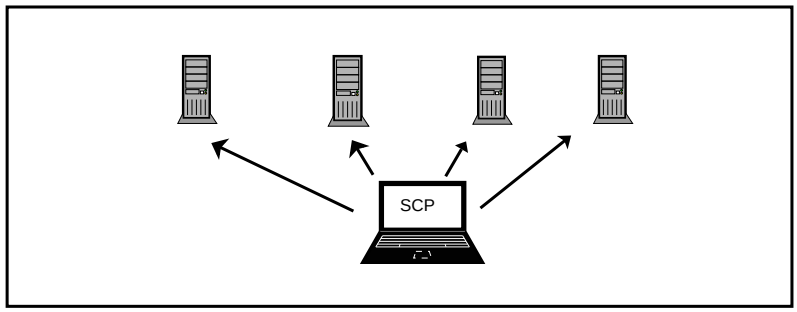
komutunu kullandığınızda `host1` adlı makinaya `remoteuser` kullanıcısı üzerinden login etmiş olursunuz. Ya da uzak makinada bir komut işletip sonucunu kendi makinanıza almak isterseniz (meselâ ikinci makinada listeleme komutu olan `ls` işletelim), şunu yaparız

```
ssh host1 -l remoteuser ls
```

Eğer uzak makinaya bir dosya kopyalamak istersek

```
scp file.txt remoteuser@host1:/tmp
```

Bu komut ile uzaktaki makinaya *sanki yerel dizinlerimiz arasında dosya kopyalıyormuş kadar rahat* bir şekilde bir dosya kopyalayabiliyoruz. Girişte bahsettiğimiz basit teknik işte budur. `ssh` ve `scp` bir kez kurulduktan sonra, uzaktaki makina, yerel makinanızın bir uzantısı hâline gelir. Bir makinayı uzaktan idare etmek demek, ya bir dosya değişimi, ya da bir komut işletmek demek olduğu için, bu iki programı kullanarak uzaktaki bir makinada yapamayacağımız şey yoktur.



Şifresiz Kullanımı Kurmak

Eğer `ssh`'i hiçbir ek ayar yapmadan kullanırsanız, (ilk kurduğumuz haliyle) her kullanışınızda size bir şifre sorulacaktır. Aynı şekilde `scp` komutu da böyle davranır. Fakat, biz meselâ yüz tane makina için arka arkaya `scp` ya da `ssh` kullanmamız gerekeceği için, şifre isteme işlemini iptal edip, güvenlik kontrolünü kullanıcıya sorulmayan *başka bir şekilde* yapmamız gerekiyor. Bu yöntem de, açık / gizli (public / private) anahtarlar kullanarak yapılan güvenlik kontrolüdür.

Windows üzerinde `ssh` ve `scp`'nin işler kodlarını kurmayı, A.8 bölümünde bulabilirsiniz. Linux üzerinde `ssh` ve `scp` genellikle otomatik olarak kurulur, eğer kurulmamışsa, Linux kurulum disklerinizde bu programı bulabilirsiniz, ya da admin'inize bu programları kurdurabilirsiniz.

Açık / gizli anahtar kurulumunu yapmak için şunları yapın: İlk önce kod gönderimi ya da uzaktan idareyi yapan yerel makinamızı tanıtan bir gizli anahtar, bir de açık anahtar üretmemiz gerekiyor.

```
\$ ssh-keygen -t rsa
```

Sorulan sorular için hiç cevap girmeden ENTER'e basarak geçin. Bu bittikten sonra, `\$HOME/.ssh/` dizininiz icinde 2 dosya göreceksiniz. Bu dosyalar `id_¬ rsa.pub` ve `id_rsa` dosyaları olacaktır. `HOME` değişkeninin nerede olduğunu komut satırından Unix/Cygwin'de `echo \$HOME` ile öğrenebilirsiniz. Windows'da dosyaların nereye yazıldığı OpenSSH tarafından zaten `ssh-keygen` sonunda size bildirilecektir.

Biraz önce üretilen dosyalardan `id_rsa`, gizli anahtarınızdır. Dosya `id_¬ rsa.pub` ise açık anahtarınızdır. Şimdi, `id_rsa.pub` kayıdındaki açık anahtarı, uzaktan erişeceğiniz servis bilgisayarına FTP ya da `scp` ile gönderin (`scp` kullanırsanız, -şimdilik- şifre girmeniz gerekecek tabii ki). Sonra, uzaktaki bilgisayardaki kullanacağınız kullanıcı hesabına girin, hesabın üst seviyesinde `\$¬ HOME/.ssh/` dizini altına `id_rsa.pub` kayıdını bırakın. Sonra, servis sisteminde

```
cat > \$HOME/.ssh/id_rsa.pub >> authorized_keys
```

komutunu calıştırın. Kuruluş işlemi bundan ibarettir. Bu son işlemden sonra artık uzaktan işlettiğiniz `ssh` ve `scp` işlemleri şifresiz bir şekilde işinizi yapmanıza izin verecektir.

Açık anahtarımızı servis makinasına eklemek için, Unix `cat` komutu ile `>>` işlecini kullandığımıza dikkat edelim. Bu demektir ki, birden fazla `.pub` dosyasına tek bir `authorized_keys` dosyasına ekleyebiliriz (ve birden fazla kullanıcıyı desteklebilmek için bunu yapmamız gerekir). Böylece aynı makinaya erişen birden fazla erişen kişi, **aynı** servis makinasına ve aynı kullanıcıya değişik açık anahtarlar ile erişebilir.

Eğer `ssh` kullanımınızda problem çıkarsa (probleme göre) şunları yapabilirsiniz:

- Servis tarafındaki `authorized_keys` dosyasının güvenliğinin açık olmaması gerekir. Eğer dosyanın Unix bazında güvenliği çok açıksa, `sshd` bağlanmaya çalışan `ssh` ya da `scp` komutunu bir şifre girmeye zorlayacaktır. `authorized_keys` dosyasının yeterince kapalı hale getirmek için

  ```
  chmod 600 authorized_keys
  ```

  ve bir üst seviyedeki `.ssh` dizini üzerinde

  ```
  chmod 700 .ssh/
  ```

  komutlarını kullanmak gerekebilir.

- SSH bağlanamama durumu var ise, problem sunucu bilgisayarında `sshd` programının işlemesi olabilir. `ps -eaf | grep sshd` ile bunu kontrol edebilirsiniz.

- Bağlanan bilgisayarlarda Cygwin'e özel bir problem (Windows'da) şudur. Cygwin ssh programı (OpenSSH), gizli anahtarın üzerindeki erişim haklarının "kullanıcıya özel" olmasını istiyor. Bu normal tabii çünkü bu anahtar gizli, ve her kullanıcı tarafından okunamaması lâzımdır. Hata şöyle olacaktır:

  ```
  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  @@@@       WARNING: UNPROTECTED PRIVATE KEY FILE!     @
  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

  Permissions 0644 for '/cygdrive/h/.ssh/id_rsa' are too open.

  It is recommended that your private key files are NOT accessible
  by others.

  This private key will be ignored.
  ```

  Hiç problem degil (diye düşünüyoruz) `chmod` kullanırsak iş biter (bölüm 10.2.1). Tek problem, Windows ve Unix'in erişim hakları yöntemlerinin birbiri ile uyuşmamasıdır. `chmod`'u çalıştırdığınızda, komut bir şey yapmış gibi geri gelecek, fakat dosyada bir değişiklik olmayacaktır. Unix dosya haklarının Cygwin'de "simule" edilmesi için, `cygwin.bat` içine (`CYGWIN`¬ `/bin/` altında) şunu eklemek lazımdır.

  ```
  set CYGWIN=tty ntea
  ```

  Bundan sonra

  ```
  chmod 0600 id_rsa
  ```

  komutu düzgün işleyecektir.

6.2 Uygulama İşleyişini Kontrol Etmek

Kurumsal uygulamamızı oluşturan servis programlarımızın uzun süre ayakta kalmasını ve işlem yapabilmesini isteriz. Eğer işletim sistemi hatası, donanım pürüzleri gibi hatalar yüzünden servis süreçlerinde çökme olursa, sektörde yaygın bir metot çökmüş olan servisin otomatik olarak ayağa kaldırılmasıdır. Bu işi yapan en ünlü ticari program olan AutoSys, çöken bir programı tekrar başlatmak, çökme birkaç kez tekrar ederse olursa belli bir eşik değerinden sonra tekrar başlatmayı bırakıp durumu sorumlu bir kimseye e-mail ile bildirmek, ya da zamanlı (scheduled) bir şekilde belli zamanlarda bir programı işletmek gibi önemli görevleri yerine getirebilir.

Kitabımızda bedava / açık yazılım yelpazesinden bulunup kullanılabilen çözümleri tavsiye ettiğimiz için, ticari bir ürün olan AutoSys yerine bizim yazdığımız iki Perl script'ini tavsiye edeceğiz. Bu script'ler AutoSys'in süreç izleme özelliğini taşıyan ufak programlar olacaklardır. AutoSys'in zamanlı çalıştırma özellikleri yerine de, Unix `cron` programını (bölüm 10.5) kullanabilirsiniz.

6.2.1 Blok Eden Servis Script'leri

Blok eden servis programlarından kastımız, başlangıç komutu verilince kendini ön plana alan türden servis programlarıdır. Bu tür programların çökme durumunu kontrol etmek istiyorsak, servisi başlatmak için başlatıcı programı (`run.sh` gibi) elle başlatmak yerine, bir takip edici program üzerinden "başlattırtmamız" gerekmektedir. Bu kontrollü başlatımdan sonra, eğer kontrol edilen program herhangi bir sebeple geri gelirse, kontrol eden program bir çökme durumu olduğunu algılayabilecektir.

Alttaki `jobRunner.pl` adlı Perl script'ini bu amaçla kullanabiliriz.

Liste 6.3: jobRunner.pl

```
#!/usr/bin/perl

use Net::SMTP;

\$argCount=\$#ARGV + 1;

die "Usage: perl t.pl <maxRestartCount> <wait> " .
    "<jobname> <progname>\n" if (\$argCount != 4);

\$maxRestartCount = \$ARGV[0];
\$wait = \$ARGV[1];
\$jobName = \$ARGV[2];
\$jobProg = \$ARGV[3];

print "\$wait";
```

```
# run the program
\$code = system(\$jobProg);

# if first run fails, we come here
\$restartCount = 0;
while (1) {
    \$restartCount++;
    if (\$restartCount == 3) {
        print "Cannot restart any more... Exiting\n";
        alert(\$jobName);
        exit(-1);
    } else {
        print "Exiting with code \$code\n";
        print "Waiting for cleanup \n";
        sleep(\$wait);
        print \$code . "... restarting \n";
        \$code = system(\$jobProg);
    }
}

sub alert(\$){
    my \$jobName = shift;
    my \$from = 'jobRunner@company.com';
    my \$to = 'admin@company.com';
    my \$subject = "Process \$jobName Failed";

    \$smtp = Net::SMTP->new('mail.company.com');

    \$smtp->mail(\$from);
    \$smtp->to(\$to);

    \$smtp->data();
    \$smtp->datasend("To: \$to\n");
    \$smtp->datasend("Subject: \$subject\n");
    \$smtp->datasend("Tried to restart \$jobName \$maxRestartCount times\n");
    \$smtp->datasend("It Failed\n");
    \$smtp->dataend();

    \$smtp->quit;
}
```

`jobRunner.pl` programının seçeneklerini komut satırından `perl jobRunner¬.pl` komutunu işleterek görebiliriz. Bu seçeneklerin açıklaması şöyledir:

- **maxRestartCount**: `jobRunner.pl`'in başlatmış olduğu ve kontrol ettiği program çökünce, en fazla kaç *sefer daha* tekrar ayağa kaldırılması gerektiğini kontrol eder. Eğer bir program `maxRestartCount` kere çökmüş ise, `jobRunner.pl` belirtilen admin kişisine bir e-mail ile durumu bildirecek,

ve programın `maxRestartCount` kere çökmüş olduğunu haber verecektir. Program bir daha başlatılmayacaktır.

- **wait**: Çökme ile tekrar başlatılma arasında kaç saniye beklenmesi gerektiğini belirler.
- **jobname**: Programı başlatan kişinin atadığı bir isim. Herhangi bir kelime olabilir.
- **progname**: İşletilecek programın başlangıç komutu. JBoss örneğinde bu `sh run.sh` olacaktır. Tüm path (full path) burada belirtilir.

`jobRunner.pl` kontrolünde meselâ JBoss başlatmak için tipik bir kullanım şöyle olabilir.

```
perl jobRunner.pl 3 10 JBossServer1 ‘‘sh /usr/local/jboss/bin/run.sh’’
```

Not: Bu şekilde çalıştırdığımız servis programını durdurmak için, programın kendisini değil, onu başlatan ve idare eden `jobRunner.pl` programını öldürmemiz gerektiğini unutmayalım. Başlatıcı programı görmek için Unix üzerinde `ps -¬ eaf | grep perl` ile alacağımız listede, bu Perl script'ini `kill -9` ile durdurabiliriz. Unix süreç kontrolü ve listelemesi için 10.1.2 bölümüne bakabiliriz.

6.2.2 Deamon Programları

Apache gibi "başlangıç programları kendini arka plana (background process) atan" türden programlar için, kontrolü bir *port bazlı* yapabilen idare bir script'ini kullanmamız gerekiyor. Çünkü kendini arka plana atan script'ler, başlatıcı çağırdıktan sonra hemen geri dönerler ve bu da `jobRunner.pl` gibi bir program için aldatıcı olacaktır (programın çökmüş olduğunu zannedecektir). O zaman, kontrol edilen programın çöküp çökmediğini önplandan geri gelip gelmemek değil, belli bir port üzerinde dinleyici olup olmadığını anlayarak kontrol edebiliriz. Sistemde bir port'un kullanılma durumunu `netstat` ile kontrol edebiliriz. `netstat` hakkında ayrıntılı detay için 10.6 bölümüne bakabilirsiniz.

Liste 6.4: deamonRunner.pl

```
#!/usr/bin/perl

use Net::SMTP;

\$argCount=\$#ARGV + 1;

die "Usage: perl t.pl <maxRestartCount> <wait> " .
    "<port> <jobname> <progname>\n" if (\$argCount != 5);

\$maxRestartCount = \$ARGV[0];
\$wait = \$ARGV[1];
```

```
\$port = \$ARGV[2];
\$jobName = \$ARGV[3];
\$jobProg = \$ARGV[4];

# if first run fails, we come here
\$restartCount = 0;
while (1) {
    if (\$restartCount == \$maxRestartCount) {
        print "Cannot restart any more... Exiting\n";
        alert(\$jobName);
        exit(-1);
    } else {
        sleep(\$wait);
        print \$code . "... checking port \$port \n";
        @netstatout = `netstat -ano`;
        foreach \$line(@netstatout) {
            @tokens = split(':|\s',\$line);
            if (\$tokens[7] =~ /\$port/) {
                \$found = 1;
            }

        }
        if (\$found != 1) {
            \$restartCount++;
            \$code = system(\$jobProg);
        }

    }
}

sub alert(\$){
    my \$jobName = shift;
    my \$from = 'deamonRunner@company.com';
    my \$to = `admin@company.com`;
    my \$subject = "Process \$jobName Failed";

    \$smtp = Net::SMTP->new('mail.company.com');

    \$smtp->mail(\$from);
    \$smtp->to(\$to);

    \$smtp->data();
    \$smtp->datasend("To: \$to\n");
    \$smtp->datasend("Subject: \$subject\n");
    \$smtp->datasend("Tried to restart \$jobName \$maxRestartCount times\n");
    \$smtp->datasend("It Failed\n");
    \$smtp->dataend();
```

```
    \$smtp->quit;
}
```

`deamonRunner.pl` seçenekleri `jobRunner.pl` ile neredeyse aynıdır, tek fark, belli aralıklarla kontrol edilmesi gereken bir port parametresidir. Bu parametrenin üzerinde dinleyici olma durumu `netstat -ano` ile belli aralıklarla kontrol eden script, eğer o port dinlenmiyorsa, dinleyen programın çöktüğünü kabul edilecek ve tekrar başlatmak için gerekli komutu uygulayacaktır. Tipik bir `deamonRunner.pl` kullanımı şöyledir:

```
perl jobRunner.pl 3 10 80 Apache1 ‘‘/usr/local/apache/bin/apachectl start’’
```

Bu komuta göre, Apache programının varlığı port 80'e bakılarak her 10 saniyede bir kontrol edilecek, eğer program çökmüş ise tekrar başlatılacaktır. Bu tekrar başlatma işlemi en fazla 3 kere tekrarlanacak, 3 kereden sonra tekrar başlatmak denenmeyecek, durum görevli admin kişisine bir e-mail ile bildirilecektir.

6.3 Uygulamadan İstatistik Almak

Sürecimizin işleyiş durumunu kontrol eden yukarıdaki script'ler sayesinde, eğer izlenen bir süreç çökmüş ise, onu tekrar başlatarak sistemin devamını sağlamak mümkün olacaktır. İşleyiş hakkında daha detaylı bilgiler istersek, bu bilgileri uygulama içinden paylaşmamız, bir şekilde dış dünyaya afişe etmemiz ve belli aralıklarla bu bilgileri güncellememiz gerekmektedir. İşleyiş bilgilerini dış dünyaya afişe edersek, bu bilgileri (tercihen) görsel bir idare programı sayesinde izleyebilir, bir ekranda gösterebilir, hattâ ilginç olan şartlar üzerinde alarm şartları tanımlamak suretiyle yetkili bir admin kişisinin (şartlar ihlâl edildiğinde) e-mail ile bilgilendirilmesini sağlayabiliriz.

J2EE dünyasında bir uygulamanın iç istatistiklerini paylaşmak ve idare bilgilerini dış dünyaya göstermek için JMX teknolojisi kullanılır. JMX, **J**ava **M**anagement E**X**tensions (Java İdare Uzantıları) cümlesinin kısaltılmışıdır. Bu teknoloji sayesinde bir uygulama, belli bir standarta uyan bir MBean class'ını (bir bean) dış dünyaya gösterebilir. Aslında JMX ile MBean afişe etmek, dağıtık nesne teknolojisi ile nesnemizi JVM dışından erişime açmaya çok benzer, ama iki fark (üstünlük) vardır: Bir: JMX ile dışarıdan bağlanan müşteri, MBean'lerinizin metotlarını dinamik şekilde gezebilir. İki: JMX teknolojisi, idare amaçlı teknolojiler arasında piyasada tutmuş bir teknolojidir, bu yüzden birçok ticari ve açık yazılım ürün istatistiklerini JMX üzerinde paylaşmak amacıyla hazır MBean'ler yazıp ürünlerine dahil etmişlerdir. Böylece uygulamamız, bu hazır MBean'leri oldukları gibi alıp, kendi MBean'lerimiz ile beraber yanyana afişe edebilir. Hibernate `org.hibernate.jmx.StatisticsService` MBean'i bu şekilde hazır yazılmış bir istatistik nesnesidir.

6.3.1 JMX ile MBean Yazmak

Dış dünyaya bilgi vermek için, Kendi MBean'imizi yazmayı öğrenmemiz gerekiyor. İki türlü MBean stili vardır, statik MBean ve dinamik MBean. Statik MBean ile uygulama sırasında *ismi* değişmeyecek parametreleri sunabiliriz. Meselâ uygulamamızın önbellek büyüklüğü, veri tabanı bağlantı havuzundaki aktif bağlantılar, ya da sistemdeki aktif kullanıcı sayısı gibi parametrelerin ne olduğu önceden bilinen parametrelerdir, ve uygulama sırasında, ya da makinadan makinaya değişmezler.

Dinamik MBean'ler ise, işleyiş anında ya da makinadan makinaya değişebilecek parametreleri afişe etmek için kullanılır. Meselâ uygulamamızın üzerinde koştuğu makinanın disklerinin kullanım ölçüleri (IO read/write) gibi bir istatistik dinamik MBean gerektirir, çünkü uygulamamız kurulduktan ve kullanıma başladıktan sonra bir makinaya ek disk, ek işlemci gibi donanım uzantıları yapmak mümkündür; Eğer istatistik paylaşma yöntemimiz sabit disk isimleri üzerinden bilgi topluyor (ve JMX ile paylaşıyor) olsaydı, yeni disk'ler eklendikten sonra bu ek birimler dış dünya tarafından görülemez olacaktı.

Statik MBean

Statik MBean tanımlamak için sonu `MBean` ile biten bir interface, ve bu interface'i gerçekleştiren bir class gerekecektir. Örnek olarak, bir Web uygulamasına yapılan toplam ziyaretleri gösteren bir MBean yazalım: Esas değerleri taşıyan sayaç, `VisitCounter` adlı bir Singleton class'ı olsun.

```
public class VisitCounter {

    private static VisitCounter instance = null;

    private VisitCounter() { }

    public static VisitCounter instance() {
        if (instance == null) {
            instance = new VisitCounter();
        }
        return instance;
    }

    public int visitCount = 0;

    public synchronized void incrementCount() {
        visitCount++;
    }

    public int getVisitCount() { return visitCount; }
}
```

Sayaç nesnesi üzerinde arttırma işlemini çağıran bir Web filtresi olacaktır. Ayrıca Java Servlet belirtimine (specification) göre bir Filtre, Web isteği başlamadan önce ve sonra işletilebilen bir kod parçasıdır. Biz filtremizi her `*.do` çağrısı için aktif olacak şekilde ayarlayalım.

Liste 6.5: VisitCounterFilter.java

```
public class VisitCounterFilter implements Filter {

    public void init(FilterConfig filterConfig) throws ServletException {
    }

    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain) throws IOException,
                                                   ServletException
    {
        chain.doFilter(request, response);
        VisitCounter.instance().incrementCount();
    }

    public void destroy() { }
}
```

Liste 6.6: web.xml

```
  <filter>
    <filter-name>VisitCounterFilter</filter-name>
    <filter-class>
      org.mycompany.kitapdemo.util.VisitCounterFilter
    </filter-class>
  </filter>

  <filter-mapping>
    <filter-name>VisitCounterFilter</filter-name>
    <url-pattern>*.do</url-pattern>
  </filter-mapping>
```

Sayaç değerlerini dış dünyaya göstermek için, JMX üzerinden kullanabileceğimiz bir statik MBean yazalım.

```
public interface UserStatMBean {
    public int getTotalRequestCount();
}

public class UserStat implements UserStatMBean {

    VisitCounter counter = null;
```

```
    public UserStat() {
        counter = VisitCounter.instance();
    }

    public int getTotalRequestCount() {
        return counter.getVisitCount();
    }
}
```

Bu MBean'i bir MBeanServer`MBeanServer` ile kayıt ettirdiğimiz zaman, dış dünya `totalRequestCount` adlı istatistiği görebilecektir. MBean kayıt ettirmek için, projenin başlangıç kodları işlettiği yerde (Web uygulamaları için `AppStartup` class'ı) şu şekilde kodlar işletmemiz gerekir.

```
public class AppStartup implements AppStartupMBean {
    public void start() throws Exception {
        ...
        MBeanServer mbeanServer =
            (MBeanServer) MBeanServerFactory.findMBeanServer(null).get(0);
        LocateRegistry.createRegistry(new Integer(jmxPort).intValue());

        UserStat userStat = new UserStat();
        ObjectName userStatName = new ObjectName("UserStat:type=UserStat");
        mbeanServer.registerMBean(userStat, userStatName);
        ...
    }
}
```

Çağrı `findMBeanServer` ile JBoss'un `MBeanServer`'ına erişmiş oluyoruz. Bu server, MBean'lerin kayıt edildiği ve merkezi bir giriş noktası sağlayan noktadır. Eğer yeni bir `MBeanServer` yaratmak istesek,

```
MBeanServer mbeanServer = MBeanServerFactory.createMBeanServer();
```

kullanırdık. Metot `findMBeanServer` kullanmakla, JBoss'un içinde olan tüm diğer MBean'lerin de kayıt ediliği merkezi noktayı almış oluyoruz.

JBoss'un kontrol ettiği MBean'leri görmek için, tarayıcınızdan `/jmx-console` adresine gidebilirsiniz. Şekil 6.1 üzerinde görüldüğü gibi `UserStat` adlı yeni MBean'imiz diğer MBean'ler yanında servis'e eklenmiştir. MBean bağlantısının üzerine tıklayarak içindeki öğe (attribute) değerlerini görebiliriz.

Dinamik MBean

Öğeleri değişken yapıda olan MBean yazmak için, statik MBean teknolojisini kullanamayız. Burada dinamik MBean'ler imdadımıza yetişiyor. Dinamik MBean yazmak için genellikle kullanılan düzen (pattern), değişken öğeleri, eklenip çıkarmasına izin verdiği ve anahtar bazlı veri tutabildiği için bir `HashMap` üzerinde tutmaktır. İşlem anında öğe olması istenen değerler, dinamik bir şekilde `HashMap`'e anahtar olarak verilir.

Bu `HashMap`'i JMX üzerinden dış dünyaya göstermek için, MBean içinde anında bir dönüşüm yapmamız gerekmektedir. Bu dönüşüm, `HashMap` anahtar (key) değerini MBean öğe ismine, `HashMap` değerini ise MBean öğe değeri hâline getirmelidir. Bu dönüşüm örneğini `StrutsHibPerformance` projesi altındaki `org.mycompany.kitapdemo.service.OsStatistics` MBean class'ı üzerinde görüyoruz.

Bu kodda, `HashMap` üzerinden dinamik MBean yaratmak için kullanılan kod parçası ("dinamik kısım" comment'leri altında listelenen bölüm) oldukça basmakalıp kodlardır. Bu kodları, dinamiklik eklemek istediğimiz diğer bir MBean'e olduğu gibi ekleyebilir, ya da tekrar kullanımı (reusability) arttırmak amacıyla, dinamiklik kodlarını bir üst class içinde tutup MBean'imizden miras alabiliriz.

`OsStatistics` class'ının ihtiyacı olan değerler, işletim sisteminde gelmektedir. Bu değerler mikroişlemci kullanım yüzdeleri, ne kadar boş bellek olduğu gibi işletim sisteminin takip ettiği bilgilerdir, ve komut satırında bunları görmek için `vmstat` komutunu kullanıyoruz. `OsStatistics`, `vmstat`'in çıktısına erişmek için `Runtime.getRuntime().exec()` metotu ile `vmstat`'i işletmekte, ve geri gelen metin bazlı çıktıyı tarayarak içinden ilgilendiği değerleri çekip çıkarmaktadır. `vmstat` hakkında detayları 5.3.2 bölümünde bulabilirsiniz.

Dinamik MBean'imiz JBoss'a gönderildikten ve ilk değerlerini topladıktan

Şekil 6.1: UserStat MBean'i JmxConsole'da

sonra, `/jmx-console` ekranından Şekil 6.2 üzerindeki gibi gözükecektir.

List of MBean attributes:

Name	Type	Access	Value
mem_bi	java.lang.String	RW	299
mem_id	java.lang.String	RW	81
mem_b	java.lang.String	RW	0
mem_so	java.lang.String	RW	0
mem_in	java.lang.String	RW	425
mem_r	java.lang.String	RW	0
mem_wa	java.lang.String	RW	0
mem_buff	java.lang.String	RW	25728
mem_swpd	java.lang.String	RW	0
mem_cache	java.lang.String	RW	145516

Şekil 6.2: Dinamik MBean Ekranı

Görüldüğü gibi, `mem_cache`, `mem_swpd` gibi değerler, direk `vmstat` çıktısından gelmektedir. `mem_swpd`, `swpd` öğesinin başına `mem_` eklenmek suretiyle yaratılmıştır. Öğelerin başına `mem` eklenmesinin sebebi, ileride `OsStatistics` MBean'ine `iostat` gibi bir Unix komutunun çıktısını da eklersek, parametrelerin birbirine karışmaması içindir.

`OsStatistics` kodlarında belli aralıklarla istatistik değerlerinin güncellenmesi için `OsStatistics` içinde olan bir Thread mevcuttur. Bu Thread'i `OsStatictics` nesnesini kayıt ettiğimizde başlatmamız gerekir. Bunun için uygulama başlangıç kodları içinde (`StrutsHibPerformance` projesi için `AppStartup`) `osStatistics.startThread()` metotunu çağırmamız yeterlidir. Başlangıç kodlarının tamamı şöyle gözükür.

```
mbeanServer = ...
OsStatistics stats = new OsStatistics();
stats.startThread();
ObjectName name = new ObjectName("OsStatistics:type=OsStatistics");
mbeanServer.registerMBean(stats, name);
```

İstatistik değerlerinde, alınma stili olarak `UserStat` ile `OsStatistics` arasında bir fark vardır. `UserStat`, kendine yapılan tüm `get` çağrılarını başka bir merkezi nesneye aktararak (delegate) gerçek değerlerin "o başka yerde" güncellenmesini bekliyordu. `OsStatistics` ise, kendi değerlerini kendi toplamakta, kendi işlemekte ve kendi muhafaza etmektedir. Bu fark, iki değişik MBean güncelleme yöntemini sahneye çıkartması için böyle hazırlanmıştır, ve MBean'lerin dinamikliği

ya da statikliği ile bir alâkası yoktur. Statik bir MBean üzerinde de değerleri toplayan, işleyen ve muhafaza eden kodlar yazabilirsiniz.

Hazır yazılmış açık yazılım ya da ticari paketlerin MBean'lerini paylaşmak için o paketlerin belgelerine bakınız. Meselâ Hibernate belgelerinde Hibernate istatistik MBean'ini paylaşmak için şunların yapılması söylenmiştir.

```
mbeanServer = ....
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "kitapdemo");
ObjectName on = new ObjectName("hibernate", tb);
StatisticsService hibstats = new StatisticsService();
hibstats.setSessionFactory(HibernateSession.getSessionFactory());
mbeanServer.registerMBean(hibstats, on);
```

Tüm bu paylasımları yapan örnek `AppStartup` kodlarını, `StrutsHibPerformance` projesinde bulabilirsiniz.

6.3.2 JmxMonitor

MBean değerlerini `jmx-console` üzerinden göstermek bir yararlı bir gözlem yöntemidir. Fakat gerçek dünyada işlemekte olan bir sistemde, bize sunulan istatistiklere sadece çıplak gözle bakmaktan daha fazla yeteneklere ihtiyacımız vardır. Düşünelim ki, `OsStatistics`'den gelen makinamının kullanılmayan bellek kapasitesi (`mem_free`) belli bir değer altına düşerse, haberimizin olmasını istiyoruz. Ya da, statik MBean örneği olarak yazdığımız `UserStat` nesnesindeki `totalRequestCount`, belli bir değeri geçtiğinde bir alarm durumu olmalı ve admin e-mail ile haberdar edilmelidir. Bu tür izleme işlemlerini nasıl yapacağız?

JmxMonitor[1] açık yazılım projesi, bu tür ihtiyaçları karşılamak için yazılmıştır. Eğer bir uygulamada afişe edilmiş MBean'ler var ise, bu MBean'ler üzerindeki her öğe için, JmxMonitor'de bu öğeyi gözleyen, ve kullanıcının tanımladığı bir ve ya daha fazla eşik değerini (threshold) ya da eşitlik şartlarına uyan durumları üzerinde bir alarm şartı tanımlayabiliriz. Alarm şartları ortaya çıkarsa JmxMonitor e-mail ve HTML önyüzden durumu admin'e bildirilecektir (JmxMonitor kullanıcısının admin görevlisi olduğunu farz ediyoruz).

JmxMonitor kodları içine hiçbir MBean isminin gömülmesi (hardcode) gerekmez. JmxMonitor, JMX teknolojisinin Java Reflection benzeri dinamik keşif özelliğini kullanarak, önce bir `MBeanServer` üzerindeki tüm MBean'leri alır daha sonra bu MBean'lerin tüm ögelerini dinamik olarak keşfeder. MBean ve öğe listesi kullanıcıya sunulur ve kullanıcı, öğelerin arasından ilgilendiklerini seçerek üzerlerinde eşik ya da eşitlik şartları tanımlar.

Bu potansiyel alarm şartları ana ekran üzerinde listelenir, ve ileri bir zamanda, eğer uyan bir eşitlik ya da geçilen bir eşik şartı olursa, ana ekranda bu alarm yanıp sönmeye başlayacaktır. Ayrıca admin'e e-mail ile durum bildirilir.

[1] http://jmxmonitor.sourceforge.net

JmxMonitor Web bazlı bir uygulamadır. Web uygulamasının içinden (`JmxMonitorStartup`) başlatılan ve alarm değerlerini kontrol eden bir Thread, belli aralıklarla sürekli kontrol yapıyor olacaktıor . Ayrıca JmxMonitor Adaptor düzenini (pattern) kullanarak, istatistik toplamak için kullandığı bağlantı teknolojisini değiştirmesi de mümkündür. Meselâ paket içinden çıkan adaptör `JmxFacadeJdk15` class'ıdır, fakat JmxMonitor'u uzatmak isteyen bir programcı, soket üzerinden bilgi toplayan bir adaptör yazabilir. Bu socket adaptörü, servis tarafından yine socket üzerinden bilgi afişe eden bir makina, port ikilisine bağlanacak, ve beklediği format'taki değerleri alarak admin'e eşik ve eşitlik seçimi için sunacaktır. Kendi yazdığınız (custom) adaptörün `JmxFacade` interface'ini gerçekleştirmesi gerekmektedir. Buna göre her adaptörün;

1. Verilen makina ve port ikilisine bağlanması mümkün olmalıdır (`connect`)
2. İstatistik sunan servisteki tüm "ana birimler" ve bu birimler "alt birimlerinin" listesi alınabilir olmalıdır (`retrieveAllObjectNames`)
3. Her ölçümün "gerçek değerini" almak mümkün olmalıdır `getLatestValue`)

Altta bir adaptörün sahip olması gereken metotlar gözükmektedir. Her adaptöre uymayan (üstteki metotlar haricinde) metotlardan bazıları, şartlara göre boş bırakılabilir.

```
public interface JmxFacade {
    public Machine getMachine();
    public void connect(Machine machine) ..
    public Map retrieveAllObjectNames() ..
    public String getLatestValue(ObjectName name, String attribute) ..
    public void refreshValues() ..
}
```

Servis Tarafı

JmxMonitor'un `JmxFacadeAdaptor15` adaptörü ile bir `MBeanServer`'a erişebilmesi için, o servisin JMX MBean'lerini RMI üzerinden sunması gerekir. MX4J paketi bu tür bir RMI köprüsünü sunmaktadır. Kullanmak için `AppStartup` içinde bulduğumuz ya da yenisini yarattığımız `MBeanServer` üzerinde bu köprünün nasıl kullanılacağını belirtmemiz gerekecektir.

```
public class AppStartup implements AppStartupMBean {
    public void start() throws Exception {
        ...
        mbeanServer = ... // MBeanServer'ı al ya da yarat
        ...
        final String localRmiRegistryHost = "localhost";
        final String localRmiRegistryPort = "42004";
```

```
final String jmxHost = "localhost";
final String jmxPort = "44334";
final String jmxUrl = "/kitapdemo";

JMXServiceURL url =
    new JMXServiceURL("service:jmx:rmi://"
                      +
                      localRmiRegistryHost
                      +
                      ":"
                      +
                      localRmiRegistryPort
                      +
                      "/jndi/rmi://"
                      +
                      jmxHost
                      +
                      ":"
                      +
                      jmxPort
                      +
                      jmxUrl);

(JMXConnectorServerFactory.
 newJMXConnectorServer(url,null,mbeanServer)).start();
...
...
```

Bu başlatım komutu ile bizim atadığımız bir RMI URL'i olan bir JMX servisi yaratmış oluyoruz. Uzaktan bağlanmak isteyenler JMX RMI çağrısını şöyle bir URL'i oluşturarak gerçekleştireceklerdir:

```
service:jmx:rmi://localhost:42004/jndi/rmi://localhost:44334/kitapdemo
```

Bu URL'in işleyip işlemediğini anlamak için en çabuk test şudur: JDK 1.5 içinde paketten çıkan JConsole adında bir uygulama mevcuttur. Bu uygulama, aynı JmxMonitor gibi RMI üzerinden dinamik olarak MBean'leri keşfeder ve gösterir (fakat JmxMonitor gibi alarm şartları koymanıza izin vermez). Bu uygulamayı `JDK_1_5/bin/jconsole` komutunu kullanarak başlatabiliriz. İlk çıkan bağlantı diyalog kutusunda, `Advanced` tab'ine giderek, üstte görülen URL'i (tabii servisimizi de başlattıktan sonra) gireriz. Şekil 6.3 üzerindeki ekran görülecektir.

Dikkat edersek, uygulamamızdan afişe ettiğimiz `UserStat`, `OsStatistics` ve Hibernate istatikleri dinamik olarak keşfedilmiş ve listelenmiştir.

JmxMonitor Kurmak

JmxMonitor, altyapı olarak kitabımızda sunulan `StrutsHibxx` projeleri ile aynı teknik mimariye sahiptir. Bu sebeple indirmesi, kurulması için yapılması gerekenler, A.1.2 bölümündekiler ile aynıdır. Ek olarak, başlangıç verilerini yüklemek için `JMXMONITOR/src/sql/init_data.sql` script'ini MySQL (ya da diğer) veri tabanınız üzerinde işletmeniz gereklidir.

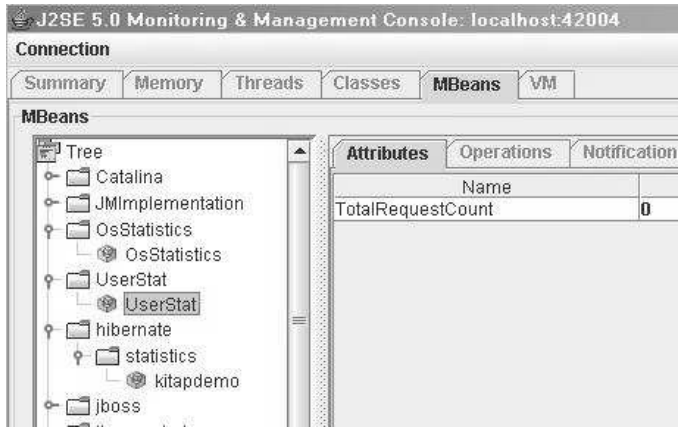
Bundan sonra, JBoss servisinizi başlatın ve `localhost:8080/jmxmonitor` adresini ziyaret edin. JmxMonitor ile bir `MBeanServer`'a bağlanmak için, o servisin makina ismi ve port numarasına girmemiz gerekiyor. Bu alanlar için örnek bazı değerler Şekil 6.4 üzerindeki gibi gözükecektir.

Alanlar için girilmesi gereken değerleri şöyle bulmamız gerekir: Bir `MBeanServer`'a bir RMI köprüsü uyguladığımızda (daha önce `AppStartup` içinde örneğini gösterdiğimiz gibi), şu parametreleri kullanmıştık

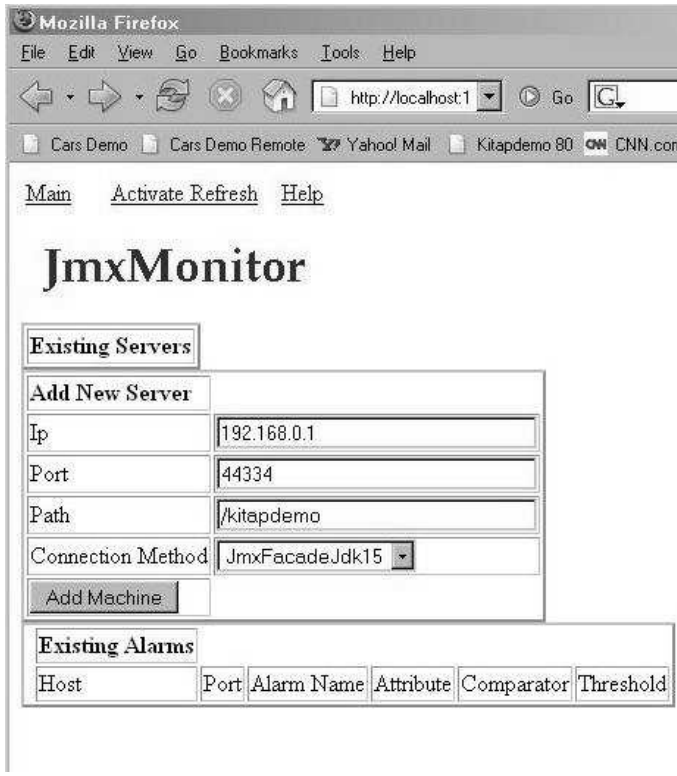
```
final String localRmiRegistryHost = "localhost";
final String localRmiRegistryPort = "42004";
final String jmxHost = "localhost";
final String jmxPort = "44334";
final String jmxUrl = "/kitapdemo";
```

Bu parametrelerden `jmxUrl`, JmxMonitor'ün URL alanına, `jmxPort`, port alanına, `jmxHost`'un ise host alanına girilmesi gerekmektedir. Bu değerleri girip, `Add Machine` düğmesine tıklarsak, JmxMonitor bu servisi listesine ekleyecektir (Şekil 6.5).

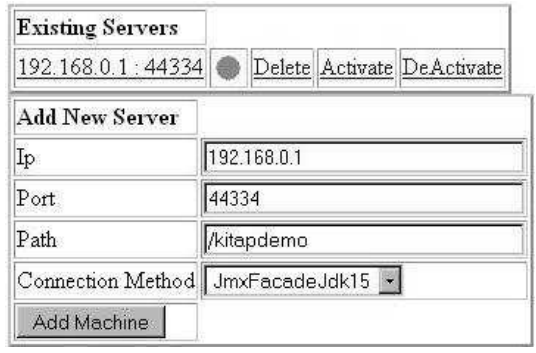
Listelenen bir servis üzerine tıklarsak, JmxMonitor servise RMI üzerinden bağlanmaya çalışacak, ve bir sonraki ekranda (Şekil 6.6) bu servisin tüm MBean'lerinin listesini bize sunacaktır.



Şekil 6.3: JConsole



Şekil 6.4: Giriş Ekranı



Şekil 6.5: Servis Eklendi

Aynı JConsole örneğinde olduğu gibi, kendi yazmış olduğumuz MBean'leri de bu ekranda görmemiz mümkündür. Bu MBean'lerden biri için bir alarm şartı tanımlamak için, meselâ `OsStatistics` MBean'inindeki bir öğe için, ilk önce öğelerin listesini görmemiz gerekiyor. Öğe listesini almak için `OsStatistics` bağlantısını tıklamamız lâzım.

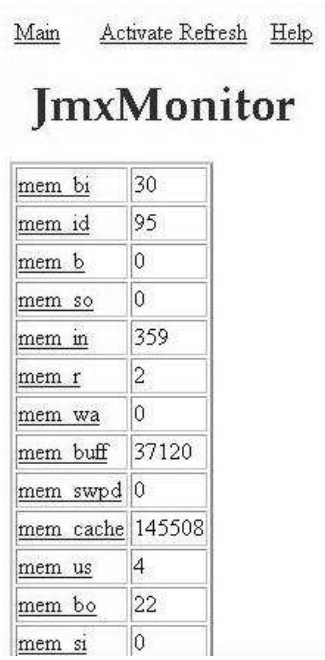
Şekil 6.7 üzerinde tıklamadan sonra tüm öğeleri görüyoruz. Bu öğelerden biri üstünde bir alarm şartı yaratmak istersek, öğenin üzerindke tıklamamız gerekecek. Örnek olarak `OsStatistics` altındaki `mem_free` öğesi üzerinde alarm yaratalım, tıklamadan sonra, Şekil 6.8 üzerindeki ekranı görürüz.

Bu ekranda alarmın `mem_free` gerçek değeri 640000 altında düşerse, bir alarm oluşmasını istiyoruz. O zaman `Comparator` yâni karşılaştırıcı olarak `<` işaretini seçeriz, ve değer olarak 640000 değerini `Threshold` yâni eşik değeri kutusuna gireriz. Alarm eklendikten sonra Şekil 6.9 üzerinde görulen ekrana geliriz. Eklenmiş alarmı silmek için, alarmın yanındaki `Delete` bağlantısına tıklayabilirsiniz.

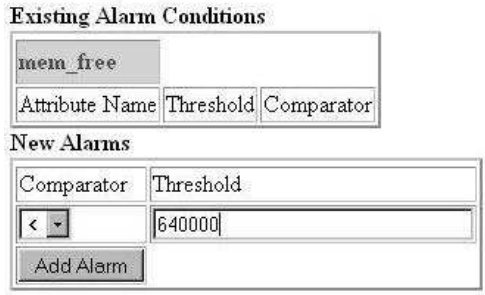
Bu şekilde devam ederek birden fazla alarm eklememiz mümkündür. Alarm eklememiz bitince, ana sayfaya dönebiliriz. Ana sayfaya dönünce eklediğimiz alarmların sayfanın altında listelendiğini göreceğiz (Şekil 6.10).

Artık yapmamız gereken tek işlem, izlemekte olduğumuz makina ve port ikilisi üzerinde yapılacak izleme/gözetlemeyi işlemini aktif hâle getirmektir. Böylece JmxMonitor, üzerinde alarm olan öğelerin en son değerlerini periyodik bir şekilde alarak, eşik veya eşitlik şartlarına uyup uyulmadığını kontrol edebilecektir. Gözetlemeyi aktif hâle getirmek için, makina ve port ikilisi

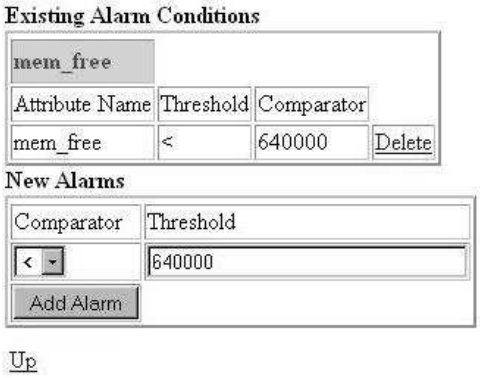
Şekil 6.6: MBean Listesi



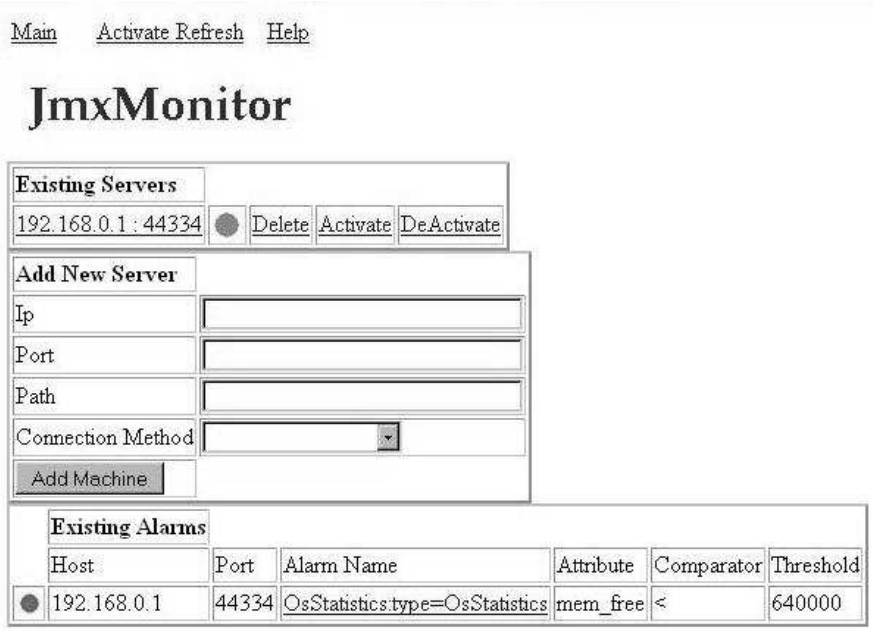
Şekil 6.7: MBean İçindeki Öğeler



Şekil 6.8: Alarm Ekleme Ekranı



Şekil 6.9: Alarm Eklendi



Şekil 6.10: Ana Sayfaya Dönüş - Alarm Gösteriliyor

yanındaki `Active` seçeneğine tıklarız. Aktif hâle getirdiğimiz birimin işareti, yeşile dönecektir (Şekil 6.11); Bu renk, o makina ve port'un izlenmeye başladığına bir işarettir. Eğer izlemeyi durdurmak istiyorsak, `DeActivate` seçeneğini kullanabiliriz.

Şekil 6.11: Aktif Olan bir Makina/Port İkilisi

Bölüm 7

Test Etmek

Bu Bölümdekiler

- JUnit ile birim testleri
- JMeter ile kabul testleri
- jMock

Programlama sırasında, üzerinde çalıştığımız kodu arada sırada işletebilmek için kodu derleme işleminden geçirmemiz gerekir. Derleme işlemini görevlerinden biri işler kod üretmektir, ve çok önemli diğer bir görevi ise tüm kodlar üzerinde sözdizim kontrolü yapmaktır. Meselâ güçlü tipleme (strong typing) bekleyen bir dil kullanıyorsak, bir eşitliğin (=) iki tarafındaki iki değişkenin tiplerinin birbiri ile aynı olması gerekecektir. Bu, bize dil tarafından verilen bir kontroldür, ve yapabileceğimiz potansiyel bir hata için hatırlatıcı nitelik taşır.

Dil tarafından getirilmiş bu şekildeki kurallar, programımızın doğruluğu için getirilmiş tecrübeye dayalı kuralların toplamıdır. Bu kurallar nasıl konulmuştur? Eğer istatistiki olarak bir tipteki değişkenin ötekine eşitlenmesi programcı tarafından çok yapılan bir hata ise, bu hatayı öngören bir kural, programımızı eşitleme hatalarından kurtarmış olur. Alternatif olarak zayıf tipli (weakly typed) diller her tipi, her diğer tipe eşitlemeye izin verip, programcıya sormadan tip değişimini işleyiş anında yaparlar. Bu dillerin kullanım beklentileri ona göredir.

Derleme aşaması sözdizim hatalarını kontrol eder; Fakat bu kontrollerden geçen kodumuz, tüm hatalar arınmış mıdır? Bu soruya cevap doğal olarak "Hayır" olacaktır. Programcılık ile uğraşan herkesin bildiği gibi, sözdizimsel kurallar potansiyel hataları bulmakta yardımcı olsalar da, programın içinde mantık hataları (bug) olması hâla muhtemeldir. Hâtta, yine istatiksel olarak diyebiliriz ki, çetrefillik seviyesi orta ve üstü seviyede olan hiçbir program ilk düzgün derlenmesinden sonra beklendiği gibi çalışmayacaktır. Unutkan olan insanlar (Kural #4), muhakkak programı yazarken mantık hatası yaparlar.

O zaman programımızı yazarken, aynen sözdizimsel hataları derleyiciye yakalattırdığımız gibi, mantık hatalarını da yakalayabilecek olan kendi kontrol edici kurallarımızı kendi üzerimizde yaratmamız gereklidir. Fakat bu kontrollerin tip kontrolleri gibi sözdizimsel seviyede değil, programın işleyişi seviyesinde olması gerekecektir.

Program işleyişini kontrol etmek istiyorsak, programımızı *işletmemiz* ve sonucunu kontrol etmemiz gerekecektir. Bu işleyişin önemli faktörlerinden biri *otomatik* yapılabilmesi olmalıdır çünkü otomatik yapılabilen kontroller, birden fazla ve sıra hâlinde kodun üzerinde işletilebilirler.

Bu tür otomatik işleyiş kontrollerini iki seviyede gerçekleştirmemiz mümkündür: Birincisi en ufak kod birimi (modül) bazında, ikincisi ise programın geneli, yâni programın dış kullanılış bazında olacaktır. Bu iki kontrolü yöntemini nasıl kuracağımızı ve kullanabileceğimizi alttaki bölümlerde göreceğiz.

7.1 Birim Testleri

Birim testleri bir fonksiyon ya da metot seviyesinde yapılır. Birim testleri isimlerini, en ufak işler kod birimi olan "fonksiyonu" test ediyor olmalarından alırlar.

Birim testinin kullanımı oldukça basittir: Normal program işleyişi sırasında bir bir metot ya da fonksiyonu test etmek istiyorsak, o metotu içeren nesneyi

`new` ile yaratırız, ve sadece test etmek istediğimiz metotu çağırırız. Meselâ elimizde iki sayıyı toplayan `add` adlı bir metot olsun.

```
public class AdderService
{
   public int add(int x, int y) {
      return x+y;
   }
}
```

Bu class'ı test etmek için, bir `main` işlevden bu class'ı şöyle kullanırız (aslında tavsiye ettiğimiz test çağırıcı nokta `main` değildir, fakat şimdilik -kurması çok basit olması sebebiyle- örneğe bu şekilde başlamak istedik)

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

   AdderService service = new AdderService();
   if (service.add(2, 2) != 4) {
      System.out.println(``test failed'');
      System.exit(-1);
   }
}
```

Bu testi komut satırından çağırdığımızda, eğer fonksiyon doğru yazılmışsa hiçbir hata mesajı görmememiz gerekir. Birim testlerinin amacı budur: Belli değerleri işlem mantığına girdi olarak verince, beklediğimiz çıktı değerlerinin verilip verilmediğini işlem anında (runtime) kontrol etmek, verilmediyse hata raporu verebilmek. Üstte verilen fonksiyon ve test çok basit örneklerdir, fakat orta ve daha üstü çetrefillikte olan bir fonksiyonu da aynı yöntemle test edebiliriz, ve bu gibi testlerden birçoğunu otomatik olarak işlem kodu üzerinde arka arkaya işletebiliriz.

7.1.1 JUnit - Birim Test İşletici

Eğer birden fazla birim testi çağırabilmek ve daha iyi karşılaştırıcı foknsiyonlar kullanmak istiyorsak, JUnit[1] projesinin test altyapısını kullanabiliriz. JUnit,

- Birim testlerinizi `main` yerine, test edici özel class'lar içinden çağırmamızı sağlar.
- Özel test edici class'da ismi "`test`" ile başlayan her metotu, JUnit otomatik olarak çağırabilir (böylece `main` içinden (elle) her çağrıyı eklememiz gerekmez)
- `assertEquals`, `assertTrue` gibi karşılaştırıcı fonksiyonlar sağlayarak, basmakalıp `if` çağrılarından bizi kurtarır

[1] http://www.junit.org

Bu altyapının üstüne, tarafımızdan, "sonu `Test.class` ile biten" her class'ın da birim test class'i kabul edilip, işletilmesini sağlayan `AllTest` adlı bir global işletici yazılmıştır. Bu global işletici, sonu `Test.class` ile biten tüm dosyaları `CLASSPATH`'ten toplayarak JUnit'i çağırabilir ve birim testlerin işletilmesini ve sonuçların toplanmasını sağlayabilir. `AllTest` yardımcı kodu Ant `build.xml` içinden çağırılabilir. Örnek kitap kodlarımızdaki her proje içindeki `build.xml` içinde bu tür bir `test` target'i bulabilirsiniz. Kullanım şöyledir:

```
...
<target name="test" depends="clean,compile">
  <java fork="yes" classname="org.mycompany.kitapdemo.util.AllTest"
        taskname="junit" failonerror="true">
    <arg value="Test.class"/>
    <classpath refid="compile.classpath"/>
  </java>
</target>
...
```

Bu `build.xml` bloğuna göre, önce `clean` ve `compile` target'leri işletilerek eski işler kodlar silinip kaynak kodlar derlenecek, sonra `AllTest` class'ı, `Test.class` parametresi ile çağırılacaktır.

`Test.class` yerine değişik bir sonekle biten birim test class'larını çağırmak istiyorsaniz, `<arg value ..>` için değişik bir değer verebilirsiniz. Şimdi örnek bir JUnit birim testini görebiliriz.

```
import junit.framework.TestCase;

public class AdderServiceTest extends TestCase {

    public void testAdd() {
        AdderService service = new AdderService();
        assertEquals(5, service.add(3, 2));
    }
}
```

Testleri işletmek için komut satırında `ant test` yazmanız yeterlidir. Bunun sonuc olarak şöyle bir çıktı göreceksiniz:

```
...
...

test:
Adding org.mycompany.kitapdemo.sample.AdderServiceTest
Adding org.mycompany.kitapdemo.util.AllTest\$Test

...
Time: 3.835

OK (3 tests)
```

Testlerimiz işlemiş, kodların doğru işlediğini bulmuş ve sonucu bildirmiştir. Eğer kodlarda bir yanlışlık olsaydı (meselâ `+` işareti yerine yanlışlıkla bir `-` işareti konmuş olsa),

```
public class AdderService {

    public int add(int x, int y) {
        return x +- y; // yanlış kod!
    }
}
```

o zaman test şöyle bir hata yakalayacaktı:

```
..
..
There was 1 failure:
1) testAdd(org.mycompany.kitapdemo.sample.AdderServiceTest)junit.fram
       ework.AssertionFailedError: expected:<5> but was:<1>
  at org.mycompany.kitapdemo.sample.AdderServiceTest.testAdd(Ad
  derServiceTest.java:30)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAcc
  essorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingM
  ethodAccessorImpl.java:25)
  at org.mycompany.kitapdemo.util.AllTest.main(AllTest.java:86)

FAILURES!!!
Tests run: 4, Failures: 1, Errors: 0
...
```

Sonuca bakarsak, `expected:<5> but was:<1>` mesajının verildiğini görüyoruz, yâni JUnit bize birim testimizin sonuç olarak 5 beklediğini, ama test edilen birimden 1 değerinin geldiğini söylemektedir. Bu hatalı bir durumdur, beklenen ve gelen değeri karşılaştıran JUnit, bir uyumsuzluk olduğunu görünce, hatalı bir kod olduğunu anlamıştır, ve çok detaylı bir tarif vermiştir. Bu detaylı târif `assertEquals` kullanımı sayesinde gelmiştir. Hatanın sebebi ise, yanlışlıkla koda koyulmuş `+-` işleminin toplama değil, çıkartma yapmasıdır!

7.1.2 Kurumsal Kodları Birim Testinden Geçirmek

`AdderService.add()` islemi, dışarıdan oldukça izole bir işlem kodudur. Bu metot üzerinde test gerçekleştirmek bu sebeble oldukça kolaydı. Fakat bir kurumsal uygulamalanın işlemesi için yazılan kodların, genellikle bir *birleştirici* (integrator) özelliği vardır; Sektörümüzde, işi sadece kurumsal uygulama yazmak olan şirketlere bu sebeple "sistem birleştiricisi (system integrator)" ismi verilir. Kurumsal bir uygulama, çok nadiren bir ada olarak tek başına iş yapmaktadır. Ya bir veri tabanına bağlanırız, ya bir e-mail server üzerinde e-mail

atarız, ya da eski (legacy) bir sisteme bağlanarak bilgi alışverişi yapmamız gerekir.

Bu iletişim gereklilikleri, doğal olarak işlem mantığı kodlarımızın içine kadar nüfuz eder. Bunun kod bakımı açısından bir sakıncası yoktur. Sadece test ederken, dış bir sistemle iletişim gerekliliklerinden dolayı birim testinden geçirmek için çağırdığınız kod parçacıkları, bir birim test ortamı içinde işlediklerinden habersiz *o dış sisteme* bağlanabilmeye çalışacaklardır. Ve bunda başarısız olacaklardır, çünkü sonuçta, içinde işlem yaptığımız ortam bir *test* ortamıdır. Gerekli olan tüm dış servisler büyük bir ihtimalle daha başlatılmamıştır, ve zaten başlatılmalarının da beklenmemesi gerekir: Bir birim testinden beklediğimiz, komut satırından, hiçbir dış sisteme ihtiyaç duymadan programın çetrefil mantığını (onu çagır bunu çağır mantığını değil) test edip bize sonucu bildirmesidir.

O zaman kurumsal uygulamaları nasıl edeceğiz? Kurumsal uygulamalar bir ada değildir, ama birim testler uygulamanın bir ada olmasını ister. O zaman ada olmayan kodları ada hâline getirmenin tekniklerini öğrenmemiz gerekiyor.

Bir Nesneyi Taklit Etmek

Taklit etmek (mocking), kurumsal uygulamaları birim testinden geçirebilmek için ihtiyacımız olan tekniktir. Bir birim testinin çağırdığı işlem kodlarının *bağlanmak istediği* dış sistemin yerine taklitini koyarsak ve o taklit nesneye istediğimiz (o test senaryosu için gereken) verileri verdirebilirsek, işlem kodlarımız hiçbir şeyden habersiz işleyip geri dönebileceklerdir. Birim testleri de böylece istedikleri işlem mantığını test edebilmiş olacaklardır. Yâni birim testlerinin işlemesi için, dış sistem nesneleri, taklit edilmelidir.

Taklitlemeyi, her dış sistem için değişik bir şekilde yapacağız (taklitlemeden muhaf tutacağımız tek dış sistem veri tabanı olacak, bunun sebeplerini 7.1.3 bölümünde okuyabilirsiniz).

- **Socket, vs gibi dış sistemlere ve ya API'ları çâğıran kodlar için**: jMock aracılığı ile bir nesnenin yerine metotlarının içi boşaltılmış ve sadece bizim istediğimiz cevapları döndüren bir Java taklit nesnesi koymak mümkündür.

- **Diğer**: Bu tür şartlar daha çetrefil taklitleme gerektirebilir, ve elle kodlama isteyebilirler. O zaman, dış sisteme bağlanan o nesneden miras alıp (`extend`), metotlarının içine testimizin beklediği cevapları verecek kodlar koyarız. Bu, jMock kütüphanesinin otomatik, "perde arkasında" yaptığının "perde önünde" olan karşılığıdır.

jMock ile istediğimiz herhangi bir nesnenin yerine onun taklidini koyabilir, ve bu taklidin metotlarına "dinamik olarak" istediğimiz cevapları verdirebiliriz. jMock, arka planda CGLIB adında baytkod üretmeyi sağlayan bir araç kullanmaktadır. CGLIB sayesinde jMock, meselâ dönüş değeri bir `String`

olarak tanımlanan metot ismi ve onun "beklediğimiz dönüş değerlerini" dinamik olarak, sanki esas taklit edilen nesneden geliyormuş gibi, işlem anında üretebilmektedir.

jMock kullanımını bir örnek üzerinde görelim. Alttaki kodlar içinde `SocketClient` adlı bir nesne görüyoruz. Bu nesne, İnternet'teki bir makina ve o makinadaki port'a bir Socket açmaktadır. Bu makinanın nerede olduğuna dikkat edelim. Evet, Japonya'da! Şimdi, bu `SocketClient`'ı kullanan işlem mantığı koduna gelelim. Bu class'ın ismi de (uygun olarak) `AppLogic`, ve yaptığı da `SocketClient` üzerinden Japonya'ya bağlanıp oradan aldığı Japonca bilgileri Türkçe'ye tercüme etmek.

```
public class SocketClient {

    public String takeWordFromJapan(){

       String inputLine = "";

       try{
            Socket socket = new Socket("www.japan.jp", 5555);

            PrintWriter out = new PrintWriter(socket.getOutputStream(),
                                              true);
            BufferedReader in = new BufferedReader(new InputStreamReader
            (socket.getInputStream()));

            out.println("DOMO ARIGATO!!! ");
            while ((inputLine = in.readLine()) != null) {
               out.println(inputLine);
            }
            out.close();
            in.close();
            socket.close();

        } catch(IOException e){
            e.printStackTrace();
        }

        return inputLine;

     }
}

public class AppLogic {

    SocketClient client = new SocketClient();

    public void setSocketClient(SocketClient client) {
```

```
        this.client = client;
    }

    public String translate() {

        String message = client.takeWordFromJapan() ;

        if (message.equals("HAI")) {

            return "evet";

        } else if (message.equals("DOZO")) {

            return "lutfen";

        } else if (message.equals("CAMPARE")) {

            return "??";

        }

        return "";
    }
}
```

Bizim amacımız, `AppLogic` nesnesini test etmektir. O zaman `AppLogic` kodlarını dışarıdan izole olmuş bir ada hâline getirmemiz gerekiyor. `AppLogic` kodlarına bakarsak, test ettiğimiz mantığın `translate` metotu içindeki `if` komutunun olduğunu görüyoruz. Demek ki, `SocketClient` yerine bir taklidini koyarsak, kodlarımızı dış dünyadan izole etmiş olacağız, ve bu taklide testimiz içinde istediğimiz Japonca kelimeyi "söylettirebilirsek", bu senaryonun gerektirdiği `translate` metotunu test edebilmiş olacağız. İşte jMock burada sahneye giriyor. jMock kullanan bir test aşağıdaki gibi olacaktır.

```
1  public class TranslateTest extends MockObjectTestCase {
2
3      public void testHai() {
4
5          Mock mockObj = mock(SocketClient.class);
6
7          AppLogic logic = new AppLogic();
8
9          logic.setSocketClient((SocketClient)mockObj.proxy());
10
11         mockObj.expects(atLeastOnce()).
12             method("takeWordFromJapan")
13             .will(returnValue("HAI"));
14
```

```
15          String answer = logic.translate();
16
17          assertEquals("evet", answer);
18      }
19  }
```

#5'te `SocketClient` nesnesinin taklit tanımını yapıyoruz. #7'de test edeceğimiz `AppLogic` nesnesini yaratıyoruz, ve taklit nesnesinin kendisi #9'da yaratıp `AppLogic` üzerinde set ediyoruz. Bu numaraya dikkat! Eğer `AppLogic` ihtiyacı olan `SocketClient` nesnesini `translate` metotu *içinde kendisi* yaratıyor olsaydı, biz bu şekilde taklide dayalı bir testi yapamazdık. Çünkü işlem kodları bizim verdiğimiz taklit dış nesnesi yerine, kendi yarattığı ve Japonya'ya bağlanmaya çalışacak esas nesneyi kullanırdı. O zaman, işlem kodlarımızın dış nesneleri her zaman dısarıdan, bir set aracılığı ile alması gerekmektedir.

Taklit nesnesine belli cevapları verdirmek için ise, jMock'un `expects`, `method returnValue` komutlarını kullanacağız. Bu özellikler jMock teknolojisinin en güçlü tarafını teşkil etmektedir, dinamik, `String` ile tanımladığımız metot isimleri ve dönüş değerlerini arka planda baytkod hâline gelip, JVM tarafından anında işleme konulmaktadır! Üstteki örnekte, #11. satırda taklit nesnesine beklenen (ve taklit edilmiş) çağrının "en az bir kere geleceğini" tanımlamış oluyoruz. Yâni `expects(atLeastOnce())` kullanımı sayesinde, taklit edilmiş nesneye "kaç kere çağrı yapılıyor olmasını" bile test senaryomuza dahil etmemiz mümkün olmaktadır. Bu da önemlidir, çünkü eğer işlem kodları biz dış nesne çağrılması beklerken hiç çağrı yapmazsa, bu da bir hata durumu olacaktır ve bu durumun yakalanması kodumuzun doğruluğu açısından faydalı olabilir.

Taklit nesne üzerinde çağırılacak metodun ismini #12'de tanımlıyoruz. #13, dönüş değerini tanımlıyor. Bu hazırlıklardan sonra, artık `AppLogic`'i çağırarak, teste gelen sonucu kontrol edebiliriz.

#15'te yapılan `translate` çağrısı, `AppLogic`'den taklit `SocketClient`'a yapılan bir `takeWordFromJapan` çağrısına sebebiyet verecektir. Biz de zaten bu senaryoyu planlamıştık ve #11'deki `expects(atLeastOnce())` tanımının sebebi buydu. Beklenen bu çağrının cevabını da pişirip taklit nesneye hazırlatmıştık, cevap "HAI" kelimesi olacaktı.

Bu cevap gelince, sıra (esas test ettiğimiz) `AppLogic` class'ına gelir, `translate` kodunun geri kalanı işler. "HAI" görünce "evet" Türkçe cevabı verilmesi gerektiğine, `if` kodları karar verecektir (eğer doğru yazılmışlarsa). Eğer hakikaten geriye "evet" cevabı döndürülürse, bu cevabı kontrol eden JUnit testimiz, başarıyla senaryonun geçtiğini rapor edecektir.

Üstte gösterilen örnek kodların tümünü `JUnitSample` projesi altında bulabilirsiniz. Birim testleri işletmek için `ant test` komutunu kullanınız.

Kendi Kodumuz ile Taklit Etmek

Eğer jMock ile taklitleme tekniğini kullanmak istemezsek, taklit nesnesini kendimiz de yaratabiliriz. Bunun için yapmamız gereken, taklit edilen nesneden miras

alıp, taklit edilen metotu tekrar tanımlamaktır (redefine). Meselâ `SocketClient` class'ının taklitini elle şöyle yazabiliriz.

```
public class SocketClientManualMock extends SocketClient {

    private String returnThis;

    public void setReturnWhichWord(String word) {
        this.returnThis = word;
    }

    // tekrar tanım
    public String takeWordFromJapan(){
        return returnThis;
    }
}
```

Bu nesnede, görüldüğü gibi, hiçbir socket işlemi yapılmıyor. Japonya'dan `String` alması gereken metot, sadece, daha önceden set edilmiş bir `String` değerini döndürmek üzere ayarlanmıştır. Bu düzen, jMock şartlarında gördüğümüz `will(returnValue(..))` kullanımına eşdeğerdir. Testin kendisi de şöyle gözükecektir.

```
public class TranslateManualMockTest extends TestCase {

    public void testHai() {

        SocketClientManualMock mockObj = new SocketClientManualMock();

        AppLogic logic = new AppLogic();

        logic.setSocketClient(mockObj);

        mockObj.setReturnWhichWord("HAI");

        String answer = logic.translate();

        assertEquals("evet", answer);

    }
}
```

Bu testin de işleyip başarıyla geri döndüğünü göreceğiz.

Bu yöntemin, jMock yöntemine nazaran dezavantajı fazladan bir class yazılmasını macbur bırakmasıdır. Bu da kod bazımızda class enflasyonuna yol açabilir. jMock ile dış class dinamik olarak üretilmiştir, ve hiçbir ek class'a gerek yoktur. İki yöntem arasında seçim yaparken tavsiyemiz, taklitlemeyi önce jMock ile başlamak eğer bir şekilde takılınır ve çok çetrefil bir ortamda jMock kullanımı zorlamaya başlarsa, elle taklitleme yapılmaya başlanmasıdır.

7.1.3 Hibernate Test Altyapısı

`SimpleHibernate` örnek projesinde Hibernate üzerinden veri tabanına erişen birçok senaryoyu görmeniz mümkündür. `SimpleHibernate` JUnit test kodlarını çalıştırmak için, komut satırından `ant test` komutunu vermeniz yeterlidir. Bu bölümde, Hibernate kodlarını test edebilmek için faydalı bir class `org/mycompany/util/TestUtil` kodlarını göreceğiz. `TestUtil` ile, bir test senaryosu için gereken test verisini yüklememiz mümkün olacak.

Test Verisi

Hibernate (ya da başka tekniklerle) veri erişimi yapan kodları test etmenin en zor tarafı test başlamadan önce veri tabanında belli bir test verisinin olması zorunluluğudur. Bu test verilerin yüklenmesini tabii ki yine Hibernate'i kullanarak ekleme, güncelleme komutları üzerinden yapmak mümkündür, fakat, test etmeye uğraştığımız zaten Hibernate veriye erişim kodlarının kendisi değil midir? Test edilen kodları teste hazırlık için kullanmak garip bir tavuk/yumurta yumurta/tavuk ilişkisi ortaya çıkarmaktadır. Bu sebeple test verisi yüklemek için pür SQL kullanarak (SQL komutları içeren bir dosyadan) veri yüklemek ideal yöntem olacaktır. Zaten test için gereken veri bazen "mevcut bir tabandan" "SQL formatında" geliyor olabilir. O zaman bu veriyi bir `.sql` dosyası üzerinden kullanabilen bir test altyapısı bizim için faydalı olacak.

`TestUtil` kodları birim testi başlar başlamaz sıfırdan şema kurmak ve veri yükleme işlemleri için tarafımızdan yazılmış bir class'tır. Hibernate kodlarını test etmek için gereken veri yükleme işlemini `TestUtil.createFromFile` ve `TestUtil.insertFromFile` metotları ile yapabilirsiniz.

Yâni veri tabanında şema yaratımı ve veri yüklemesi için bize iki `SQL` dosyası gerekiyor. Bu dosyalardan biri şema üretmemizi sağlayacak `CREATE TABLE` komutları içeren dosya olmalıdır, ikincisi, `INSERT` içeren veri yükleme dosyası olmalıdır. Bu iki dosyanın bir örneğini aşağıda görüyoruz.

Liste 7.1: tables_mysql.sql

```
DROP TABLE IF EXISTS car;
CREATE TABLE car (
  license_plate varchar(30) default '',
  description varchar(30) default ''
) TYPE=MyISAM;
```

Liste 7.2: sample_data.sql

```
truncate table car;
insert into car(license_plate, description)
      values('34 TTD 2202','ornek description');
```

Bu dosyaları taban üzerinde işletmek için `TestUtil`'in iki çağrısını sırasıyla `tables_mysql.sql` ve `sample_data.sql` üzerinde kullanıyoruz. `TestUtil`, bu

dosyaların içeriğini `hibernate.cfg.xml` içinde belirtilmiş tabanda işletecektir. Böylece tabana istediğimiz örnek veri yüklenmiş olacaktır.

Liste 7.3: SimpleCarTest.java

```
public class SimpleCarTest extends TestCase {

    public SimpleCarTest() { }

    public void testCar() throws Exception {
        Connection c = TestUtil.createTestConnection();

        TestUtil.createFromFile("tables_mysql.sql", c);
        TestUtil.insertFromFile("sample_data.sql", c);

        try {
            Session s = HibernateSession.openSession();
            HibernateSession.beginTransaction();
            Car car = (Car) s.get(Car.class, "34 TTD 2202");
            assertEquals("ornek description", car.getDescription());

            HibernateSession.commitTransaction();

        } finally {
            HibernateSession.closeSession();
        }
    }
}
```

`SimpleCarTest` çok basit bir testi gerçekleştirmektedir. Test başında `SQL` dosyalarının belli verileri yüklediğini bildiğimiz için testin tek yapması gereken, Hibernate ile veriye erişimi test etmektir. Örnekte `Car` eşlemesi üzerinden Hibernate `get`'i test etmiş oluyoruz. Hibernate `get`, kendisine verilen ID üzerinden tek bir nesneyi yüklememizi sağlayan bir çağrıdır. `Car` eşlemesindeki `licensePlate` öğesi kimlik olduğu için `get`'e (daha önce test verisini yüklemiş olduğumuz) bir `licensePlate` verince, geriye tek bir nesne gelmesini bekleriz. JUnit birim testi de aynen bunu yapmaktadır. Kontrol amaçlı olarak biraz önce okuduğumuz nesne üzerinde `description` olarak `'rnek description'` metnini bulmayı amaçlıyoruz (çünkü örnek veride böyledir), ve bu şartı `assertEquals` ile kontrol ediyoruz.

`TestUtil` kullanan JUnit testleri hakkında bilmemiz gereken diğer önemli noktalar şunlardır:

- `TestUtil`'in `TestUtil.createFromFile` ve `TestUtil.insertFromFile` metotlarına geçilen her `sql` dosya, `CLASSPATH` belirtilen bir dizinde mevcut olmalıdır. `TestUtil`, kendisine işletilmesi için verilen `sql` dosyalarını CLASSPATH altında aramak üzere programlanmıştır. `SimpleHibernate`

ve diğer tüm kitap örnek kodlarında `src/sql` dizinini biz CLASSPATH'e ekledik.

- Her test başında `TestUtil.createFromFile` ile şemayı sıfırdan yaratmamız çok önemlidir, çünkü kodlama süreci içinde şemamız başkası tarafından değiştirilmiş olabilir. Kodlar ve şema bir bütün olduğu için, en son kod, en son şema ile çalıştırılmalıdır. Bu sebeple her test başında şemayı sıfırdan yaratarak, kod ve şema uyumsuzluklarını birim test seviyesinde yakalayabilmiş oluyoruz. Kural #3 bağlamında, basit bir alışkanlık edinerek (her test başında şema tekrar yaratılır), şema ve kod arasındaki uyumsuzlukları erkenden yakalama şansına kavuşuyoruz. Emin olun ki bu uyumsuzlukların kod içerisinde haftalarca kalıp büyümesi kod kalitesi açısından hiç iyi olmayacaktır!

- Birim test felsefesine göre her birim testi kendi kendine yeter bir şekilde yazılmalıdır, ve bir birim testinin sonucu diğer birim testini etkilememelidir (her birim testi başında şemayı sıfırdan kurmak için bir sebep daha, çünkü bir test sonucunu bir ötekini veri tabanı üzerinden bile etkilememelidir).

Hibernate Birim Testleri Hangi Tabana Bağlanıyor?

Örnek projelerimizdeki birim testleri, testlerin işlediği makinadan erişilebilen bir veri tabanının varlığını şart koşmaktadır. Fakat gereksinimler bundan ibarettir. Meselâ testlerimizin çalışması için tabanda hiçbir şemanın kurulmuş olması gerekmez. `TestUtil` aracılığı ile şemayı sıfırdan kurmak JUnit birim testleri içinden otomatik olarak gerçekleştirilecektir.

> **Not**: `TestUtil` bu bağlamda uzun bir değişim sürecinden geçti. Kodun daha önceki versiyonları, birim testleri işleten kişinin bilgisayarında bir veri tabanının olamayacağını ihtimale katarak, SQL şema script'lerini MySQL ya da Oracle formatından HSQLDB formatına anında çevirmeye uğraşıyordu. HSQLDB, gömülü (embedded) bir taban olduğu için, kullanmak için bir servis başlatmaya gerek bırakmaz, bu sayede JUnit tarafından rahatlıkla hafızada başlatılarak, yüklenip testler için hazır hâle getirilebilir. Bu yöntemi takip etmekte amaç, test işletici üzerinde en az külfeti getirmek idi.
>
> Fakat, bu yöntemin işlemesinde problemler bulunmuştur;
>
> 1. HSQLDB, bazı çetrefil Hibernate sorgularını işletmekte problemler çıkarmış
> 2. Oracle ya da MySQL şema script'lerinden HSQLDB script'lerine otomatik olarak çevirim yapan Perl script'lerinin bakımı ve kodlaması fazla zaman alıcı bir eylem hâline gelmiştir.

> Bu sebeple, Kural #3 ışığında birim testlerini işletmek isteyen her programcının 'ciddi bir veri taban ürününe erişiminin olması' prensibini takip etmek uygun gözükmüştür.

Değişik Test Senaryoları

Eğer birim testlerinizde değişik senaryoları test etmek isterseniz, bu, çoğu zaman *değişik örnek veri* kullanmak anlamına gelecektir. Bu yeni senaryolarda (ve JUnit testlerinde), şemayı üreten satır önceki örneklerimizdeki gibi kalacak, fakat *değişik* bir örnek veri dosyasını işleten ve *değişik* değerleri `assertEquals` ile kontrol eden bir kod olacaktır. Meselâ, bu şekilde bir yeni testi altta görelim:

Liste 7.4: SimpleCarTestManyRows.java

```
public class SimpleCarTestManyRows extends TestCase {

    public SimpleCarTest() { }

    ...

    public void testCarNew() throws Exception {
        Connection c = TestUtil.createTestConnection();

        TestUtil.createFromFile("tables_mysql.sql", c);
        TestUtil.insertFromFile("sample_data_2.sql", c);

        try {
            Session s = HibernateSession.openSession();
            HibernateSession.beginTransaction();
            Car car = (Car) s.get(Car.class, "52 TT 30");
            assertEquals("description 4", car.getDescription());

            HibernateSession.commitTransaction();

        } finally {
            HibernateSession.closeSession();
        }
    }
}
```

Yeni testimizin değişik veriye ihtiyacı var. Bu veriyi, alttaki dosyadan sağlayabiliriz.

Liste 7.5: sample_data_2.sql

```
truncate table car;
insert into car(license_plate, description)
   values('34 TTD 2202',description 1');
insert into car(license_plate, description)
   values('14 TF 399','description 2');
```

```
insert into car(license_plate, description)
   values('34 RF 493','description 3');
insert into car(license_plate, description)
   values('52 TT 30','description 4');
```

Görüldüğü gibi yeni testte yüklenen veri `sample_data.sql` değil, `sample_data_2.sql` adlı dosyadır, çünkü testimiz için yaratmamız gereken senaryo bunu gerektiriyordu. Yeni veri üzerinden testin kontrol edeceği araba nesnesi, `52 TT 30` no'lu plakayı taşıyan arabadır. Test, `get` komutu ile bu nesneyi yükler `assertEquals` ile sonucu kontrol eder.

7.2 Kabul Testleri

Birim testleri en ufak modül (metot) bazında test yapıyorsa, kabul testleri de (acceptance test) programın dışından, birden fazla modüle dokunabilecek şekilde bir kullanıcı gözünden test etmemize yarar. Kabul testlerine sektörde fonksiyonel (functional) test, ya da entegrasyon (integration) testleri isimleri de verilmektedir. Kabul testleri mümkün olduğu kadar dış sistemlerle entegrasyonu yapılmış bir sistem üzerinde gerçekleştirilmelidir.

Kabul testlerini en basit şekilde elle (uygulamayı bizzat kullanarak) yapmak mümkündür. Uygulamamız test edilmeye hazır hâle gelince kodlar test makinasına gönderilir, ve bir test kullanıcı(lar) belli test senaryolarını uygulama üzerinde işletip, uygulama için kabul testlerinden geçti ya da kaldı raporu verebilirler. Test kullanıcısı kabul testlerini gerçekleştirirken elinde belli test senaryoları olur. Kullanıcı bu senaryoları sırasıyla program üzerinde işletir sonuçlarını kontrol eder, ve bir yere not eder. Hatalı gözüken sonuçlar, hata takip sistemine (bug tracking system) girilir, ve düzeltme süreci başlamış olur.

7.2.1 Otomatik Kabul Testleri

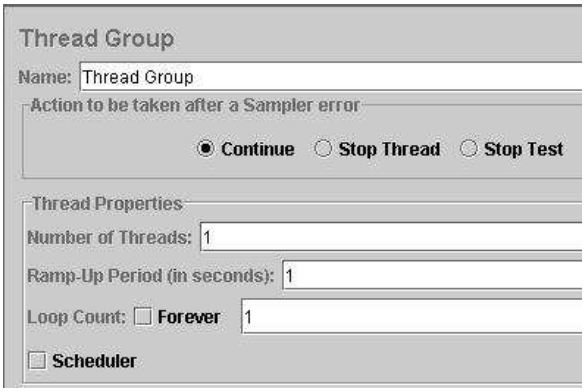
Elle yapılan testler için zaman ve insan gücü harcanması gerekir. Fakat yapılan işin oldukça mekanik olması sebebiyle otomatik yapılan kabul testleri de artık tercih edilmeye başlanmıştır (Kural #7). Kurumsal Web ortamında otomatik kabul testleri işletmek için birçok ürün mevcuttur; Bizim tavsiyemiz, 5.3.1 bölümünde işlediğimiz ve yük testleri için kullanılan JMeter ürününü *kabul testleri için* kullanmaktır.

JMeter'i bu şekilde iki amaçlı kullanabilmemizin sebebi, bu ürününün Web uygulamasından gelen cevapları işleyebilen bir birim sağlamasıdır; Kullanacağımız bu birim, `Reponse Assertion` adlı birimdir.

JMeter Kullanımı

Bir kabul testinin yük testinden en önemli farkı, kabul testi için bağlanan tek Thread'in (sanal kullanıcı olarak) yeterli olmasıdır, çünkü test edilen arka

plan kodlarının ölçeklenebilmesi değil, doğru çalışıp çalışmadığıdır. Şekil 7.1 üzerinde gereken Thread ayarlarını görüyoruz.



Şekil 7.1: Kabul Test için JMeter Thread Sayısı

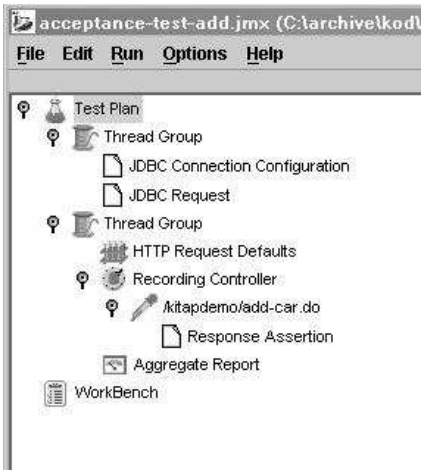
Geri kalan ayarlar, 5.3.1 bölümünden tanıdık olacaktır. JMeter sanal kullanıcısının bir dinamik sayfaya bağlanıp form'a test değerleri girmesi için `Http Request` biriminin eklenmesi ve gerekli parametrelerin bu birime verilmesi yeterlidir. Örnek için `StrutsHibPerformance` projesinin `resources/acceptance-test-add.jmx` dosyasında içinde tanımlanmış `/kitapdemo/add-car.do` birimine bakabiliriz. Bu örnek kabul testini `StrutsHibPerformance` projesi üzerinde kullanabiliriz.

Örnek test dosyasını, JMeter menüsü `File | Open` ile açabilirsiniz. Şekil 7.2 üzerinde kabul testin tüm birimlerini, Şekil 7.3 üzerinde ise `/kitapdemo/add-car.do` biriminin detaylarını görüyoruz.

Bir kabul testinin yapması gereken doğruluk kontrollerini JMeter'ın `Response Assertion` birimini kullanarak yapabiliriz. Bu kontrol, JUnit birim testlerinden kullandığımız `assert` metotları ile eşdeğer bir özelliktir; Amacımız, testin beklediği cevap değerleri uygulamadan geri gelmezse, bu kontrol edici birim ile bu hatayı yakalayıp bize bildirmesini sağlamaktır.

`Response Assertion` birimi, yapılan bir Web isteğinden sonra geri gelen cevap (response) içindeki değerlere bakabilme yeteneğine sahiptir. Bu cevap içeriği, bir Web sayfasını tarayıcınız ile ziyaret edince tarayıcınıza gönderilen içerik ile aynıdır. Tarayıcıya gelen HTML içeriği görmek için Mozilla'da `View | Page Source` seçeneğini kullanılabiliriz.

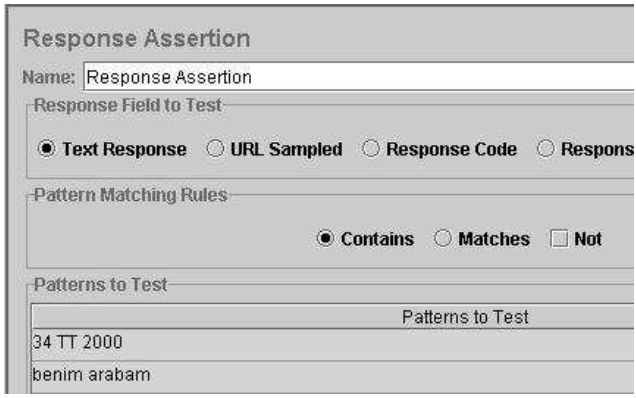
`Response Assertion` biriminin kontrolünü yapabilmesi için, hangi Web isteğinin içeriğine bakacağını bilmesi gerekiyor. Yâni `Response Assertion` her zaman bir "`Http Request`'e" *alt birim* olarak eklenmelidir. O zaman `Response Assertion` eklemek için, bir `Http Request`'e tıklayıp, mouse sağ tıklama ile `Add | Assertions | Response Assertion` seçeneğini seçmeliyiz. Eğer kontrol birimini yanlış yere eklediysek, bu birimi sürükle/bırak ile gereken `Http`



Şekil 7.2: Örnek bir Kabul Testi

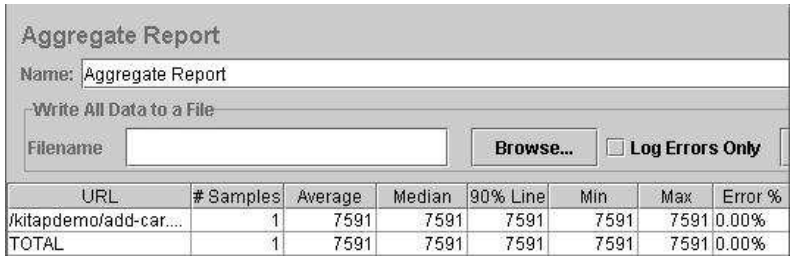
Şekil 7.3: Araba Eklemek

`Request` üzerine götürüp, `Add As Child` seçeneğini kullanmalıyız. Alt birim olarak eklenmiş `Response Assertion` birimini Şekil 7.2 üzerinde görüyoruz. Örneğimizde kullanılan `Response Assertion` içeriğini Şekil 7.4 üzerinde görülmektedir.



Şekil 7.4: Araba Kontrolü Yapan Response Assertion

Bu cevap kontrolünde, cevap metni içinde aranacak değerler "34 TT 2000" ve "benim arabam" kelimeleridir. Bu kelimelerin orada olması lâzımdır, çünkü `add-car.do` Struts Action'ını işini bitirdikten sonra yönlendirmeyi `main.do`'ya yapar. `main.do` ise, o anda sistemde olan tüm arabaları listelemek ile yükümlüdür, ve biz de daha önceden `add-car.do` ile plakası "34 TT 2000" ve açıklaması "benim arabam" olan bir arabayı sisteme eklettirdiğimiz için, bu kelimelerin araba listesinde olmasını beklememiz normâldir. `Run | Start` ile işletebileceğiniz ekleme kabul testinin sonuçları `Aggregate Report` biriminde şöyle (7.5) gözükecektir. Eğer `Error` kolonu altında `0.00%` görüyorsak, tüm testlerin başarı ile geçtiğini anlarız.



Şekil 7.5: Sonuçlar

Özet olarak, JMeter ile bir Web uygulamasını test etmek için, şunları yapmak gerekir:

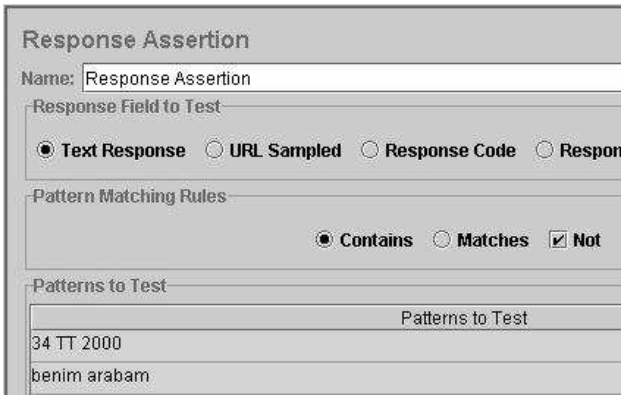
- Test edilen Action'a `Http Request` ile bazı test değerleri göndermek
- `struts-config.xml` dosyasına bakarak, iş bittikten sonra yönlendirmenin nasıl yapılacağına bakmak
- Bu yönlendirmeye göre, hangi değerlerin geleceğine göre bir kontrol edici `Response Assertion` birimini `Http Request` altına eklemek

Eğer `Http Request` birimlerini elle eklemek istemiyorsak, 5.3.1 bölümünde anlatıldığı gibi, kullanıcı programı tarayıcıda kullanırken, hareketlerinin kaydedilmesini sağlayabiliriz.

Kontrol Çeşitleri

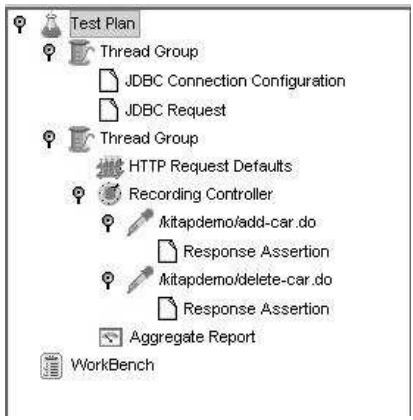
Bir `Response Assertion` tanımlarken, yapacağımız ayarlardan bir tanesi, "hangi içerik kontrolünün" yapıldığını tanımlamaktır. Üstte gösterilen kontrol, bir "mevcudiyet" kontrolü, ya da, bizim beklediğimiz değerin *olmasını* bekleyen türden bir kontroldür. Fakat bir metnin olma kontrolünü yaptığımız gibi, *olmamama* kontrolünü de yapabiliriz. Bu durum, meselâ bir arabanın sistemden silindiği `delete-car.do` Action'ı için gereklidir. Bu Action bir araba sildiği için, mantıken o arabanın bir sonraki araba listesi ekranında olmaması gerekecektir.

Eğer `Response Assertion`'ın yaptığı kontrol çeşidini değiştirmek istersek, bunu `Pattern Matching Rules` ayarları altından yapabiliriz. Meselâ, örnek olarak biraz önce eklediğimiz arabayı `delete-car.do` ile silelim, ve `delete-¬car.do` altına koyacağımız bir kontrol Şekil 7.6 üzerindeki gibi olsun. Bu kontrolün detaylarında, uymama kontrolü için `Contains` seçeneğine ek olarak, `Not` (değil) seçeneğini de seçmemiz gerekti. Eğer `Not` seçilmemiş olsaydı kontrol, pozitif bir kontrol olacaktı.

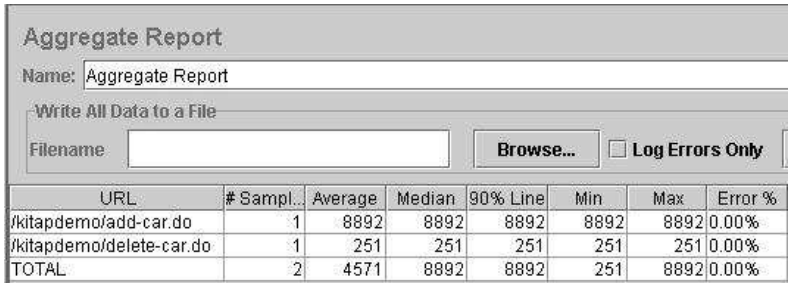


Şekil 7.6: Uymama Kontrolü

Bu testi de işletince, geriye gelen sonuç Şekil 7.8 üzerindeki gibi olacaktır. Yine hata yüzdesi `0.00%` seviyesindedir, yâni tüm testler başarıyla geçmiştir.



Şekil 7.7: Ekleme ve Silme Testleri



URL	# Sampl...	Average	Median	90% Line	Min	Max	Error %
/kitapdemo/add-car.do	1	8892	8892	8892	8892	8892	0.00%
/kitapdemo/delete-car.do	1	251	251	251	251	251	0.00%
TOTAL	2	4571	8892	8892	251	8892	0.00%

Şekil 7.8: Ekleme ve Silme Test Sonuçları

7.3 Ne Kadar Test Gerekli?

Birim ve kabul testlerini ciddi bir şekilde uygulamaya karar verilince ve teknikleri öğrenilince, programcıları en çok düşündüren kararlardan biri "ne kadar testin yeterli olduğu" sorusudur. Ne de olsa testler, varlığı ve yokluğu gereklilik kodları seviyesinde kontrol edilen işler değillerdir. Eğer yazmazsak, yazılmayan testler yapılmayan kontrollerdir. Uygulamamız üzerinde bir değişiklik olmaz.

7.3.1 Birim Test Miktarı

Birim testleri için "ne kadar" sorusunun cevabında iyi bir kulağa küpe kural (rule of thumb) öncelikle kodun ne kadar çetrefil olduğu ile alâkalıdır. Kodun çetrefilliğini en basit olarak, üstün körü bir şekilde, koda çıplak gözel bakış kullanarak bile yapılabilir. Ne zaman ki bir metot kodlarının girinti (indentation) seviyesi iki ya da daha fazla seviye içeri giriyor, o zaman o metotu test etmemiz iyi olacaktır. Girinti seviyesi, bir metot içinde `while`, `if`, `else` komutlarının kullanımına bir işaret olduğu için, basit bir litmus testi olarak faydalı bir göstergedir.

Diğer şartlarda, birim testi yazmak ve yazmamak kararı için her durumu ayrı ayrı tartabiliriz. Biz, meselâ Hibernate kullanılan kurumsal kodlarımızda her kalıcı nesne ve o nesne üzerindeki her ilişkiyi bir şekilde kullanan (genelde bir `get` komutu ile) bir testi yeterli buluyoruz. Bu çeşit bir test en azından eşlemelerin doğru yapıldığına dair bize bir rahatlık vereceği için yapılması faydalı olmaktadır. Ama her kalıcı nesne için dört işlemin (ekle, sil, güncelle, yükle) test edilmesi fuzuli (overkill) olabilir. Bu tür birim testlerini yazmak hızlı geliştirmenize engel olacağı için tarafımızdan tavsiye edilmeyecektir.

7.3.2 Kabul Test Miktarı

Felsefe olarak şunu da eklemek gerekiyor. Kabul testleri birim testlerinden daha önemlidir. Tecrübe gösteriyor ki, ufak, mikro seviyede yazılan testlerden geçen ve doğru çalıştığı zannedilen bir uygulamanın bütününün çalışmaması mümkündür. Özellikle birden fazla dış sistemin biraraya geldiği kurumsal uygulama dünyasında iç ve dış entegrasyon, sistemin işleyişi için had safhada önem taşır. O zaman kabul testlerini, özellikle anlattığımız şekilde otomatik olanlarını hazırlamamız, ve güncel tutmamız önemlidir.

Kabul test miktarı açısından, en azından, her orta ve üstü zorlukta olan gereklilik (functionality) ve bu gerekliliğin değişik senaryolarının test edilmesi iyi olacaktır. Eğer daha fazla zaman ve kaynak ayırabiliyorsanız, en basit sayfaları bile otomatik test havuzunuza dahil etmemiz, projemiz için ileri safhalarda çok faydalı bir seçim olacaktır.

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*.

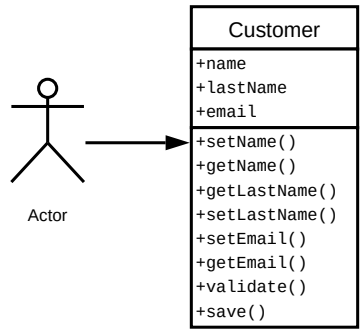
Diğer programcılık dallarında bu her zaman geçerli olmayabilir. Meselâ bilimsel formül hesabı (number crunching) yapması gereken türden programlar, dış sistemlerle fazla etkileşime girmezler. Girdi olarak bir dosya alırlar, ve çıktı olarak bir diğer dosya üretirler (ya da çizim (plot) basabilirler). 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

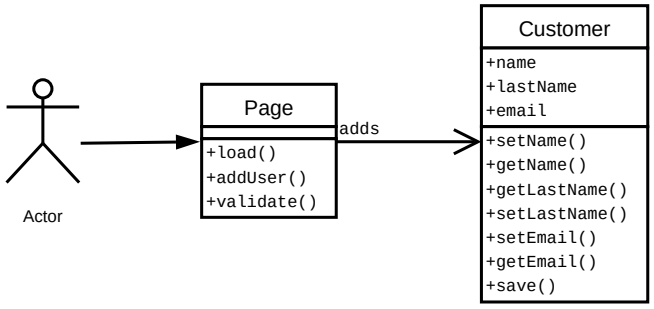
Bu modele göre önyüz, bir `Customer` (müşteri) nesnesini yaratacak, üzerinde `setName`, `setLastName`, vs. ile gereken verileri koyacak, daha sonra veri doğruluk kontrolü için `validate` çağrısını yapacak, ve en sonunda `save` ile müşteri nesnesini kaydedecektir.

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.

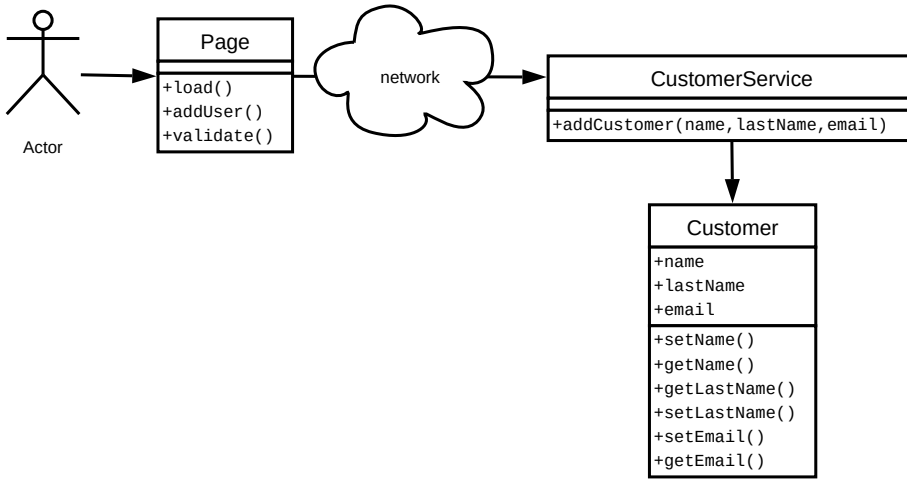


Şekil 8.2: Network Üzerinden Müşteri Eklemek

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.

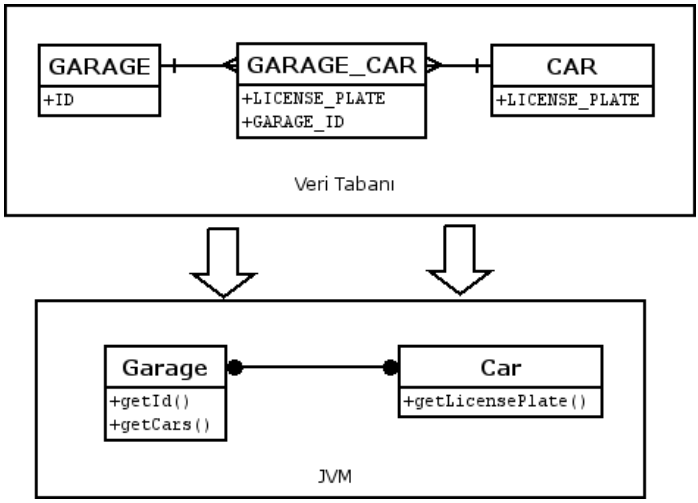
Fakat böyle basitçi bir yöntem ne yazık ki bizi fazla uzağa götürmez. İnsan dili değişik nüanslara çok açıktır ve çok daha katı kurallara göre yazılması gereken bir bilgisayar sistemi için bizi yanlış yöne sevkedebilir. Mesela üstteki örnekte aslında class olması gereken şey, *hareket*'tir. Ama bu kelime 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();
}
```

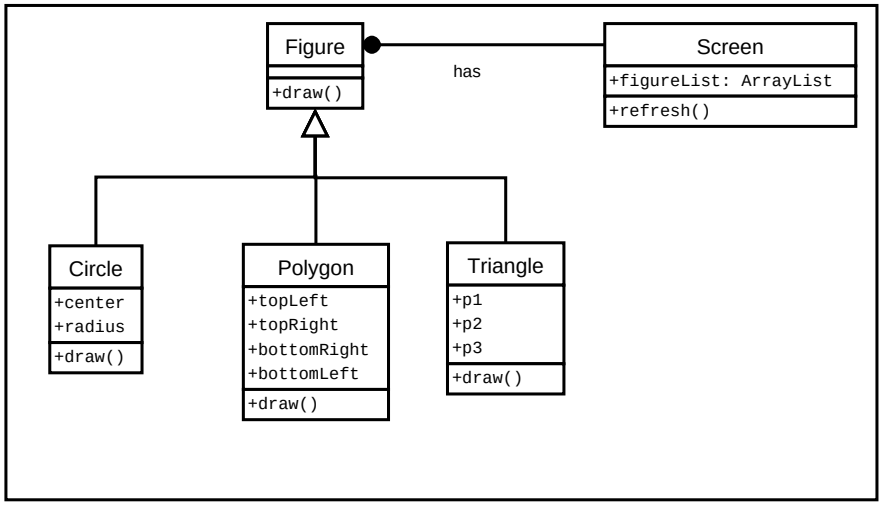
ile tüm figürleri tekrar çizmemiz mümkün olacaktır.

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:

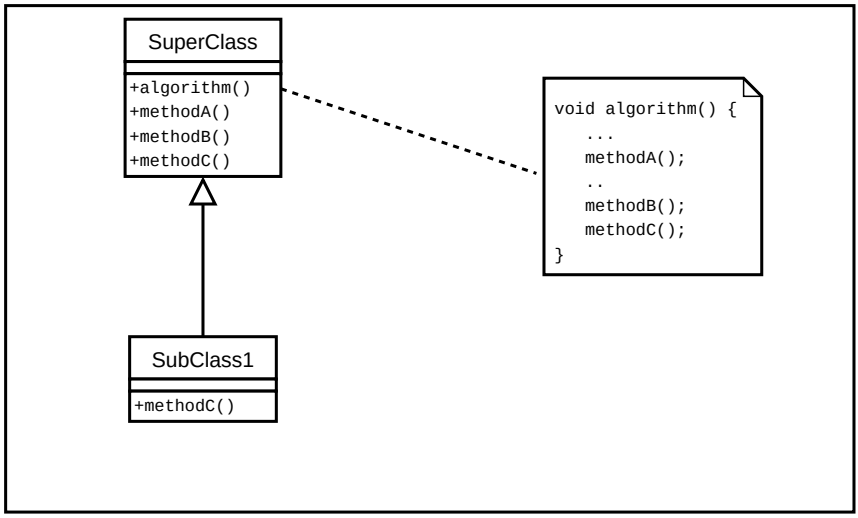
- Command
- Template Method
- Facade
- Singleton



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.

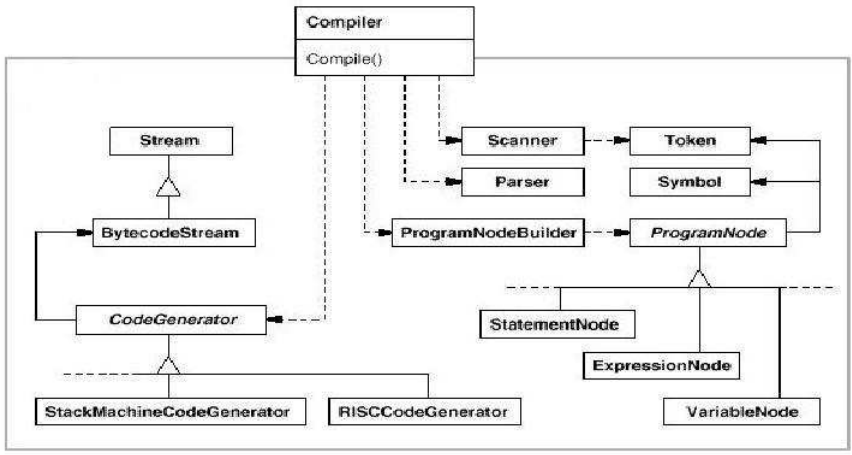
Facade

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.

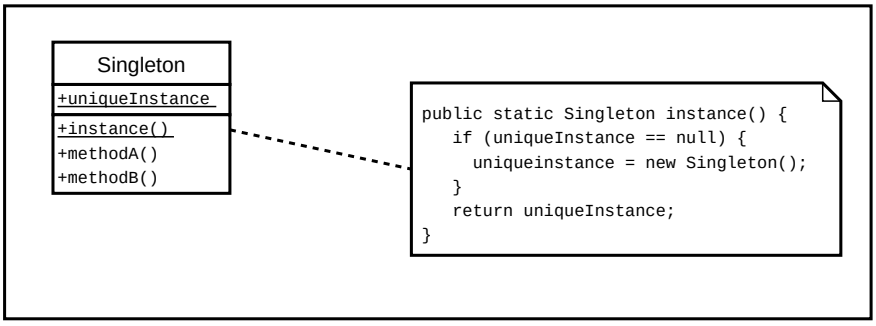
8.3.2 POJO'lar ve İşlem Mantığı

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.

Bölüm 9

Veri Tabanları

Bu Bölümdekiler

- İlişkisel Teori ve Kavramları
- İndeksler, Görüntüler, Tetikler
- Oracle, PostgreSQL ve MySQL'i Unix üzerinde kurmak

Her kurumsal uygulamanın etrafında döndüğü, bir depoya alıp verdiği, sorgulayıp bulduğu ve biriktirdiği metin ve sayısal bilgilerin toplamına *veri* diyoruz. Bir kurumsal uygulamada veriye erişim, en hayati noktalardan biridir. İşyerleri verilerinin doğruluğuna ve bütünlüğüne çok önem verirler. Hâtta kitabımızın konusu olan kurumsal uygulamalar için denebilir ki, aslında bir uygulamanın yegâne görevi veri tabanındaki veriyi göze güzel bir şekilde sunmaktan ve göze güzel gelen şekilde almaktan ibarettir.

Veri depolama ve geri almanın modellerinden biri olan ilişkisel model, E. F. Codd [9] tarafından ortaya sürülmüş ve günümüzdeki modern tüm ilişkisel veri tabanların işleyişini tanımlayan bir teorik altyapıdır. Bu modelin temeli küme teorisine dayanır, ve tamamıyle tanımlı, iç bütünlüğü kurulmuş sağlam matematiksel bir yapıdır. Yapı öyledir ki, bir ilişkisel model üzerinde işleyen bir sorgunun "doğru olup olmadığını" bile matematiksel olarak ispatlayabilirsiniz[1]. İlişkisel model, ortaya atıldığı 70'li yıllarda, akademik çevrelerde müthiş bir çarpışmaya sebebiyet vermişti. O vakitte yarışta olan diğer veri modeli olan hiyerarşik model, ilişkisel model ile girdiği çarpışmadan yenik ayrılmıştır[2]. Akabinde ilişkisel modeli ticari ürün hâline getiren ve SQL'i ilk destekleyen şirketlerden olan Oracle, bu alana çok hızlı girmesiyle müthiş bir piyasa hakimiyetine ve finansal başarıya kavuşmuştur. Önyüze odaklı olan Microsoft'tan sonra bir servis tarafı teknolojisi satıcısı olan Oracle'ın ikinci büyük yazılım şirketi olması, veri depolaması ve erişiminin önemine işaret etmektedir.

Günümüzde artık SQL dili ile erişilen ilişkisel veri tabanları, ticariden açık yazılıma, küçükten büyüğe giden geniş bir yelpazede bulunabilen hale gelmişlerdir. Günümüzde eğer finans, telekom, sağlık sektöründen biri için bir kurumsal yazılım geliştiriyorsak, arka plandaki veri tabanını olarak ilişkisel bir teknoloji bulmamızın olasılığı artık çok yüksektir.

Verinin depolama ve erişimin bu seviyede önemi sebebiyle, verinin tutulduğu ilişkisel veri tabanlarını, ilişkisel şemaların altyapısını tanımlayan ilişkisel modeli, ve tüm teori ve teknolojinin bir araya geldiği SQL erişim dilini öğreneceğiz.

9.1 İlişkisel Model

İlişkisel modelin temeli, birden fazla kolonu birarada tutarak onlar arasında bir *alakâ* kurulmasını sağlayan *ilişki* kavramıdır [10, sf. 138]. İlişki, modern veri tabanlarında bir *tabloya* tekabül eder. O zaman bir tabloda, yâni bir ilişkide, tipik olarak birden fazla kolon olur. Tablo 9.1 üzerinde örnek bir ilişki görüyoruz.

Eğer bir tabloya veri eklemek istiyorsak, o tablonun ilişki yapısına uyan verileri SQL ile veri tabanına verebiliriz. Bunun için SQL'de `INSERT` komutunu kullanabiliriz. Örnek bir INSERT ibaresi altta gözükmektedir.

[1] Aynı şekilde bir analiz ne yazık ki yazılım mühendisliği gibi alanlarda mümkün olmamaktadır, çünkü bu alanlardaki eşya makina değil, insandır

[2] Nesnesel veri tabanları da hiyerarşik modelin günümüzdeki yansımalarıdırlar, ve bir türlü sektörde kabul görmemelerinin sebebi, 70'lerdeki savaşı kaybetmiş olmalarının bir sonucudur

Tablo 9.1: Örnek Tablo (Car)

LICENSE_PLATE	DESCRIPTION

Tablo 9.2: Örnek Tablo (Veri İle)

LICENSE_PLATE	DESCRIPTION
34 TTD 2202	sarı araba
34 GD 22	kırmızı araba
35 TG 54	benim tarifim
16 RR 344	bu araba ikinci el
33 RE 43	sarı araba

```
INSERT INTO CAR (LICENSE_PLATE, DESCRIPTION)
VALUES ('34 TTD 2202', 'sarı araba');
INSERT INTO CAR (LICENSE_PLATE, DESCRIPTION)
VALUES ('34 GD 22', 'kırmızı araba');
INSERT INTO CAR (LICENSE_PLATE, DESCRIPTION)
VALUES ('35 TG 54', 'benim tarifim');
INSERT INTO CAR (LICENSE_PLATE, DESCRIPTION)
VALUES ('16 RR 344', 'bu araba ikinci el');
INSERT INTO CAR (LICENSE_PLATE, DESCRIPTION)
VALUES ('33 RE 43', 'sarı araba');
..
```

Bu komut, `CAR` adlı tabloya beş satır veri ekleyecektir. Komut işledikten sonra sonuç, Tablo 9.2 üzerindeki gibi gözükecektir. `CAR` tablosundan veri almak için, SQL'in `SELECT` komutu kullanabiliriz.

```
SELECT * from CAR;
```

komutu, `CAR` arabasındaki tüm verileri ve tüm kolonları (`*` işareti ile) geri getirir. Eğer geri getirilen veriler üzerinde (satır bazında) filtreleme yapmak istiyorsak, `SELECT` komutuna `WHERE` komutunu eklemek zorundayız. Meselâ

```
SELECT * from CAR WHERE LICENSE_PLATE='34 TTD 2202'
```

komutu, plakası `34 TTD 2202` olan tüm arabaları (yani tek satır) geri getirecektir.

Bir tablodaki veriyi güncellemek için, `UPDATE` komutunu kullanmamız gerekir. `UPDATE`, güncellenmek istenen verinin yeni hâlini `set` komutundan sonra alır. Güncelleyeceği satırın hangisi olacağını ise `WHERE` komutuna verilen bir filtre değeri ile anlar.

```
UPDATE CAR set DESCRIPTION = 'siyah araba'
WHERE LICENSE_PLATE = '34 TTD 2202'
```

Tablo 9.3: Garage Tablosu

ID	DESCRIPTION
1	Garage 1
2	Garage 2

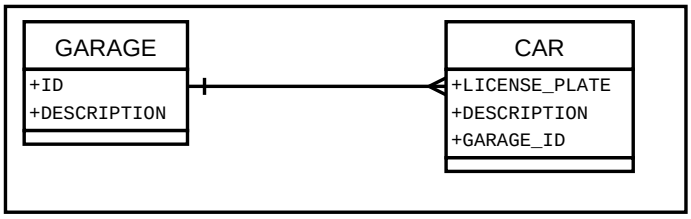
Tablo 9.4: Yeni Car

LICENSE_PLATE	DESCRIPTION	GARAGE_ID
34 TTD 2202	sarı araba	1
34 GD 22	kırmızı araba	1
35 TG 54	benim tarifim	2
16 RR 344	bu araba ikinci el	2
33 RE 43	sarı araba	2

Böylece daha önce 'sarı araba' olan tanım (description) `UPDATE` komutundan sonra 'siyah araba' hâline gelecektir.

9.1.1 Tablo Arası İlişkiler

İlişkisel modelde kolonların arasında ilişki kurulduğu gibi (bir tablo), tablolar arasında da ilişki kurmak mümkündür. Bunu yapmak için ilişkisel model, yabancı anahtar (foreign key) kavramını kullanır. Yabancı anahtar, bir tablodaki asal anahtar, kimlik niteliği taşıyan bir kolonun, diğer bir tabloya bir işaret edici olarak konmasıdır. Meselâ Tablo 9.3 üzerinde gösterilen yeni tabloya işaret etmek için, bu tablodaki bir satırı (garajı) tekil olarak seçebilen bir kolon değerini (`ID`), işaret *eden* tablonun üzerine yabancı anahtar olarak (`GARAGE_ID` kolonu) koyarız (Tablo 9.4). Bu ilişkinin kuşbakışı görüntüsü Şekil 9.1 üzerinde görülmektedir.



Şekil 9.1: Garage ve Car İlişkisi

Bütünlük Kontrolleri

İki tablo arasında yabancı anahtar ilişkisi var ise, işaret *eden* tabloya bir veri eklerken, yabancı anahtarın işaret ettiği *diğer* tablodaki satırın mevcut olmasını kontrol etmek, iyi bir prensip sayılır. Meselâ, sisteme bir `CAR` ekliyorsak ve bu sistemin iş kuralları bağlamında "bir arabanın her zaman bir garaj altında olması gerektiğini" biliyorsak, o zaman `CAR` tablosu üzerindeki `garage_id` kolonu üzerinde bir bütünlük kontrolü (integrity constraint) koyabiliriz.

Bütünlük kontolleri, bizim belirttiğimiz bir kolon ve hedef kolon üzerinde, her veri eklendiğinde veri tabanı tarafından yapılan kontrollere verilen isimdir. Bu kontrollere göre eklenen yabancı anahtarın değeri, işaret edilen tablodaki asal anahtar içinde olup olmadığı kontrol edilir; Eğer bu veri yok ise, veri tabanı `INSERT` komutunuza bir hata mesajı ile cevap verecektir. Bütünlük kontrolleri koymak için MySQL tabanında şu komutu kullanabilirsiniz.

```
alter table car add FOREIGN KEY (garage_id) REFERENCES garage(id);
```

Diğer tabanlarda sözdizim farklı olabilir, fakat işi özünde bilmeniz gereken, kaynak ve hedef tablo+kolon ikilisini bilmektir. Yukarıdaki bütünlük kontrolü komutu işletildikten sonra, artık `car.garage_id` kolonuna koyulacak her verinin `garage.id` kolonunda olması mecbur olacaktır.

Birleştirim

İki tablo arasında ilişkiyi kurduktan sonra eğer bir tablodaki satırdan, o satırın ilişkisini olduğu diğer tablodaki satıra atlamak istersek, ilişkisel modelde birleştirim (join) kavramını kullanmamız gerekir. Birleştirim, küme teorisinden gelen bir kavramdır, ve SQL'de şu şekilde gösterilir.

```
SELECT * FROM GARAGE, CAR;
```

Bu komut, her iki tablodaki *her* satırı diğer tablodaki diğer *her* satır ile teker teker kombinasyona sokarak, bir kartezyen kombinasyonu olarak geri getirir. Birleştirime sokulan her tablo, `FROM` ibaresinden sonra birbirinden virgülle ayrılarak belirtilmelidir. Eğer `CAR` tablosunda 5, `GARAGE` tablosunda 2 satır var ise, kartezyen birleşim 10 tane satır geri getirecektir. Kartezyen birleşim sonucunu Tablo 9.5 üzerinde görüyoruz.

Fakat bu birleşim, hiçbir pratik uygulama için faydalı olmayacaktır. Geri gelen sonuç listesini bir şekilde daraltmamız gerekmektedir. Çoğunlukla ihtiyacımız olan, başlangıç tablolarındaki bir tablodan (ve onun bir satırından) bir diğerine (onun bir satırına) atlayabilmektir. O zaman, kartezyen sonuçta *yabancı anahtarı* ve asal anahtarı birbirine uyan satırları bu büyük birleşimden çekip çıkartmayı deneyebiliriz. Çünkü, gördüğümüz gibi, artık yabancı anahtar ve tekil anahtarlar, aynı büyük tablo içinde yanyana gelmişlerdir, ve basit bir `WHERE` şartı, birbirine uyan satırları çekip çıkarmak için yeterli olacaktır.

```
SELECT *
```

Tablo 9.5: Kartezyen Birleşimi

ID	DESCRIPTION	LICENSE_PLATE	DESCRIPTION	GARAGE_ID
1	Garage 1	34 TTD 2202	sarı araba	1
2	Garage 2	34 TTD 2202	sarı araba	1
1	Garage 1	34 GD 22	kırmızı araba	1
2	Garage 2	34 GD 22	kırmızı araba	1
1	Garage 1	35 TG 54	benim tarifim	2
2	Garage 2	35 TG 54	benim tarifim	2
1	Garage 1	16 RR 344	bu araba ikinci el	2
2	Garage 2	16 RR 344	bu araba ikinci el	2
1	Garage 1	33 RE 43	sarı araba	2
2	Garage 2	33 RE 43	sarı araba	2

Tablo 9.6: Filtrelenmiş Kartezyen Birleşimi

ID	DESCRIPTION	LICENSE_PLATE	DESCRIPTION	GARAGE_ID
1	Garage 1	34 TTD 2202	sarı araba	1
1	Garage 1	34 GD 22	kırmızı araba	1
2	Garage 2	35 TG 54	benim tarifim	2
2	Garage 2	16 RR 344	bu araba ikinci el	2
2	Garage 2	33 RE 43	sarı araba	2

```
FROM GARAGE, CAR
WHERE ID = GARAGE_ID;
```

Sonucu Tablo 9.6 üzerinde görüyoruz. Filtrelenmiş kartezyen kombinasyonundan gelen bu şekilde sonuçlar, daha pratik uygulaması olabilecek türden sonuçlardır. Garajlar ve onların altında olan arabalar, artık aynı satırda yanyana hâlde gösterilmiştir. Eğer daha da detaylı bir sonuç görmek istersek, meselâ

Tablo 9.7: Tek Garajın Altındaki Arabalar

ID	DESCRIPTION	LICENSE_PLATE	DESCRIPTION	GARAGE_ID
1	Garage 1	34 TTD 2202	sarı araba	1
1	Garage 1	34 GD 22	kırmızı araba	1

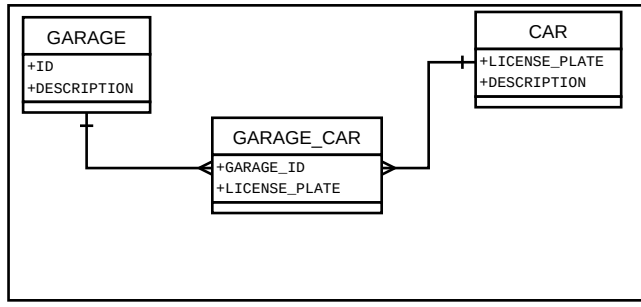
sadece 1 no'lu garaj altındaki tüm arabaları görmek istersek, üstteki sorguyu biraz daha genişleterek şöyle bir SQL kullanabiliriz.

```
SELECT *
FROM GARAGE, CAR
WHERE ID = GARAGE_ID
AND GARAGE_ID = 1
```

Bu sorgunun sonucu Tablo 9.7 üzerinde sergilenmiştir. Artık sadece 2 no'lu ve altındaki arabaları listemizde görmekteyiz.

Eğer geri getirilen kolonları (satırları değil) filtrelemek istiyorsak, `SELECT` komutundan sonra belirtilen kolon listesi ile bu isteğimizi belirtebiliriz. Örneğimizde, `*` işaretini kullanarak tüm kolonları almak istediğimizi belirttik, fakat `SELECT LICENSE_PLATE` ile sadece plaka no'larını almamız mümkün olabilirdi.

Çoka çok türden ilişkiler kurmak için iki tablo arasında bir ara tablo kullanmayı seçebilirsiniz. Bu durumda ilişki, Şekil 9.2 üzerindeki gibi gözükecektir.



Şekil 9.2: Ara Tablo ile İlişki

Ara tabloyu birleştirime dahil etmek için, `FROM`'dan sonra gelen listeye, ara tabloyu da eklemeniz gerekecektir. Bunun yapmanın genel mantığı, iki tablolu kullanım ile aynıdır.

```
SELECT *
FROM GARAGE, CAR, GARAGE_CAR
WHERE ID = GARAGE_ID AND
```

```
CAR.LICENSE_PLATE = GARAGE_CAR.LICENSE_PLATE AND
GARAGE_ID = 1;
```

Bir Tabloda Olan, Ötekinde Olmayan Satırlar

İki tabloda birbirine uyan satırları birleştirim (join) ile getirmek mümkündür. Eğer bir tabloda olan ama bir diğerinden *olmayan* satırları geri getirmek istersek, SQL'in `WHERE NOT EXISTS` komutunu kullanmamız gerekecektir. `NOT EXISTS`, her zaman bir alt sorgu (subsquery) içinde kullanılır. Alt sorgular, SQL sorgularınızı daha detaylandırmak için kullanılan tekniklerden biridir; Bu teknik ile ana sorgudan sonra parantezler içinde üst sorgudaki küme içinden daha detaylı seçimler yapmak mümkündür.

Alt sorguları, çoğu durumda, bir birleştirim formuna çevirmek mümkündür. Fakat sektörümüzde genel kullanım birleştirim ile çözülebilecek problemlerin direk birleştirim kullanılarak çözülmesi yönünde olduğu için, alt sorgular birleştirim sözdizimi kadar rağbet görmemiştir; `NOT EXISTS` kullanımı bunun haricindedir, çünkü standart SQL bağlamında "bir satırda olan ama ötekinde olmayan sonuçları bulma" sorgusunu daha temiz bir şekilde gerçekleştirmenin yolu alt sorgular üzerinden `NOT EXISTS` komutunu kullanmaktan geçer.

`NOT EXISTS` kavramını anlatmak için bir örnek geliştirelim: Bu örnekte iki tablo yaratalım, `passengers` (yolcular) `cars` (arabalar) [3]. Bu örnekte yolcular bir treni oluşturan kompartmanlarda oturmaktadırlar. Her yolcu, `passengers` tablosunda ismiyle kimliklendirilecektir, ve kompartmanlar `compartment` kolonu ile `cars` tablosu üzerinde tutulacaktır.

```
CREATE TABLE passengers (
   name VARCHAR(15),
   compartment INT);

INSERT INTO passengers VALUES ('smith',20);
INSERT INTO passengers VALUES ('jones',25);

CREATE TABLE cars (
   compartment INT,
   class VARCHAR(10));

INSERT INTO cars VALUES (20,'compartmant 1');
```

Bu şemaya ve örnek veriye göre, `smith` adlı yolcu `20` no'lu ve `cars` *tablosunda mevcut* bir kompartmanda oturmaktadır. Bizim sorguyla bulmak istediğimiz `jones` adlı yolcudur, çünkü bu yolcu `25` no'lu ve `cars` üzerinde *olmayan* bir kompartmanda oturmaktadır.

[3] http://dev.mysql.com/tech-resources/articles/4.1/subqueries.html

> **Not:** Sadece örnek olarak verilmiş bu şemanın gerçek dünya şartlarına pek uyduğu söylenemez, çünkü bu tür bir uygulamalarda genellikle `cars`'ın asal anahtarı `compartmant` ve `passengers`'daki yabancı anahtar `compartmant` arasında bir bütünlük kontrolü (9.1.1) konacak, ve `passengers`'a yapılan her `INSERT` kontrol edilerek `INSERT INTO passengers VALUES ('jones',25)` komutu bu kontrolden geçmeyecekti. Sadece örnek amaçlı olarak bu şemayı ve veriyi kullanıyoruz, `NOT EXISTS` genellikle arasında yabancı/asal anahtar ilişkisi olmayan tablolar için kullanılır.

`NOT EXISTS` içeren alt sorgu kullanan örneğimizdeki SQL komutu, şöyle olacaktır.

```
SELECT * FROM passengers
WHERE NOT EXISTS
   (SELECT * FROM cars
    WHERE cars.compartment = passengers.compartment);
```

Bu sorguda üst sorgu, `passengers` tablosundaki verilerden sorgulamaya başlamıştır. Üst sorgudan sonra parantez içinde bir sorgu daha görüyoruz, demek ki alt sorgu mevcuttur. Alt sorguların işleyişini hayal etmek için şöyle düşünmek faydalıdır; Sanki üst sorgudaki her satırın teker teker alt sorguya verildiğini düşünün. Eğer üst sorguda `NOT EXISTS` kullanımı mevcut ise, ve alt alt sorgudan, üst sorgudan verilen artı kendi yaptığı bazı filtrelemeler sonucunda "hiç bir sonuç" geri gelmez ise, üst sorgudaki `NOT EXISTS` şartına uyulmuş olacağı için, üstten alta verilmiş olan satır, başarılı bir satır olarak geri döndürülecektir.

Alt sorguda `cars.compartment` kullanımına dikkat çekmek isterim; Yâni üst sorgudaki tabloya alt sorgudan erişmek mümkündür (ama tersi mümkün değildir). Ayrıca bu durum önceki paragrafta bahsettiğimiz üstten alta teker teker gönderilen satırlar betimlemesine de uygundur.

Bu örnek sorgudan cevap olarak tek bir `passengers` satırı geri gelecektir; O da `25` no'lu kompartmanda oturan `jones` adlı kişidir.

9.1.2 Veri Modelini Normâlleştirmek

Normâlleştirme (normalization), ilişkisel bir veri tabanı modelinin daha arı hale getirilmesidir. Normâlleştirme, aynı zamanda "ilk taslak" veri tabanı tasarımının üzerinde revizyonlar yapmanının yolu, taslağı son haline yaklaştırmanın yöntemlerden birisidir. Normâlleştirmenin altyapısı matematikseldir, aynen ilişkisel modelin kendisinin matematiksel olması gibi. Temel alınan kavram, işlevsel bağlılık (functional dependency) denen bir kavramdır. Normalleştirmenin amaçları şunlardır.

Veri Bütünlüğü: Sadece bu öğe bile normalleştirme ile uğraşmak için yeterli bir sebeptir. Veri, bütünlüğü bozulmamış bir şekilde kalır, çünkü her çeşit bilgi, sadece bir kere tek bir yerde saklanır. Yani, bir veri çeşidinin kopyasının

veri tabanının değişik çizelgeleri üzerinde saklanması gerekmez. Eğer veri gereksiz şekilde kopyalanmış ise, bu değişik kopyalar, kopyalardan habersiz olan uygulama kodları yüzünden bir süre sonra birbirinden farklı değerleri taşımaya başlayabilirler. Bu, doğruluk ve tutarlılık açısından çok kötü bir sonuçtur. Bu gibi durumlarda ilişkisel veri taban paketinizin otomatik bütünlük kontrol (automatic integrity check) mekanizmaları bile işe yaramaz. Tedavinin, uygulama seviyesinde yapılması gerekir. Fakat bu da uygulama programlarını daha kompleks hâle getirecek, dolayısıyla bakımını zorlaştıracaktır.

Uygulamadan Bağımsız Veri Modeli: Normalleştirme, genelde bilinen ve takip edilen "ilişkisel model, verinin içeriğine göre kurulmalı, uygulamaya göre değil" kavramını bir adım daha öne alır. Bu sayede veri modeli, üzerinde onu kullanan uygulama değişse bile daha tutarlı, sabit ve değişmez olarak kalacaktır. Uygulama programının gereksinimlerinin veri tabanı mantıksal (logical) model üzerindeki etkisi sıfır olmalıdır. Daha ileride göreceğimiz gibi, uygulama, mantıksal model üzerinde değil, fiziksel (physical) model üzerinde etki yapar.

Depolama Yeri Azaltımı: Yabancı/göstergeç anahtarların haricinde, tamamiyle normâlleştirilmiş bir veri tabanı gereksiz (kopyalanmış) veri miktarını en aza indirecektir. Kopyalanma miktarı azaldığı için, depo yerine olan ihtiyaçta azalır. Ve gene bu sayede, veri tabanı motorunun arama yapması da daha rahatlaşacaktır.

Birinci Normal Formu

Bu form, tekrar eden hiç bir gurup taşımaz. Bir başka deyişle, bir hücre üzerinde taşınan değer tek ve basit olmalıdır. Aşağıdaki örnek veride, son sene nüfusu %5'den fazla artmış olan şehirlerden bazılarını görüyoruz. Şehir bilgilerinin bazılarının aynı hücre içerisinde guruplandığını görüyoruz. Bu yüzden bu tablo, normal değildir (hâtta 1NF bile değildir).

Bu tablodaki bilgiye bakarak, şehir bilgilerinden mesela `BU_YILIN_NFUSU` kolonundaki nüfuslardan hangisinin hangi şehre ait olduğunu nereden bilebiliriz? Bir kolon içinde bile birçok nüfus ve birçok şehir var. Sorumluluğu veri modelinden uygulama programına atarak, sıraya göre bir eşleme düşünülebilir, fakat bu da en temel ilişkisel veri tabanı kuralı olan 'kolon içinde sıra olmaz' kuralının ihlalidir.

Bu veri yapısını 1NF (birinci normal formu) hâline getirmek için, tekrar eden gurupları (verileri) tek kolondan çıkarıp, değişik satırlara yaymak gerekir. Alttaki tabloda 9.9 üzerinde, tablo 9.8 verisinin 1NF'e getirilmiş halini görebilirsiniz.

Bu yeni veri modelinde, eyalet kolonu asal anahtar (primary key) olarak addedildi. Fakat bu son tabloda da bazı problemler var. Güncellemek ve silmek için, hala birçok veriye teker teker uğrayıp, verinin bütünlüğünü kod yazarak birarada tutmamız gerekiyor. Mesela eğer yeni bir şehir satırı ekleyecek olsak, beraberinde eyalet verisi de eklememiz lazım. Ya da bir eyaletin son şehrini veri tabanından silsek, bu eyaletin verisini veri tabanından tamamen kaybetmiş oluyoruz. Demek ki 1NF'ten daha optimal bir yapıya geçmemiz gerekiyor.

Tablo 9.8: Normal Olmayan Tablo

EYALET	KISALTMA	EYALET NÜFUSU	ŞEHİR	GEÇEN YILIN NÜFUSU	BU YIL NÜFUS	YÜZDE ARTIŞ
North Carolina	NC	5M	Burlington, Raligh	40k, 200k	44k, 222k	10%, 11%
Vermont	VT	4M	Burlington	60k	67.2k	12%
New York	NY	17M	Albany, New York City, White Plains	500k, 14M,		
100k	570k, 14.7M, 106k	8%, 5%, 6%				

Tablo 9.9: 1NF

EYALET	KISALTMA	EYALET NÜFUSU	ŞEHİR	GEÇEN YIL NÜFUS	BU YIL NÜFUS	YÜZDE ARTIŞ
North Coralina	NC	5M	Burlington	40k	44k	10%
North Coralina	NC	5M	Raleigh	200k	222k	11%
Vermont	VT	4M	Burlington	60k	67.2k	12%
New York	NY	17M	Albany	500k	540k	8%
New York	NY	17M	New York City	14M	14.7M	5%
New York	NY	17M	White Plains	100k	106k	6%

İkinci Normal Formu

2NF yapısında 1NF örneğinde gördüğümüz şekilde yarım bağımlılık bulunmaz. Anahtar olmayan her kolon, asal anahtara tamamen bağlı olmalıdır. Anahtar kolon, birden fazla kolonu kapsıyorsa, anahtar olmayan kolonlar anahtar kolonların hepsine tamamen bağlı olmalıdır. Tablo 9.9 bu kritere hâla uymuyor. Şehir bilgileri eyalet bilgisine bağlı değil. Daha detaylı olarak, bütün şehir kolonları (`SEHIR`, `GECEN_YIL_NUFUS`, `BU_YIL_NUFUS`, `YUZDE_ARTIS`) asal anahtar eyalet kolonuna `EYALET` tamamen bağlı durumda değildir.

Bu yüzden, birbirine tam bağlı olmayan bilgileri ayrı tablolara parçalamamız gerekiyor. Tablo 9.10 ve 9.11 üzerinde 2NF şemanın son hâlini görüyoruz.

Tablo 9.10: 2NF Eyalet Tablosu

EYALET	KISALTMA	EYALET_NÜFUSU
North Carolina	NC	5M
Vermont	VT	4M
New York	NY	17M

Tablo 9.11: 2NF Şehir Tablosu

EYALET	KISALTMA	ŞEHİR	GEÇ YIL NÜFUS	BU YIL NÜFUS	YÜZDE ARTIŞ
North Coralina	NC	Burlington	40k	44k	10%
North Coralina	NC	Raleigh	200k	222k	11%
Vermont	VT	Burlington	60k	67.2k	12%
New York	NY	Albany	500k	540k	8%
New York	NY	New York City	14M	14.7M	5%
New York	NY	White Plains	100k	106k	6%

Üçüncü Normal Formu

3NF veri modelinde, anahtar olmayan hiçbir kolon, başka anahtar olmayan kolona bağlı olamaz. 2NF'de bütün kolonların asal anahtara bağlı olduğunu

söylemiştik. 3NF'e göre, bu bağlantı **dolaylı bile olamaz**. Mesela, `YUZDE_ARTIS` kolonuna bakalım. Bu kolon, `GECEN_YILIN_NUFUSU` ve `BU_YILIN_NUFUSU` kolonlarına bağlıdır, çünkü bu iki kolondaki değerlerden **hesaplanır** (Hesaplanan kolonlara literatürde türetilen (derived) kolon ismi de verilmektedir). Üretilen kolonlar asal anahtara bağlılardır ama, bu bağlılık dolaylıdır, yani bağlı oldukları esas iki kolon asal anahtara bağlı olduğu için onlar da asal anahtara bağlılardır.

Üçüncü normal formunda böyle kolonlara izin verilmez. `YUZDE_ARTIS` kolonu 3NF'de şemadan atılması gerekmektedir. Türetilen değerler, uygulama programı tarafından anlık hesaplanacaktır. Eğer hesaplanan değer çok sık erişiliyor ise, o zaman Oracle görüntü kavramı kullanılarak bu hesabın hayali bir tablo çerçevesinde servis edilmesi performans açısından yararlı olabilir.

Veri tabanının 3NF (en son) halini Tablo 9.12, 9.13 ve 9.14 üzerinde görüyoruz.

Tablo 9.12: 3NF

EYALET	EYALET NÜFUS
North Carolina	5M
Vermont	4M
New York	17M

Tablo 9.13: 3NF

EYALET	KISALTMA
North Carolina	NC
Vermont	VT
New York	NY

9.2 Yardımcı Kavramlar

Tablo, SQL gibi ana kavramların üstüne, çoğu veri taban ürünü bazı ek servisleri kullanıcılarına sunmaktadır. Bu servisler, bir tablo üzerinde yapılan silme, güncelleme, ekleme gibi bir işlem olduğu *anda* bir ek komutun işletilmesini sağlayan tetik (trigger) kavramı olabilir. Tetik, normâl proglamlama dillerinden bize tanıdık gelecek bir kavramdır, bir çengel (hook) olarak görülebilir, ve çengel, takıldığı yer aktif olunca kendisi de aktif hâle gelecektir. Diğer kavramlardan görüntü (view), dizi (sequence) ve eşanlam (synonym) sayılabilir.

Tablo 9.14: 3NF

ŞEHİR	KISALTMA	GEÇEN YIL NÜFUS	BU YIL NÜFUS	YÜZDE ARTIŞ
Burlington	NC	40k	44k	10%
Raleigh	NC	200k	222k	11%
Burlington	VT	60k	67.2k	12%
Albany	NY	500k	540k	8%
New York City	NY	14M	14.7M	5%
White Plains	NY	100k	106k	6%

Bu kavramları şimdiye kadar Oracle üzerinde kullandığımız için, bu bölümde de Oracle üzerinden işleyeceğiz. Fakat kendi ürününüz üzerinde bu kavramların karşılıklarını bulabilirsiniz (özellikle PostgreSQL üzerinde, çünkü bu taban özellik olarak Oracle'a en yakın olan açık yazılım ürünüdür).

SID

Veri tabanı kelimesini kullandığımızda genelde birçok şeyden aynı anda bahsediyoruz. Oracle paket programının tamamına, içinde bilgi depolayan kayıtları, kullanıcı isimlerinin toplamina aynı anda veri tabanı deniyor. Fakat, Oracle dunyasinda, bu kavramlari daha da berraklaştırmamız gerekecek. SID= Veri tabanı diyeceğiz, ve, veri tabanı Oracle paket programı değildir gibi bir tanım yapacağız. Peki o zaman, veri tabanı nedir?

Oracle'a göre veri tabanı, hakkında konuşabileceğimiz en büyük kayıt birimidir. İçinde tablolar, onların yazıldığı dosyalar, bu tablolara erişecek kullanıcı isimleri, ve paraloların toplamına veri tabanı diyoruz. Bir proje icinde şu kelimeleri duyabilirsiniz.

- Hangi veri tabanına bağlandın, tablo x'i bu tabanda bulamıyorum...
- (Admin'e) Benim kullanıcı ismimi bu veri tabanında da yaratır mısınız ? Kullanıcı şifrem kabul edilmiyor
- Fazla veri tabanı yaratmaya gerek yok, bir tane üzerinde çalışsak olmaz mı?
- Hayır olmaz, çünkü veri tabanı idarecileri iki veri tabanı olursa idaresi kolaylaşır diyorlar.

SID, veri tabanına erişmemizi saylayacak bir isimden ibarettir. Örnek olarak SATIS, BORDRO, MUSTERI gibi veri tabanı isimleri olabilir.

9.2.1 Tablo Alanı

Tablo alanı (tablespace), tabloların üzerinde depolanacağı dosya ismidir, yâni `/usr/dizin1/dizin2/cizelge_alan_1.dbf` gibi bir gerçek Unix dosyasından bahsediyoruz. Oracle veri tabanının, gerçek dünya ile (işletim sistemi) buluştuğu yere tablo alanıdır.Tablo ile tablo alanlarının bağlantısı, alan yaratılırken sâdece bir kere yapılır. Ondan sonra ne zaman bu tabloya erişseniz, önceden tanımlanmış olan dosyadan oku/yaz otomatik olarak yapılacaktır.

Tablo alanı yaratmak için, şöyle bir komut işletebiliriz.

```
CREATE TABLESPACE cizelge_alan_1 DATAFILE
'/usr/local/dizin1/oracle/cizelge_alan_1.dbf' SIZE 200M;
```

9.2.2 Şema

Şema=kullanıcı gibi bir tanım yapabiliriz, fakat Oracle dünyasında şema biraz daha güçlü bir kavramdır. Eğer daha basit veri tabanlarına alışıksak, herhalde her kullanıcının her tabloyu görmesine alışığızdır. Fakat Oracle için tabloların erişilip erişilmeyeceği, tabloların nerede tutulacağı (yâni sahiplenme mekanizması) şema bazında idare edilmektedir. Bir veri tabanına (`SID=ORNEK1`) bağlandığınız zaman, bu tabanda bazı tabloları göremeyebilirsiniz. Görmek için, belli bir kullanıcı ve şema kullanarak bağlanmanız gerekecektir. Aynı isimdeki iki tabloyu değişik şemalarda yaratabiliriz; Oracle bundan yakınmaz. Yâni SID'den sonra Oracle'ı paylaştırmanın/bölmenin ikinci bir yolu şemadır.

Oracle idarecileri az miktarda SID ve paylaştırmak için çok miktarda şema yaratmayı tercih ederler. Şema yaratmak için kullanıcı yaratmamız yeterlidir.

```
CREATE USER hasan IDENTIFIED BY hasanslx DEFAULT TABLESPACE alan1;
```

Bu konu hakkında bazı notlar şunlardır:

- Kullanıcı MEHMET olarak SID'e bağlandıysanız, ve CREATE TABLE komutunu işletip bir tablo yarattıysanız, bu tablo MEHMET şemasına ait olacaktır. Bu tabloyu başkaları göremez. Görmesi için özel izin 'vermeniz' gerekir.

- Eğer MEHMET kullanıcısı olarak bir tablo yarattıysanız ve detay belirtmediyseniz, bu tablo DEFAULT TABLESPACE diye yukarıda belirttiğimiz yokluk değeri (default) `alan1` altında yaratılır.

9.2.3 Görüntü

Görüntüler, önceden depolanmış ve sorgulanabilen SQL kodlarıdır. Bir görüntü, güvenlik ötürü bazı çizgeleri göstermeden, bu çizgelerin verisinin bir kısmını göstermek için kullanılabilir. Mesela halkla ilişkiler bölümü için, isim, soyad ve adres bir görüntü içinden gösterilebilir, öteki bilgiler saklanabilir. Ya da, dağınık

veri tabanlarından toparlanacak bilgiler, bir görüntü ile önceden kodlanır ise, artık birden fazla veri tabanına bağlanmak yerine, tek bir görüntü üzerinden veri alınabilir. Özet olarak, normalde işleyen her SQL kodu, görüntü yaramak için kullanılabilir.

```
create or replace view goruntu_1 as
select isim, soyad
from MUSTERI;
```

9.2.4 Dizi (Sequence)

Tekil sayı yaratmak için kullanılan Oracle nesnesine dizi adı verilir. Dizi yaratmak için başlangıç değeri, artış miktarı ve bitiş sayısı vermek yeterlidir. Yani, 1..n arası her seferinde 1 kadar artacak bir dizi yaratmak mümkün. Bu sayılar genelde kimlik no gibi özgün olması gereken değerler için kullanılır. Her seferinde kod içinde, 'önceki değer + 1' demek yerine, SQL kullanarak her diziden yeni sayı isteyişimizde, güncel dizi artış değeri kadar otomatik olarak arttırılır ve alınan sayı SQL sonucu olarak verilir.

Bir dizi yarattığınız zaman, `NEXTVAL` ve `CURRVAL` sanal kolonlarını kullanırız. Mesela `UYE_DIZISI` adlı bir dizi yarattı isek, `SELECT UYE_DIZI.NEXTVAL FROM DUAL` diyerek o anki değeri alabiliriz. Bu komut sonucunda ayrıca dizi değeri 1 arttırılır. `SELECT UYE_DIZI.CURRVAL FROM DUAL` kullanımı o anki değeri verecektir. Fakat diziyi arttırmaz.

```
CREATE SEQUENCE DIZI_ISMI
INCREMENT BY 1 -- artış
START WITH 100; -- başlangıç
```

9.2.5 Tetik

Tetikler (trigger), depolanmış PL/SQL kodlarıdır. Önceden öngörülen belli bazı tablolara, gene önceden öngörülen şekilde bir erişim olduğunda tetikler ateşlenirler, ve kodlanan işlemi yerine getirirler. Tetikler, satır ekleme, silme, güncelleme ya da bütün bu temel işlemlerin değişik guruplamaları için kodlanabilirler. Tetiklerin en çok kullanıldığı alan, veri bütünlüğü için kısıtlamalar koymaktır. Bu tür kısıtlamalar Oracle terimleri ile, veri bütünlük kontrolleri (integrity constraints) olarak bilinir, ve satırlar arasında bazı kontroller getirebilir. Fakat bu tür satır bazlı kontrollerin yetmediği hallerde, tetik kodları işe yarayabilir. Çünkü tetik kodları içine PL/SQL ile kodlayabildiğiniz her türlü kod koyabilirsiniz.

```
create to replace trigger musteri_tetik_1 INSTEAD OF
insert on musteri for each row
begin
insert into maas values
('1', '2'); -- buraya daha değişken kod koyabilirsiniz
```

```
end;
/
```

9.2.6 Dizi ve Tetik

Bazı veri şemalarında ID kolonu atanan bir değer değil, üretilmesi gereken bir sayıdır. Bu sayı, en basit hâliyle 1'den başlayarak her yeni veri satırı için birer farkla artması gereken bir sayıdır.

O zaman her satırı yazarken bu ID'nin üretilmesi gerekmektedir. Bu üretimi, ya uygulama içinde yapacağız, ya da veri tabanın otomatik olarak yapmasını sağlayacağız.

Otomatik attırım bazı veri tabanlarında bir kolon tipi olarak bile karşınıza çıkabilir. Oracle'da bu işi yapmak için bir sequence ve bir trigger kullanmamız gerekiyor. Oracle, bu tekil sayının üretimini ve tabloya atanması işini birbirinden ayırmıştır. Bu da sanıyorum isabet olmuştur, auto-increment kolon tipi hakikaten çok basitleyici bir çözümdür.

Oracle'da otomatik ID üretimi için iki şey gerekir; Bir sequence, bir de trigger. Meselâ, şöyle bir tablomuz olduğunu düşünelim.

```
create table test (
id number,
veri varchar2(20),
...
);
```

Bu tablo için bir sequence ve bir trigger yaratalım.

```
create sequence test_seq
start with 1
increment by 1
nomaxvalue
;

create trigger test_trigger
before insert on test
for each row
begin
select test_seq.nextval into :new.id from dual;
end;


INSERT into test (veri) values ('blablabla');
```

gibi bir `INSERT` kullandığımızda, hiç ID'ye dokunmamıza gerek kalmadan, bir sonraki ID sayısı hesaplanacak ve kolona koyulacaktır.

9.2.7 Veri Taban Köprüsü

Köprüler (database link) nesne (tablo, kolon, vs) bazında değil, tüm veri tabanı bazında bağlantı kurar. Yani, uzakta olan bir veri tabanına, sürekli olarak ve uzak adresini kullanarak erişmek istemiyorsanız, bir köprü, yani kestirme yaratarak o veri tabanına sanki yerel bir tabanmış gibi erişebilirsiniz.

```
CREATE PUBLIC DATABASE LINK baglanti_1
CONNECT TO kullanici
IDENTIFIED BY sifre
USING 'UZAK_VERI_TABAN_SID_DEGERI';
```

9.2.8 Eşanlam (Synonym)

Eşanlamlar, bir veri tabanındaki tablolarden diğer tablolara kestirme işaret olarak görülebilir. Bir eşanlam yaratmak için eşanlam ismi, ve o eşanlamın yerini tuttuğu Oracle nesnesinin ismi gereklidir. SQLPlus kullanarak bu eşanlama eriştiğimizde, Oracle arka planda eşanlamın yerini tuttuğu nesneyi bulur, ve işlemi o öteki nesne üzerinde yapar.

İki türlü eşanlam vardır. Umumi (public) ve özel (private). Özel eşanlamlar içinde yaratıldıkları şemaya özel olurlar, o şemaya ait kalırlar ve başka kullanıcılar tarafından erişilemez, hatta görülmezler bile. Eğer eşanlamlar umumi ise, bütün şematikler tarafından kullanılabilirler.

Oracle'ın bir nesneye erişmek için hangi tür bir algoritma izlediğine gelelim. Eğer şöyle bir kod işletildi ise,

```
SELECT * FROM FROM MAAS
```

Oracle MAAS nesnesini bulmak için şunları yapar.

- MAAS adlı bir tablo ya da görüntü var mı?
- Bu ikisi yoksa, Oracle MAAS adında özel bir eşanlam arar.
- Var ise, özel eşanlamın gösterdiği nesne kullanılır.
- Yok ise, MAAS adlı umumi eşanlam aranır.
- Hiçbiri yok ise, Oracle ORA-00942 hata mesajını verecektir.

```
create synonym MUSTERI for BIZIM_UZUN_MUSTERI_ISMI;
```

9.2.9 İndeksler

Bir tablodan veri almak için, kullandığımız `SELECT` komutunun `WHERE` kısmı içinde filtre şartları belirtmemiz gerekir. Bu şartlar, veri tabanı tablosu taranırken gereken satırları diğerlerinden ayırıp onları çekip çıkarabilmemiz için kullanılır. Arama şartları tanımlandıktan sonra, veri tabanı kapalı perdeler arkasında tarama işlemini gerçekleştirmek için şunları yapar: Tabloyu tararken, ilk satırdan başlayıp teker teker sonuncu satıra doğru tüm satırlara bakar, ve filtre şartlarının uyup uymadığını kontrol eder.

Fakat tüm satırlara bakılan türden bir arama, özellikle büyük tablolar için çok uzun zaman alacaktır. Eğer tüm satırlara bakılan türden bir tarama yapılmasını istemiyorsak, veri tabanına arama yaparken bir şekilde *yardım etmenin* yolunu bulmalıyız.

İndeksler bu türden yardım yöntemidirler. İndeksler, kendileri veri içermezler, sadece gerçek veriyi içeren tablonun belli satırlarına *işaretler* taşırlar. Çok hızlı aranabilen türden veri yapılarıdırlar. Bu işaretler, belli anahtar kolonlarına göre (indeksle) yapılmıştır. İndeksler bu açıdan bir fihriste benzerler.

Bir indeksin kullanıma konması şöyle olur: Eğer `SELECT` filtremiz içinde kullanılan kolonlar üzerinde bir indeks mevcutsa, veri tabanı filtreyi önce indeksden tarar, buradan gelen satır işaretini kullanarak ta gerçek tabloya giderek aranan gerçek satırı döndürür.

İndeksler gerçek tabloyu indeksledikleri için, bu tabloda olan her değişim indeksleri etkiler. Meselâ tabloya yeni bir satır eklenirse, indeks bu ekleme işlemini yansıtacak şekilde güncellenir. Bu sebeple gereğinden fazla indeks eklenmesi bir tabloya yapılan `INSERT` işlemlerini yavaşlatacaktır. Tam kararında kurulan indeksler, hem `SELECT` sorgularımızı hızlandırır, hem de `INSERT`'lerimiz üzerinde fazla bir etkide bulunmazlar.

Şu anda piyasada olan her profosyonel veri taban ürünü, indeks teknolojisini desteklemektedir. Bu bölümün geri kalanında PostgreSQL, Oracle ve MySQL tabanlarında indeks eklemenin yollarını, ve en önemlisi, herhangi bir sorgunun üzerinde analiz yaparak indeks kullanıp kullanmadığını görebilmek için gereken komutları öğreneceğiz.

Elimizde `car` ve `garage` adlı iki tablo olduğunu düşünün. Bu tablolar arasında bire çok türünden bir ilişki olsun ve bir `car` birçok garajın altında olabilsin. Bu ilişkiyi fiziksel anlamda, `car` tablosu üzerinde bir `garage_id` koyarak gerçekleştiriyoruz.

PostgreSQL

Bir sorgunun indeks kullanıp kullanmadığını analiz etmek için, o sorguyu `psql` ile PostgreSQL komut satırına girip, `EXPLAIN` komutu eşliğinde işletmemiz gerekmektedir.

```
explain
select * from car where license_plate = '34 THY 334';
```

Eğer hiçbir indeks tanımlamamışsak, şöyle bir cevap geriye gelecektir.

```
QUERY PLAN
-----------------------------------------------------
Seq Scan on car (cost=0.00..330.00 rows=44 width=170)
Filter: ((license_plate)::text = '34 THY 334'::text)
```

Bu çıktıda ilk dikkatimizi çeken kelimeler, `Seq Scan` kelimeleri olmalıdır. Bu kelimeler Sequential Scan (sırayla tarama, teker teker bakma) kelimesinin bir kısaltmasıdır, ve `SELECT` komutunun teker teker her satıra baktığı anlamına gelir. Bu performans açısından istediğimiz bir durum değildir. Biz indekslerin kullanılmasını istiyoruz. Ve gördük ki, hiçbir indeks kullanılmamıstır. Çünkü daha indeks tanımlamadık! Aynı şekilde,

```
explain
select c.*
from
car c, garage g
where
c.garage_id = g.id and
g.id = 2 and
c.available = 't';
```

sorgusunun analizi de şu cevabı getirecektir.

```
QUERY PLAN
------------------------------------------------------------------------
Nested Loop (cost=18.50..372.26 rows=88 width=170)
-> Seq Scan on car c (cost=0.00..352.00 rows=22 width=170)
Filter: ((available = true) AND (2::numeric = garage_id))
-> Materialize (cost=18.50..18.54 rows=4 width=23)
-> Seq Scan on garage g (cost=0.00..18.50 rows=4 width=23)
Filter: (id = 2::numeric)
```

Çok kötü. Bu sorguda da, hem iki tablo birleştirimi (join) hem de `available` kolonu üzerinde yaptığımız filtreleme üzerinde `seq scan` taraması yapıldığını görüyoruz. Satır sayısı oldukça fazla olan `car` ve `garage` tabloları kullanıyor olsaydık, bu son sorgunun da performanı çok kötü olacaktır.

O zaman indeksleri ekleyelim. PostgreSQL'da indeks şöyle eklenir:

```
CREATE UNIQUE INDEX car_license_plate_idx on car (license_plate);
CREATE UNIQUE INDEX garage_id_idx on garage (id);
CREATE INDEX car_available_idx on car (available);
CREATE INDEX car_garage_idx on car (garage_id);
```

`car` tablosunda `license_plate` kolonu üzerinden işleyen bir indeks ekledik. Aynı şekilde `garage.id` kolonuna, `car.available` kolonuna ve `car.garage_id` kolonuna indeksler ekledik. İndeksler eklendikten sonra, sırasıyla her iki sorguda yapılan `EXPLAIN` komutu, şu sonucu döndürecektir.

```
QUERY PLAN
------------------------------------------------------------------------
```

```
Index Scan using car_license_plate_idx on car (cost=0.00..6.00 rows=1
width=170)
Index Cond: ((license_plate)::text = '701790-171'::text)

QUERY PLAN
------------------------------------------------------------------------------
Nested Loop (cost=0.00..15.03 rows=2 width=170)
->  Seq Scan on garage g (cost=0.00..1.05 rows=1 width=23)
Filter: (id = 2::numeric)
->  Index Scan using car_garage_idx on car c (cost=0.00..13.96 rows=2
width=170)
Index Cond: (2::numeric = garage_id)
Filter: (available = true)
```

Sonuca bakarak görüyoruz ki, birinci sorguda `Seq Scan` ibaresinde kurtulduk. Artık bu sorguya erişim, `Index Scan` ile olmaktadır, yâni verdiğimiz sorgu `car` tablosuna yarattığımız indeks üzerinden erişilmektedir. Bir hız ilerlemesi kaydettik. İkinci sorguda, `available` kolonu üzerindeki sorgudaki filtre ibaresi de indeks kullanmaya başlamıştır.

Fakat PostgreSQL sorgu hızlandırıcısı (optimizer), `garage.id` üzerindeki indeksi devreye sokmamıştır, çünkü `garage` tablosunda çok az veri vardır (örnek verimiz böyle idi) ve bu şekilde az veri taşıyan tablolarda indeks kullanıp kullanmamak bir hız farkı getirmeyecektir.

Oracle

Oracle üzerinde indeks yaratmak için `CREATE INDEX`, sorguların indeks kullanımını analiz etmek için `EXPLAIN PLAN` komutları kullanılır. `EXPLAIN PLAN`, komutu işlettiğini şemada işleyebilmek için `PLAN_TABLE` adlı bir sistem tablosunu kullanır. Oracle 10g üzerinde bu tablo her veri tabanı içinde kurulmuş olur, ama eğer kullandığınız versiyonda mevcut değilse, tabloyu şu komut ile yaratabilirsiniz (`ORACLE_HOME`, Oracle paketinin kurulmuş olduğu dizini temsil eder).

```
\$ sqlplus user/pass@SID
@ORACLE_HOME/product/<versiyon>/<Db>/rdbms/admin/utlxplan.sql
```

`utlxplan.sql` dosyasında `plan_table` ve ilgili tüm tabloların yaratılması için gereken işlemler yapılmaktadır.

Analiz için şu komutu kullanabilirsiniz.

```
explain plan SET STATEMENT_ID='SORGU_1' FOR
select * from car where license_plate = '34 THY 334';

SELECT LPAD(' ',2*DEPTH) | | OPERATION | | ' ' | | OPTIONS | | ' '
| | OBJECT_NAME | | ' ' | | DECODE(ID, 0, 'COST= '| | POSITION)
"QUERY PLAN"
FROM PLAN_TABLE START WITH ID=0 AND STATEMENT_ID='SORGU_1'
```

```
CONNECT BY PRIOR ID=PARENT_ID;
```

Sonuç şöyle gelir:

```
QUERY PLAN
---------------------------------

SELECT STATEMENT COST= 3
TABLE ACCESS FULL CAR
```

Bu işletim planına bakarsak, `TABLE ACCESS FULL` kelimelerini hemen gözümüze çarpar. Bu kelimeler bize işletilen sorgunun hiçbir indeks kullanmadığını göstermektedir, ve Oracle tablodaki satırlara teker teker ve sırayla erişmeye çalışmaktadır. Aynı şekilde diğer sorgumuza bakarsak;

```
explain plan SET STATEMENT_ID='SORGU_1' FOR
select c.*
from
car c, garage g
where
c.garage_id = g.id and
g.id = 2 and
c.available = 't';

SELECT LPAD(' ',2*DEPTH) | | OPERATION | | ' ' | | OPTIONS | | ' '
| | OBJECT_NAME | | ' ' | | DECODE(ID, 0, 'COST= '| | POSITION)
"QUERY PLAN"
FROM PLAN_TABLE START WITH ID=0 AND STATEMENT_ID='SORGU_1'
CONNECT BY PRIOR ID=PARENT_ID;
```

şu sonucu görürüz

```
QUERY PLAN
---------------------------------

SELECT STATEMENT COST= 6
MERGE JOIN CARTESIAN
TABLE ACCESS FULL CAR
BUFFER SORT
TABLE ACCESS FULL GARAGE
```

Bu analiz sonucunda da bol bol `TABLE ACCESS FULL` görmekteyiz. Demek ki gerekli yerlere indeksler eklememiz gerekiyor.

```
CREATE UNIQUE INDEX car_license_plate_idx on car (license_plate);
CREATE UNIQUE INDEX garage_id_idx on garage (id);
CREATE INDEX car_available_idx on car (available);
CREATE INDEX car_garage_idx on car (garage_id);
```

Şimdi `EXPLAIN PLAN`'i tekrar işletirsek, şu sonuçları göreceğiz.

```
QUERY PLAN
----------------------------------------------

SELECT STATEMENT COST= 1
TABLE ACCESS BY INDEX ROWID CAR
INDEX UNIQUE SCAN CAR_LICENSE_PLATE_IDX

QUERY PLAN
-----------------------------------------------

SELECT STATEMENT COST= 0
NESTED LOOPS
INDEX UNIQUE SCAN GARAGE_ID_IDX
TABLE ACCESS BY INDEX ROWID CAR
BITMAP CONVERSION TO ROWIDS
BITMAP AND
BITMAP CONVERSION FROM ROWIDS
INDEX RANGE SCAN CAR_AVAILABLE_IDX
BITMAP CONVERSION FROM ROWIDS
INDEX RANGE SCAN CAR_GARAGE_IDX
```

Bu sonuçlar çok daha iyidir. Birinci sorgunun analizi `TABLE INDEX BY ROWID CAR` kelimesini taşıyor, bu demektir ki tabloya erişim artık bir indeks üzerinden gerçekleşmektedir. Kullanılan indeks `CAR_LICENSE_PLATE_IDX` isimli indekstir, bu da hemen altındaki satırda belirtilmiştir.

İkinci sorgunun sonucu biraz daha çetrefildir, fakat burada da görüldüğü gibi artık `TABLE ACCESS FULL` sözü kaybolmuştur, onun yerine indekslerin kullanılmaya başlandığı `INDEX UNIQUE SCAN`, `INDEX RANGE SCAN` ve `TABLE ACCESS BY ROWID` kelimelerinden belli olmaktadır. `INDEX UNIQUE` ile `INDEX RANGE` arasındaki fark, ilkinin tekil bir satıra işaret edebilmesi, ikincisinin tekrar eden anahtar değerlere işaret eden türden bir indeks olmasıdır.

Sonuç olarak Oracle dünyasında sorgu optimizasyonu ile amacımız, `EXPLAIN PLAN`'den gelen sonuçlarda görülebilecek `TABLE ACCESS FULL` ibaresinden kurtulmaktır.

MySQL

MySQL veri tabanında analiz için `explain` komutu kullanılır. Aynen PostgreSQL örneğinde olduğu gibi, analiz etmek istediğiniz sorgunun önüne `explain` ibaresini eklerseniz, MySQL analiz sonucunu gösterecektir. Örnek olarak `car` ve `garage` üzerindeki yapılan iki sorguyu analiz edelim:

```
explain select * from car where license_plate = '32 TF 22';
```

```
+-------+------+---------------+------+---------+------+------+-------------+
| table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+-------+------+---------------+------+---------+------+------+-------------+
| car   | ALL  | NULL          | NULL |    NULL | NULL |    5 | Using where |
+-------+------+---------------+------+---------+------+------+-------------+
```

```
explain
select c.*
from
car c, garage g
where
c.garage_id = g.id and
g.id = 2 and
c.available = 't';
```

```
+-------+------+---------------+------+---------+------+------+-------------+
| table | type | possible_keys | key  | key_len | ref  | rows | Extra       |
+-------+------+---------------+------+---------+------+------+-------------+
| c     | ALL  | NULL          | NULL |    NULL | NULL |   47 | Using where |
| g     | ALL  | NULL          | NULL |    NULL | NULL |    4 | Using where |
+-------+------+---------------+------+---------+------+------+-------------+
```

`explain` komutundan gelen üstteki sonuçlar, her tablo için bir satır olmak üzere listelenecektir. Her tabloya yapılan erişimin yöntemini, `type` altında bulabilirsiniz. MySQL dünyasında, sorgu analizinde bir tablo için `type` için `ALL` görmek *iyi değildir*. `type` için gelmesi muhtemel değerleri en iyiden en kötüye olacak şekilde bir sıralamasını aşağıda görüyoruz.

1. `system`
2. `const`
3. `eq_ref`
4. `ref`
5. `ref_or_null`
6. `index_merge`
7. `ununique_subquery`
8. `range`
9. `index`
10. `ALL`

`ALL` değeri en alttadır, yâni yukarıdaki sorgularımızın eriştiği tablolara indeks eklemezsek, yukarıdaki sorgular en kötü performans ile işletiliyor olacaktırlar. O zaman bu durumu düzeltmek için indekslerimizi ekleyelim:

```
CREATE UNIQUE INDEX garage_id_idx on garage (id);
CREATE INDEX car_available_idx on car (available);
CREATE INDEX car_garage_idx on car (garage_id);
CREATE UNIQUE INDEX car_license_plate_idx on car (license_plate);
alter table garage add PRIMARY KEY(id);
alter table car add PRIMARY KEY(license_plate);
alter table car add FOREIGN KEY (garage_id) REFERENCES garage(id);
```

Tekrar `explain` eşliğinde iki sorguyu işletelim ve sonuçları görelim.

```
+-------+------+----------------------+---------+---------+-------+------+-------+
| table | type | possible_keys        | key     | key_len | ref   | rows | Extra |
+-------+------+----------------------+---------+---------+-------+------+-------+
| car   | const | PRIMARY,license_plate | PRIMARY |      30 | const |    1 |       |
+-------+------+----------------------+---------+---------+-------+------+-------+

+-------+------+--------------+-----------+---------+-------+------+-------------+
| table | type | possible_keys | key       | key_len | ref   | rows | Extra       |
+-------+------+--------------+-----------+---------+-------+------+-------------+
| g     | const | PRIMARY,id    | PRIMARY   |      31 | const |    1 |             |
| c     | ref   | garage_id     | garage_id |      32 | const |   10 | Using where |
+-------+------+--------------+-----------+---------+-------+------+-------------+
```

Sonuçlar daha iyileşti. Birinci sorguda `car` tablosuna olan erişim, sorgu iyilik derecesinde 2. seviyede olan `const` seviyesine yükseldi. İkinci sorguda ise `garage` (`g`) ve `car` (`c`) tablolarının erişimi ise `garage` için `const` ve `car` için iyilik derecesinde 4. seviyede olan `ref` erişimine yükseldi.

9.2.10 Oracle SQL*Loader

Metin bazlı bilgileri Oracle veri tabanına yüklemek istiyorsanız, bunun en rahat yolu SQL*Loader adlı programı kullanmaktır. SQL*Loader, kontrol dosyası denilen bir ayartanım dosyası eşliğinde, virgül ayırımlı, boşluk ayırımlı, tab ayırımlı, ya da sabit uzunluktaki kolonlar içeren metinlerin hepsini veri tabanına yükleyebilir. Bunu bir örnek üzerinde görelim: Önce üzerinde yükleme yapacağımız tabloyu veri tabanında yaratalım.

```
create table musteri (

  cust_nbr       number(7)       not null,
  cust_name      varchar2(100)   not null,
  cust_addr1     varchar2(50),
  cust_addr2     varchar2(50),
  cust_city      varchar2(30),
  cust_state     varchar2(2),
  cust_zip       varchar2(10),
  cust_phone     varchar2(20),
  cust_birthday  date)
/

create table hesap (

  cust_nbr       number(7)       not null,
  acct_nbr       number(10)      not null,
  acct_name      varchar2(40)    not null)
/
```

Üstteki komutları kullanarak tabloyu yarattıktan sonra, örnek metin dosyalarını `SqlLodader` projesinden alıp kullanabilirsiniz.

Sabit Uzunluklu Kayıt Yüklemek

Şimdi, SQL*loader'ın çalışması için bir kontrol dosyası lazım. Müşteri verisi için yazılmıs aşağıdaki kontrol dosyası örneğini, load1.ctl adında diskinize yazın. Bu örnek, sabit uzunluklu bir veri dosyasını yüklemek için verilmiştir.

```
LOAD DATA
INFILE 'cust.dat'
INTO TABLE musteri
   (cust_nbr     POSITION(01:07)  INTEGER EXTERNAL,
    cust_name    POSITION(08:27)  CHAR,
    cust_addr1   POSITION(28:47)  CHAR,
    cust_city    POSITION(48:67)  CHAR,
    cust_state   POSITION(68:69)  CHAR,
    cust_zip     POSITION(70:79)  CHAR,
    cust_phone   POSITION(80:91)  CHAR,
    cust_birthday POSITION(100:108) DATE "DD-MON-YY" NULLIF
                                    cust_birthday=BLANKS)
```

Artık müşteriyi veri tabanına yüklemeye hazırız. Aşağıdaki komut ile bunu yapabiliriz.

```
\$ sqlldr kullanici/sifre control=load1.ctl log=load1.log bad=load1.bad
   discard=load1.dis
```

Bilgisayardan gelen yanıt:

```
SQL*Loader: Release 8.0.3.0.0 - Production on Wed Mar 10 8:10:23 1999
(c) Copyright 1997 Oracle Corporation. All rights reserved.
Commit point reached - logical record count 23
```

Çıktıya (`sqlloader_musteri_cikti.txt`) bakarak SQL*Loader'ın başarıya ulaşıp ulaşmadığını anlayabiliriz.

Bu sonuca göre, olanlar şunlardır: 2'inci kayıt veri tabanına kabul edilmedi, çünkü tarih verisi Oracle tarafından geçersiz bulundu. Bu geçersiz kayıt, kötü (bad) dosyasına yazıldı. Kötü dosyasının ismini komut satırından `bad=load1.bad` ibaresini kullanarak verilmişti.

Değişken Uzunluklu Kayıt Yüklemek

Şimdi hesap verisini virgül ayrıklı bir veri dosyasından veri tabanına yükleyelim. Bu şekilde bir yükleme için lazım olan kontrol dosyası (load2.ctl) aşağıdadır.

```
LOAD DATA

INFILE 'acct.dat'

INTO TABLE hesap

FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(cust_nbr, acct_nbr, acct_name)
```

Bu kontrol dosyasını da komut satırından işletelim.

```
\$ sqlldr kullanici/sifre control=load2.ctl log=load2.log
  bad=load2.bad discard=load2.dis
```

Bilgisayardan gelen yanıt:

```
SQL*Loader: Release 8.0.3.0.0 - Production on Wed Mar 10 8:10:23 1999
(c) Copyright 1997 Oracle Corporation. All rights reserved.
Commit point reached - logical record count 12
```

Çıktıya (`sqlloader_hesap_cikti.txt`) bakarak SQL*Loader'ın başarıya ulaşıp ulaşmadığını anlayabiliriz.

Bu çıktıya bakarak, şu sonuca varabiliriz: Bütün kayıtlar hatasız bir şekilde yüklendi. Bu örnek veri dosyası değişken uzunluklu veri içerdiği için, metin dosyasının içinde bir ayraç gerektiğini hatırlatmamız lazım. Bu örnek içinde ayraç olarak virgül işareti kullanıldık.

Kayıt Eklemek

INTO TABLE hesap şeklindeki sözdizimi, boş olan bir tablo farzediyordu. Eğer içinde zaten veri mevcut olan bir tabloya veri eklemek istiyorsanız, INTO TABLE ifadesinin başına APPEND eklemeniz gerekiyor. Yani,

```
LOAD DATA

INFILE 'acct.dat'

APPEND INTO TABLE hesap

FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    (cust_nbr, acct_nbr, acct_name)
```

Özet

SQL*Loader için gereken ayarları burada özetleyelim.

- Yüklenecek veri dosyasının ismi, kontrol dosyasında INFILE ibaresinden sonra verilir.
- Hedef tablosunun ismi, kontrol dosyasında INTO TABLE ibaresinden sonra gelir.
- Değişken uzunluklu veri dosyaları için ayraç karakteri, kontrol dosyasında FIELDS TERMINATED BY ibaresinden sonra verilir.
- Sabit uzunluklu veri dosyaları için başlangıç ve bitiş kolonları, kontrol dosyasında POSITION(BAŞLANGIÇ:BİTİŞ) olarak tanımlanır.

9.3 Transaction

Veri tabanını üzerinde işletilen her SQL komutu bir transaction altında yapılır. Transaction, bir ve ya daha fazla SQL işlemini kapsayabilir, zâten çoğunlukla kullanılma amacı budur: Hep beraber etki etmesi gereken değişiklikler için birleştirici bir ünite sağlamak. 2.2.5 bölümündeki örneği tekrarlamak gerekirse, bir banka uygulamasında bir müşterinin iki banka hesabı olsa, ve bu müşteri bir hesaptan ötekine para transfer etmek istese, transaction sayesinde bir hesaptan eksiltmek için yapılan `UPDATE` SQL işlemi, para eklemek için yapılan ekleme `UPDATE` SQL işlemi ile aynı anda etki edebilecektir. Bu iki işlemi aynı transaction altına koyabiliriz.

Transaction kavramı, bir ilişkisel tabanda veri doğruluğunu, veri taban bağlantısını, tablolar üzerine konan kilitleri ve verinin en son görüntüsünü biraraya getiren ve hepsi ile yakın alâkası olan merkezi bir kavramdır. Bir tabanda gerçekleştirdiğimiz her SQL sorgusu, bu sebeple, bir transaction altında yapılacaktır; Eğer bundan haberimiz yoksa, yâni bilerek bir transaction başlatıp durdurmuyorsak kapalı kapılar altında bizim için bir tane muhakkak başlatıp bitiriliyordur.

SQL kullanırken bir transaction'ın başlaması, SQL sorgusunun tabana gelmesiyle ya da transaction başlatma emrinin verilmesiyle olur. Eğer o anda işleyen başka bir transaction yok ise, taban bir tane otomatik olarak başlatır. Transaction'ın bitmesi `commit` ya da `rollback` komutlarından bir tanesi ile olacaktır. `Commit`, o transaction altında yapılan değişikliklerin *kalıcı* olmasına karar verildiği anlamına gelir.

Bir `commit` gelmeden önce diğer transaction'lar (yâni diğer veri taban bağlantılarında SQL işletmekte olanlar) bizim son değişikliklerimizden tamamen habersiz olacaklardır. Bu diğer bağlantılar, sadece verinin bizim transaction *başladığı andaki son* hâlini görürler. Ama bizim bağlantımız içinde içinde biz kendi güncelleme, silme, ekleme işlemlerimizin sonucunu görebiliriz.

Eğer bir transaction altındayken (ve bir takım işlemlerden sonra) bir `rollback` komutu gelirse, bu, yapılan veri değiştirme işlemlerinden *vazgeçildiği* anlamına gelecektir. O zaman veri tabanı, yapılan hiçbir değişikliği kalıcı yapmaz, ve öteki bağlantılar ve transaction'lar eski veriyi görmeye devam ederler.

> **Not**: Günümüzdeki popüler kullanıma göre iki değişik transaction, her zaman iki değişik bağlantı (db connection) anlamına gelir. Aynı bağlantı içinden içiçe geçmiş (nested) transaction başlatmak gereksiz bir karmaşıklık yaratmaktadır, ve bu sebeple konumuz dışındadır.

Eğer bir transaction'ı başlatmış süreç (process) çökerse, o veri tabanı bağlantısı, ve o bağlantıda başlatılmış olan transaction `rollback` edilecektir. Bu, kurumsal bir uygulamanın veri bütünlüğü için çok önemlidir; `Commit` sinyali gelen kadar hiçbir değişikliğin kalıcı olmaması hayati önem taşır. Çöken bir sürecin o ana kadar veri tabanını ne durumda bıraktığını bilemeyiz, belki yapılacak SQL

işlemlerinin sadece yarısı gerçekleştirilebilmiştir. Bu durumdaki bir transaction'ı commit etmek, intihar demek olurdu! Meselâ müşteri örneğine dönersek, belki müşterinin hesabında para eksiltilmiş, ama öteki hesabına para eklenemeden süreç çökmüştür. Bu durumdaki bir transaction commit edilecek olsa, müşteri para kaybetmiş olurdu! Bu tolere edilecek bir davranış değildir. Bu sebeple veri taban ürünleri, çöken bir sürecin bağlantısını ve onun içindeki olduğu transaction'ı otomatik olarak `rollback` ederler.

Ayrıca tüm modern veri taban ürünlerinde transaction ve satır seviye kilit (row level lock) yakınen (birebir) bağlantılıdır. Şöyle ki; Eğer bir satır üzerinde `UPDATE` komutu ile güncelleme yapılmışsa ve o transaction `commit` edilmeden ikinci bir `UPDATE` aynı satırı güncellemeye kalkarsa, ikinci `UPDATE` birincinin bitmesini *bekleyecektir*. Bitmek, daha önce belirttiğimiz gibi, ya `commit` ya da `rollback` ile olması mümkün bir işlemdir.

İkinci `UPDATE` birinci bittiği anda işleme konur ve o da bittiğinde (`commit` ya da `rollback` ile) geri gelir.

9.4 Beklemeden Kitlemek

Eğer bir satır üzerinde kilit olmasını bekliyor, ve o kilit üzerinde takılmak istemiyorsak, Oracle'a o satırı *beklemeden* kitlemeyi denettirebiliriz. Bunun için;

```
SELECT ... WHERE .. FOR UPDATE NOWAIT;
```

komutu kullanılır. Normâlde `SELECT FOR UPDATE` komutunun kitleme davranışı, `UPDATE` komutununki ile aynıdır. Ayrıca `NOWAIT` eki, Oracle'a eğer bir kilit mevcutsa beklemeden dönmesini söyleren ek bir davranış sağlar.

Kilit beklemeden dönme yöntemi toptan (batch) işlem yapan programımız için faydalı olabilir. Meselâ, her yeni `GARAGE`'ın altındaki her `CAR` nesnelerini bir şekilde işlemden geçirmemiz istense ve uygulamamızın çok süreçli (multi process) bir hâlde çalışması istense, birden fazla sürecin aynı tablo üzerine akın etmesi bir problem doğurabilirdi. Paylaşım mekanizması olarak `GARAGE` tablosu üzerinde `SELECT FOR UPDATE NOWAIT` ile alınan bir kilidi kullanırsak, ikinci, üçüncü, vs. gelen süreçler, beklemeden dönecekler, ve başka bir `GARAGE` satırını kilitlemeye (kapmaya) uğraşacaklardır.

Böylece bir `GARAGE` satırının işlenmekte olduğunu, üzerinde kilit olmasından anlayabiliriz. Alternatif olarak `GARAGE`'a `STATUS` adında bir kolon eklesek (üzerinde 'işleniyor','işlenmiyor' gibi bir durum kodu kullanılarak), bu işimize yaramazdı, çünkü bir `GARAGE`'ı işlemekte olan sürecin çökme durumunda, `STATUS` kolonu eski hâline getirilemeyeceği için, sürekli işleniyor modunda gibi gözükecekti; Halbuki kilit kullanıldığı şartlarda, sürecin çöküşü satır kilidinin otomatik olarak bırakılması anlamına gelecek, böylece yeni bir sürecin aynı satırı kitleyerek işleme kalınan yerden devam etmesine izin verilecebilecektir. Altta bahsedilen toptan işlemin taklit kodunu (pseudocode) görüyoruz.

```
list = run_query(‘‘SELECT * from GARAGE’’);
```

```
for garage in list
begin
   result = ‘‘SELECT * FROM GARAGE ’’ +
            ‘‘where ID = garage.id FOR UPDATE NOWAIT’’;
   if (result.size == 0)
   begin
      continue;
   else
      process_cars(garage);
   end;
end
```

9.5 Kurmak

Bu bölümde MySQL, PostgreSQL ve Oracle'ı SuSe Linux üzerinde nasıl kuracağımızı göreceğiz.

9.5.1 Linux Üzerinde Oracle

İlk önce, tüm Oracle işletim seviyesi işlemlerini yapmak için, `oracle` adında bir Unix kullanıcısı yaratmanız gerekiyor. Aşağıdaki komutları `root` olarak işletin.

```
groupadd dba

groupadd oinstall

groupadd oper

mkdir /home/oracle

chown -R oracle /home/oracle

useradd -g oinstall -G dba,oper oracle

passwd oracle
```

Oracle, bir Unix kuruluşundan belli ayarlar bekler, sistemin paketten çıktığı haliyle çalısmayacaktır. SuSE Linux için kernel seviyesinde yapılması gereken ayarlar aşağıda verilmiştir.

Liste 9.1: /etc/sysctl.conf

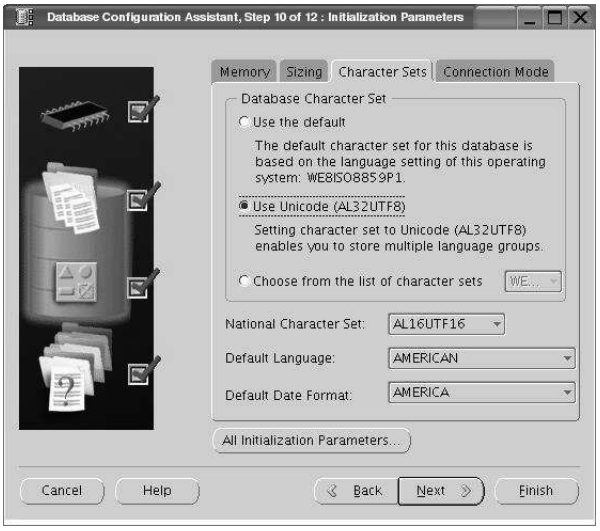
```
kernel.shmall = 2097152
kernel.shmmax = 2147483648
kernel.shmmni = 4096
kernel.sem = 250 32000 100 128
fs.file-max = 65536
net.ipv4.ip_local_port_range = 1024 65000
```

Bu ayarların işleme konması için, SuSE Linux üzerinde `/sbin/chkconfig boot.sysctl on` komutu işletilmelidir.

Artık kuruluş işlemini başlatabiliriz. Bunun için `oracle` kullanıcısı altından `./runInstaller` komutunun işletilmesi gerekiyor. Bu program, X Windows (10.8) üzerinden çalışan görsel bir programdır. Olağan değerleri kullanmak için "Next" düğmesine basarak geçebilirsiniz. Çoğu Oracle kuruluşu için olağan değerler uygundur. Daha ileri türden kuruluşlar için, DBA'inize danışın.

Oracle işler kodlarının kuruluşu bittikten sonra, sıra bir veri tabanı yaratmaya geliyor. Oracle kuruluş sırasında kurulan veri tabanı yerine bir başkasını yaratmak isterseniz, `dbca` görsel programını başlatarak istediğiniz tabanı yaratabilirsiniz. Aynı şekilde, taban için bir isim vererek, geri kalan olağan değerler ile taban yaratmak mümkündür.

Bu aşamada en önemli ve değişik yapmanız gereken işlem, taban seviyesinde Türkçe karakter desteği için yapmanız gereken değişikliktir. Şekil 9.3, istenilen karakter desteği için kullanılması gereken ekranı ve seçilmesi gereken değerleri gösteriyor.



Şekil 9.3: Türkçe Karakter Set Desteği

Taban yaratıldıktan sonra, `/etc/oratab` dosyasına girin, ve burada, yeni yarattığınız taban için olan satırın en sonundaki değeri "N" değerinden "Y" değerine getirin. Bu dosya, `dbstart` komutu kullanıldığında hangi veri tabanlarının başlatılması gerektiğini kontrol eder. Bundan sonra `oracle` kullanıcısından `dbstart` kullanarak tabanınızı başlatabilirsiniz.

```
oracle@linux:~> dbstart
```

```
SQL*Plus: Release 10.1.0.3.0 - Production on Mon Jun 6 15:07:36 2005

Copyright (c) 1982, 2004, Oracle. All rights reserved.

SQL> Connected to an idle instance.
SQL> ORACLE instance started.

Total System Global Area 285212672 bytes
Fixed Size                778776 bytes
Variable Size           95428072 bytes
Database Buffers       188743680 bytes
Redo Buffers              262144 bytes
Database mounted.

Database opened.
SQL> Disconnected from Oracle Database 10g Enterprise Edition Release
     10.1.0.3.0
- Production
With the Partitioning, OLAP and Data Mining options

Database "orcl" warm started.
```

Bu taban üzerinde eğer admin seviyesi işlemler yapmak isterseniz, `dbca`'den admin için verdiğiniz şifreyi kullanarak Oracle'a bağlanabilirsiniz. Meselâ, yeni bir kullanıcı yaratmamız gerekse:

```
sqlplus system/<şifre>@orcl

create user kitapdemo identified by kitapdemo default tablespace users;

grant dba to kitapdemo;
```

Şifre olarak `kitapdemo` metnini kullanacak, ve olağan tablo depolama yeri (tablespace) `users` olacak `kitapdemo` adında bir kullanıcı yarattık, ve bu kullanıcıya DBA hakları verdik.

Diğer bir makinadan yeni tabanınıza bağlanmak için, o makinada Oracle Client kuruluşunu yapmış olmanız gerekir. Bu kuruluşdan sonra Oracle dizinini bulun (meselâ `ORACLE_HOME`), ve bu dizin altındaki `tnsnames.ora` dosyasına servis makinanız vere yeni Oracle tabanınız hakkında bilgileri ekleyin:

Liste 9.2: ORACLE_HOME/ora92/network/admin/tnsnames.ora

```
SAMPLE =
(DESCRIPTION = (ADDRESS_LIST =
(ADDRESS = (PROTOCOL = TCP)(HOST = 192.168.0.1)(PORT = 1521))
)
(CONNECT_DATA =
(SID=SAMPLE)
)
```

```
)
```

Burada belirtilen `HOST`, veri tabanını barındıran makinanın IP adresidir. `SID`, `dbca` ile yarattığınız veri tabanının ismidir. Artık makinanızdan `sqlplus` komutunu kullanarak uzaktaki tabana bağlanabilir ve tablolar yaratıp veri ekleyebilirsiniz. Daha önce yarattığımız `kitapdemo` kullanıcısını kullanalım.

```
sqlplus kitapdemo/kitapdemo@orcl

create table vsvs (kolon1 varchar2(20) ..);
...
```

9.5.2 Linux Üzerinde PostgreSQL

SuSE Linux üzerinde PostgreSQL kurmak için, öncelikle `http://www.postgresql.org/ftp/binary/v8.0.2/linux/suse/sles8-i386/` adresinden

- `postgresql-libs-8.0.2-1.i586.rpm`
- `postgresql-8.0.2-1.i586.rpm`
- `postgresql-server-8.0.2-1.i586.rpm`

dosyalarını indirin ve bu dosyaları `root` kullanıcısı tarafından `rpm -i` ile, teker teker ve gösterildiği sırada, işletin.

Bir PostgreSQL servisini kullanmak için ilk yapmanız gereken, veri tabanı alanını sıfırlamaktır. Fakat bunun için kullanılacak `initdb` komutunu `root` olarak işletmenize izin verilmez. PostgreSQL servisini işletmek için yeni bir kullanıcı yaratmanız gerekiyor. Unix kullanıcısı `root` olarak şunları işletin.

```
mkdir /home/postgres

useradd -d /home/postgres -s /bin/bash -p postgres postgres

chown -R postgres /home/postgres
```

Bu yeni kullanıcı olarak sisteme girin.

```
su - postgres

initdb -E UNICODE

/usr/bin/pg_ctl -D /var/lib/pgsql/data -l logfile start

createdb test
```

Yukarıdaki komutlarla, veri taban alanını sıfırladık ve Türkçe karakter kullanacak `test` adlı bir veri tabanı yarattık. En son olarak veri taban servisini başlattık.

Şimdi demo adında yeni bir kullanıcı ekleyelim. Sorulan y/n sorularına "Enter" tuşuna basarak geçebilirsiniz.

```
postgres@linux:~> createuser -P
Enter name of user to add: demo
Enter password for new user: <biz burada demo değerini kullandık>
Enter it again: <..>
Shall the new user be allowed to create databases? (y/n)
Shall the new user be allowed to create more new users? (y/n)
CREATE USER
```

PostgreSQL'ın çalıştığı sistemi ve servisini JDBC üzerinden gelecek dış bağlantılara açmak için, aşağıdaki dosyaları değiştirmeniz gerekiyor.

Liste 9.3: /var/lib/pgsql/data/postgresql.conf

```
listen_addresses = '*'
```

Liste 9.4: /var/lib/pgsql/data/pg_hba.conf

```
host all      all        192.168.0.2     255.255.255.255 trust
```

satırını ekleyin. Bu satırda belirtilen 192.168.0.2 IP değeri, *bağlanan* bilgisayarın IP adresi olmalıdır. Bu değerlerin etkiye geçmesi için, komut satırında postgres kullanıcısı olarak

```
/usr/bin/pg_ctl restart
```

komutunu işleterek yeni değerlere işleme sokabilirsiniz. Ayrıca, PostgreSQL kullanıcısının (örneğimizde demo) bir tabloya erişebilmesi için

```
grant all on <tablo ismi> to public;
```

komutunu vermeyi unutmayın.

9.5.3 Linux Üzerinde Mysql

MySQL'i Linux ve Solaris üzerinde kurmanın en sağlam yolu, RPM değil, tar¬ .gz sonekli bir dosyayı kullanıp bu dosyayı uygun yere açmanızdır. Bunun için, http://dev.mysql.com/downloads/ adresinden istediğiniz genel sürüm numarasına giderek, buradan Linux (non RPM package) downloads altındaki Standard satırından Pick A Mirror'a tıklayarak mysql-standard-xxx-pc¬ -linux-i686.tar.gz gibi bir dosyayı indirmeye başlayın. İndirdikten sonra aşağıdaki komutları root olarak sırasıyla uygulayın.

```
tar xvzf mysql-standard-4.0.23-pc-linux-i686.tar.gz

ln mysql-standard-4.0.23-pc-linux-i686 -s /usr/local/mysql

cd /usr/local/mysql
```

```
groupadd mysql

useradd -g mysql mysql

./scripts/mysql_install_db --user=mysql

chown -R root .

chown -R mysql data

chgrp -R mysql .

bin/mysqld_safe --user=mysql &

bin/mysql
```

TAR dosyasını açtık, veri tabanını hazırladık ve `mysqld_safe` ile veri taban servisini başlattık. En son komut ile, mysql komut satırına girmemizi sağladı. Bu komut satırında ilk iş olarak MySQL `root` kullanıcısı için bir şifre tanımlamanız iyi olur. Altta bu şifreyi `admin` olarak seçtik.

```
mysql> SET PASSWORD FOR root@'localhost' = PASSWORD('admin');
```

Artık, MySQL komut satırına salt `bin/mysql` ile değil, `bin/mysql -p` ile girmemiz gerekiyor. Şifre sorulunca `admin` girebiliriz.

En son yapılması gereken, `root` kullanıcısını dış makina bağlantılarına açmaktır. Bunu yapmazsak, MySQL'e sadece `localhost`'tan bağlanabiliriz, ki bu da dağıtık bir uygulama için pek faydalı olmazdı.

```
mysql> GRANT ALL ON *.* TO root@'%'
-> IDENTIFIED BY 'admin'
-> WITH GRANT OPTION;
```

Eğer `mysqld_safe` başlamaz ise, `mysqld` programını deneyebilirsiniz. Başlattıktan sonra `mysql` komutu problem çıkartırsa, meselâ `mysql.sock` dosyasının bulunamaması gibi, o zaman bu dosyayı istenen/aranan yere bir Unix sembol bağlantı (symbolic link) ile bağlayarak tekrar deneyebılırsınızç

```
ln -s /var/lib/mysql/mysql.sock /tmp/mysql.sock
```

Bölüm 10

Unix

Bu Bölümdekiler

- Unix felsefesi
- Süreçler
- Kullanıcılar
- Araçlar

Unix bir güzelliktir. Unix, en az ile en fazla yapmanın en başarılı örneği, bir yalın tasarım abidesidir. Unix bize bir yazılım mimarinin nasıl olması gerektiğinin en çıplak şekliyle anlatır, tarif eder. Piyasada bu kadar zaman kalabilmiş, ve kritik uygulamalarda servis tarafın pazar payını tamamen eline geçirmiş olmasında bu doğru tasarımın büyük etkisi vardır. Aslında, kitabımızın diğer bölümlerinde önerdiğimiz mimariler ve teknolojiler, ruh olarak Unix felsefesine birebir uyumludurlar. Çünkü biz bu felsefenin takipçileri, ve bu felsefeyi projelerimizde kullanabilmiş sektörün elemanlarıyız.

Peki bu felsefe nedir?

Unix, tasarım bakımından özel hallerden kaçınmış bir işletim sistemidir[14, sf. 77]. Herkes için herşey olmaya çalışmaz, ama herşeyin yapılabilmesine izin verecek en baz altyapıyı sağlar. Bunun yan etkilerinde bir tanesi, altyapının uygulamalardan net bir şekilde ayrılması, ve böylece uygulamanın bir başkası ile rahat bir şekilde değiştirilebilmesidir.

Meselâ komut satırı programı, DOS'ta olduğu gibi işletim sisteminin içine sokuşturulmamıştır. Bu sadece bir görevten ibarettir, ve Unix'te ayrı programı ayrı süreç altında (aynen diğer uygulamalar gibi) çalışır. Unix'te "birşeyler yapan herşey" bir süreç, bir işlemdir[14, sf. 77]. Bu ayrılık sayesinde Unix dünyasında birçok komut satırı alternatifi geliştirilmiştir (`bash`, `sh`, `tcsh`, `ksh`) ama DOS dünyasında hâla sadece tek bir komut satırı mevcuttur.

Unix işletim sisteminin yalın yapısı, insanları en başta en çok şaşırtan, ve sonra merakını uyandıran faktörlerden biridir. Unix'de neredeyse herşey altı temel işlemle yapılır, ki bunlara "sistem çağrıları" adı verilmiştir; Bu altı temel sistem çağrısıyla neredeyse herşeyi geliştirebilirsiniz. Bu altı temel çağrıyı kavrarsanız, Unix'i kavramış olursunuz[14, sf. 88].

Yâni Unix'in güzelliklerinden birisi, karmaşık şeyler oluşturmak için karmaşık ara birimlere gereksinmediğinizi anlamış olmasıdır. Basit şeylerin etkileşimiyle istediğiniz kadar karmaşık yapılar oluşturabilirsiniz. Yapmanız gereken, karmaşık problem çözme yapıları yaratmak için basit işlemler arasında iletişim kanalları (pipe) oluşturmaktır[14, sf. 78].

Temel bilimlerde de bir fiziksel olayı açıklayacak matematiksel kuramlar arasında seçim yapılırken aranan özelliklerden birisi basitliktir. Bilim dünyasına yön veren önemli bir deyiş olan Occam'ın Usturası der ki: "Mevcut alternatifler arasında en basit açıklamayı seçiniz". Temel bilimler, tarih boyunca ve günümüzde bu desturu takip etmektedirler. Kurumsal programcılıkta da aynen temel bilimlerde olduğu gibi, basitlik *esastır* (Kural #1). Bilgisayar dünyasında, karışık yapıyı yaratmak programcının işidir (uygulama yazmak), kıyasla temel bilimler karışıklıktan basit açıklamaya doğru gitmeye çalışır; Fakat felsefe her iki taraf için aynıdır. Sadece gidilen yön değişiktir.

10.1 Unix Araçları

Bahsedilen türden basit yapı, Unix için birçok aracın yazılmasını sağlamıştır. Unix felsefesine uyumlu olan Unix araçları, aynen işletim sistemin kendisinin olduğu gibi, sadece *bir* işi, *en iyi* şekilde yapmak için yazılırlar.

Kitabımızın ana amacı kurumsal yazılım sektörde çalışan programcılar ve teknik liderlere yardım etmek olduğu için, her Unix aracının tüm özelliklerini teker teker anlatmayacağız. Bizim faydalı olduğunu düşündüğümüz bilgi, *ihtiyaçlar* ışığında hangi Unix aracı ve araçlar dizisinin (çoğu zaman iletişim kanalları ile birkaçını birbirine bağlayarak) kullanılması gerektiğini anlatmaktır.

10.1.1 Komut Birleştirme

Unix araçları arasında iletişim kanalı oluşturmak için "`|`" işareti kullanılır. Meselâ eğer `command1` ve `command2` adında iki Unix programımız olsa ve `command1`'in çıktısını `command2`'ye göndermek istesek, o zaman

```
command1 | command2
```

komut sırasını kullanırdık. Burada en son işletilecek komut olan `command2`'nin çıktısı (büyük ihtimalle) ekrana basılacaktır.

Ekrana basılacak bir çıktıyı bir dosyaya yönlendirmek istersek, "`>`" işareti ya da "`>>`" işaretinden sonra bir dosya ismi kullanılır. "`>`" işareti, yeni bir dosyaya yazmak, "`>>`" ise mevcut bir dosyaya eklemek için kullanılır.

```
command1 | command2 >> /tmp/out.txt
```

Bu işletim sırasına göre, önce `command1` çağırılacak, onun çıktısı `command2`'ye girdi olarak verilecek ve `command2`'nin çıktısı `/tmp/out.txt` adlı dosyaya yazılacaktır.

10.1.2 Süreçler

Unix'de işlettiğiniz her program bir süreç (process) hâline gelir.

Listelemek

O ana kadar sizin başlatmış olduğunuz süreçleri görmek istiyorsanız, `ps` komutunu kullanabilirsiniz. Şuna benzer bir çıktı gelecektir

```
 PID   PPID   PGID    TTY  UID   STIME COMMAND
 452      1    452    con 1004 11:55:54 /usr/bin/bash
1656    452   1656    con 1004 11:55:58 /usr/bin/ps
```

Bu listedeki `PID` kolonu sürecin kimlik numarasıdır. Eğer sistemde, sizin dahil, tüm kullanıcıların başlatmış olduğu süreçleri görmek istiyorsanız,

```
ps -eaf
```

komutunu kullanabilirsiniz. Bu sefer daha büyük bir liste gelecektir (çünkü herkesin süreçlerini görmek istediniz).

Öldürmek

Bir süreci öldürmek için `PID` no'sunu öğrenip o numarayı kullanarak şu komutu kullanabilirsiniz.

```
kill -9 <PID>
```

Eğer bir programı `ps -eaf` listesinde çıkan bir "isme" göre öldürmek istiyorsanız, çoğu Unix'de bunun için bir `pkill` komutu vardır. Ama dikkat edin! Mesela `pkill java` gibi bir komut, sistemde işleyen tüm Java programlarını öldürür; Eğer aynı makinada birden fazla kişi Java uygulamasını test ediyor ve siz de `root` iseniz, insanların çalışmasını etkileyebilirsiniz. Bu yüzden bu komutu dikkatli kullanın.

10.1.3 Dosyalar

Ekrana Basmak

`cat` ile bir dosyayı tamamen ekrana basabiliriz. Eğer dosyayı kısım kısım görmek istiyorsak, `less <dosya>` komutu işe yarar. `less` işlerken bir sonraki sayfaya geçmek için `SPACE`, geri gitmek için `b`, en tepeye gitmek için `g`, ve en alta gitmek için `G` tuşları kullanılabilir. `q` ile `less` programından çıkmak mümkündür. `less` komutu, ondan önce gelmiş olan `more` komutundan daha kuvvetlidir, tavsiyemiz `less` kullanmanızdır. Buradaki isimlendirme esprisi, raslantısal bir şekilde (ya da bilerek) Unix felsefesinin özünü yine ortaya koymuştur: (daha az anlamına gelen) `less` komutu (daha fazla anlamına gelen) `more`'dan daha kuvvetlidir!

Dosya İçinde Bulmak

Herhangi bir dizin altındaki bir veya daha fazla dosyalar *içinde* bir kelimeyi aramak istiyorsanız, `grep` komutunu kullanabilirsiniz.

```
grep 'aranan kelime' *.java
```

gibi. Bu komut sonucundan bulunan dosyalar listenecektir.

Dosya Bulmak

Bir isim düzenine uyan tüm dosyaları bulmak için, `find` komutu kullanılır.

```
find /usr/local -name '*.java'
```

Bu komut `/usr/local` seviyesinden başlayarak sonu `*.java` ile biten tüm dosyaları bulup geri getirecektir. `find` komutundan hemen sonra gelen dizin, aramanın nereden başlayacağına işaret eder.

Eğer bulunan dosyalar *üzerinde* bir komut işletmek istiyorsanız, `find` ile `xargs` komutunu birbiri ile iletişim (pipe) kurdurarak kullanabilirsiniz. Meselâ sonu `*.java` ile biten tüm dosyaları ekrana basmak için

```
find . -name '*.java' | xargs cat
```

Dosya ve Dosya İçinde Bulmak

İşte Unix felsefesinin bir örneği: Öğrendiğimiz iki komutu birleştirerek, bir dizinden başlayarak hem o dizin hem de altındaki tüm dizinler altında belli bir düzene uyan dosyalar *içinde* bir kelimeyi arıyorsak,

```
find . -name '*.java' | xargs grep -l 'aranan kelime'
```

komutunu kullanabiliriz. `-l` seçeneği, bulunan dosya ismini ekrana basmak için kullanılır.

Dosya ve Dizin Büyüklüklükleri

Bir dizinin ya da tek dosyanın ne kadar yer tuttuğunu anlamak için, `du` kullanılır. Tek başına kullanılınca, komutun kullanıldığı dizin altındaki *her dizinin* ne kadar büyük olduğu ekranda gösterilecektir. Eğer `du -s` kullanılırsa, bu alt dizin büyüklüklerinin toplamı alınacaktır, yâni tek sayı geri gelecektir.

Eğer bir dizinin altındaki hangi alt dizinin en fazla yer tuttuğunu merak ediyorsak (ki bu bazen çok işe yarar), o zaman, `du`'dan gelen sonuçları başka bir Unix komutuna iletmek gerekecektir. Bu diğer komur, sıraya dizme (sort) komutudur.

```
du | sort -n
```

Bu komut dizinleri büyüklük sırasına göre küçükten büyüğe doğru dizer. Peki `sort` komutu, `du`'dan gelen sonuç üzerinde ilk kolona (büyüklüğe) bakacağını nereden bildi? Çünkü `du`'dan gelen sonuçlarda büyüklük, ilk kolonda idi, ve `sort`, satırları ilk karakterlerinden başlayarak dizmek için yazılmıştır.

Satır Sayısı

Bir dosyanın kaç satır ve kaç kelime içerdiğini anlamak için `wc` kullanılır. Tek başına `wc` bir dosyadaki satır, kelime sayısını beraber gösterir, ayrı ayrı bilgi almak için `wc -l`, sadece satır sayısını, `wc -w` kelime sayısı için kullanılabilir.

İçerik Değiştirmek

Bunun için Perl kullanacağız. Komut satırında Perl, hem yerinde değişiklik ve hem de düzenli ifade kullanabilmektedir.

```
perl -pi -e 's/filan/falan/sg'
```

komutu, `filan` ile `falan` kelimesini değiştirir. İş bittikten sonra bir yedek dosyasi (`.bak`) bulacaksınız.

10.2 Kullanıcılar

Unix'de her kullanıcı `/etc/passwd` dosyasında tutulur. Bu dosyaya erişmeye sadece `root` kullanıcının hakkı vardır. Projenizde genelde programcılara `root` erişimi verilmez, ama teknik lidere bu hak tanınır. Bu, Unix'i daha yeni öğrenmekte olan programcıların yanlışlıkla sistemi hasar vermesine karşı yapılır.

En güçlü kullanıcı olan `root`, birçok admin odaklı işi yapabilir, bu yüzden hakları tüm diğer kullanıcılardan daha fazladır. Mesela `root`, herkesin sürecini `kill` ile öldürebilir. Diğer kullanıcılar böyle bir şey yapmaya kalkışsa, sistem onlara izin vermeyecektir.

Bir kullanıcın giriş yaptığında hangi komut satır progamını (shell) kullanacağı (birçok seçenek mevcuttur), `/etc/passwd` içinde tanımlıdır. Bu shell, kullanıcı yaratılırken `useradd` komutuna parametre olarak verilir;

```
useradd -d /home/user123 -s /bin/bash user123
```

Parametre `-d` kullanıcı (`HOME`) dizini, `-s` ise shell tipi için kullanılır.

Bir kullanıcı sisteme girdiğinde, ya da bir xterm, yeni bir shell başlattığında, kullandığı shell türüne göre işletilen ayar dosyası değiştiktir. Shell `sh`, ve `bash` ise, önce `/etc/profile` script'i, sonra kullanıcı `HOME` dizini altındaki `.profile` işletilir. Eğer admin isek ve tüm kullanıcıların etkileneceği türden bir değişiklik yapmak istersek, bunu `/etc/profile` içinde yaparız. Her kullanıcı kendi isteğine göre `.profile`'ı değiştirebilir.

Her kullanıcı için bir `HOME` değişkeni mevcut olacaktır. Bu değişkeni ekranda göstermek için `echo \$HOME` komutunu kullanabilirsiniz.

Erişim haklarını daha geniş bir kategori üzerinden idare edebilmeye admin'ler için gereklidir. Bu ihtiyaca cevaben Unix, "kullanıcı grupları" kavramını destekler. Bir kullanıcı birden fazla gruba dahil olabilir. Bir kullanıcıyı `useradd` ile yaratırken hangi gruba dahil olmasını istediğinizi biliyorsak, `-g` seçeneği ile bu grubu tanımlayabiliriz. Sisteme bir grup eklemek için `groupadd` komutunu kullanmak gerekiyor.

Eğer `useradd` işlemi bittikten sonra kullanıcıyı bir gruba dahil etmek istersek, `/etc/passwd` dosyasını güncelleyerek bunu başarabiliriz. Bu dosyada kullanıcının tanımlandığı satırı bulup, oradaki grup listesine yeni grup no'sunun eklenmesi gerekiyor. Grup ismine bakadar grup numarasını bulmak için `/etc/group` dosyasına bakabiliriz.

Bir kullanıcı hangi gruba dahil olduğunu görmek isterse, komut satırından `groups` komutunu vermesi yeterlidir. Süper kullanıcı `root` başkalarının gruplarına bakabilir, bunu için `groups <username>` komutunu kullanır.

10.2.1 Dosya Hakları ve Kullanıcılar

`ls -al` ile bir dizindeki tüm dosyaların kullanım haklarını görebilirsiniz.

```
drwxr-xr-x  9 burak None 0 Jun 14 15:57 Example
-rwxr-xr-x  9 burak None 0 Jun 14 15:57 Ex.txt
```

Bu listede iki birim görüyoruz; `Example` ve `Ex.txt`. Her dosya ya da dizin için onun olduğu satırın başına bakarsak, `drwxr-xr-x` gibi bir tanım görürüz. `Example` dizini örnek alalım. `drwxr-xr-x` ne demektir?

Bu tanım kelimesini, öncelikle zihnimizde dörde bölmemiz gerekir. Yâni biraz önceki örnek, şuna dönüşür: `d`, `rwx`, `r-x`, ve `r-x`. Bu bölümlerden tek hâneli birinci bölüm birimin dosya mı, dizin mi olduğunu belirtir. `d` dizin, `-` dosya demektir.

Ondan sonra gelen üç büyük grup, sırasıyla kullanıcı (user), grup (group) ve ötekiler (others) içindir. Kullanıcı, kullanıcın kendisi için geçerli olan haklar, grup, kullanıcının içinde olduğu gruptaki herkes için olan haklar, ötekiler ise bunun haricinde kalan herkes için geçerli olan haklardır. Böylece kendinize verdiğini bir hakkı, grubunuzdaki bir kişiye vermemeyi (meselâ) seçebilirsiniz.

Her kelime grubu içindeki haklar, şunlar olabilir. O dosyayı okumak (read), değiştirebilmek (write), ve işletebilmek (execute). Okumak için `r`, değiştirebilmek için `w` ve işletebilmek için `x` karakteri kullanılır. Eğer bu haklar var ise onun karakteri, yok ise `-` karakteri kullanılacaktır.

Hakların karakterleri bir grup içinde hep aynı yerlerde çıkarlar; Grubu içinde `r` hep birinci sırada, `w` ikinci sırada, ve `x` ise üçüncü sırada olacaktır.

Dosya Kullanım Hakkı Vermek ve Almak

Hak vermek için `chmod` komutu kullanılır. Kullanım `chmod <haklar> dosya` şeklindedir. Bir dosyanın tüm haklarını aynı anda set edebilen sayı yöntemi yerine (meselâ `chmod 777 file.txt`) biz, `+` ve `-` ile hak ekleme çıkarma yöntemini tercih ediyoruz. Bunun için, meselâ kullanıcıya okuma hakkı vermek için `chmod u+r <dosya>` komutu kullanılır. Hak eksiltmek için `+` yerine `-` kullanmak gerekir. Aynı şekilde gruba okuma hakkı vermek için `chmod g+r <dosya>` kullanılır. "Öteki" kullanıcılar için `o`, ve her kullanıcı *herkes* için `a` kullanılmalıdır. `a` ile tüm kullanıcılara (`u`, `g`, `o`) belli bir hakkı aynı anda verme yeteneğine kavuşuyoruz.

10.3 Scripting

Bir arada işlemesini istediğini komutları bir dosyaya koyup, bir script olarak işletebilirsiniz. Script'leriniz herhangi bir shell için yazabilirsiniz, ama bizim tavsiye ettiğimiz shell, her Unix sisteminde olması garanti olan `sh` shell'idir.

Bir script'i işletmek, kurumsal programcılar tarafından Unix'in en az anlaşılan ve en son "tamamiyle" öğrenilen tekniklerden biridir. Bir script içinde set

edilen değişkenler, *çağıran* shell'i nasıl etkileyecektir? Bir script'i birkaç şekilde çağırma şekillerinden hangisi uygundur?

Elimizde `script.sh` adında bir script olduğunu farz edelim. Bu script içinde şunlar olsun;

```
VAR1=/tmp/vs/vs
command1
command2
```

Bu script'i üç şekilde işletebilirsiniz.

1. `sh script.sh`: Script'in işlemesi tamamlandıktan sonra, çağıran shell, içeride tanımlanan ve set edilen `VAR1`'ı göremez.

2. `script.sh`: Avantajı, işletmek için shell üzerinde sadece tek kelime kullanmasıdır. Bunun için `script.sh` dosyasının en üst satırına `#!/bin/sh!` ibaresini eklemeliyiz. Bir de, işletmeden önce (sadece bir kez) `chmod u¬ +x shell.sh` ile bu dosyayı "sadece kendi kullanıcımız için" işletilebilir hâle getirmeliyiz. Bundan sonra komut satırından uygulanacak tek başına `shell.sh` komutu, çalışacaktır. Fakat `VAR1`, aynen biraz önce olduğu gibi, çağıran shell tarafından gözükmeyecektir.

3. `. shell.sh`: Bu kullanım, Unix'de kaynaklama (sourcing) yapar, yâni script *sanki içindeki her satır, komut satırı üzerinde elle yazılıyormuş gibi* işletilir. Ayrıca daha önceki işletim stillerinde, `shell.sh` çağırımı yeni bir süreç altında işliyordu, ve orada yaratılan her değişken yapılan her iş o süreç bitince onunla beraber ölüyordu. Kaynaklama yönteminde durum değişiktir. Bu yöntemde, çağırım geri gelince `VAR1`'in değeri çağıran shell tarafından görülüyor olacaktır.

Ayrıca eğer 1. ve 2. yöntemlerde de `shell.sh` işledikten sonra içeride tanımlanan ve set edilen değişkenlerin dışarıdan (çağıran shell içinden) görülmesini istiyorsak, o zaman bu yöntemlerde script içinde değişkene bir değer set ettikten sonra onu dışarıya "ithal" etmeliyiz; Bourne shell (`sh`) içinde bu `export VAR1` komutu kullanılarak yapılır.

10.4 Makina Başlayınca Program İşletmek

Bazı programların bilgisayar açıldığı zaman "otomatik" olarak başlamasını istiyor musunuz? Mesela, bir Linux makinasını tamamen Oracle icin ayırdık, ve Oracle, bilgisayar açılır açılmaz başlamalı. Sistem idarecisi olarak her seferinde oracle kullanıcısına girip, durdurma/başlatma yapmak istemiyoruz..

Unix'te bu işler, diğer pek çok şeyde olduğu gibi, metin bazlı ayar dosyaları üzerinden yapılıyor. Bir Unix sisteminde bilinen, ve önemli dizin bölgeleri vardır. Mesela `/etc/` dizini bunlardan biridir. Çogu Unix versiyonu, `/etc/` altına

onemli ayar dosyalarını koyar. Yâni, bir script ile `/etc/` altındaki dosyaları değiştirirseniz, sistemin işleyişi değişecektir. Dikkatli olmamız isabetli olacaktır.

Durdurup başlatma ayar dosyaları `/etc/rc.d` dizini altındadır. Buraya girip `ls -al` işletirseniz, aşağıdaki tabloyu görebilirsiniz.

```
drwxr-xr-x  10 root    root        4096 Sep 15 01:09 .
drwxr-xr-x  43 root    root        4096 Sep 15 01:23 ..
drwxr-xr-x   2 root    root        4096 Sep 6 15:51 init.d
-rwxr-xr-x   1 root    root        3219 Jul 10 2001 rc
drwxr-xr-x   2 root    root        4096 Apr 4 13:25 rc0.d
drwxr-xr-x   2 root    root        4096 Apr 4 13:25 rc1.d
drwxr-xr-x   2 root    root        4096 Apr 4 13:25 rc2.d
drwxr-xr-x   2 root    root        4096 Apr 4 13:25 rc3.d
drwxr-xr-x   2 root    root        4096 Apr 4 13:25 rc4.d
drwxr-xr-x   2 root    root        4096 Apr 4 13:25 rc5.d
drwxr-xr-x   2 root    root        4096 Apr 4 13:25 rc6.d
-rwxr-xr-x   1 root    root        3200 Sep 15 01:08 rc.local
..
..
```

`rc` kelimesinden sonra gelen sayı, "başlama seviyesini" belirtir. Bir Linux makinasını değişik başlama seviyesinde başlatmamız mümkündür. Meselâ çok önemli bir sistem bakımı gerekiyorsa ve bu bakım yapılırken hiçbir kullanıcının sisteme bağlanamaması lâzım ise, başlama seviyesi 0 yada 1, bu iş için kullanılabilir. Bu seviyelerde kullanıcı giriş programları sağlanmamıştır, ve böylece sadece `root` sisteme terminalden girerek istediği bakımı yapabilir.

Sistemi 4. başlama seviyesinde başlatmak demek, `rc4.d` altına girip, oradaki başlatma script'lerini işletmektir. Şimdi `rc4.d` altında ne var görelim.

```
lrwxrwxrwx   1 root    root          14 Apr 2 13:36 K74ntpd
lrwxrwxrwx   1 root    root          16 Apr 2 13:38 K74ypserv
lrwxrwxrwx   1 root    root          16 Apr 2 13:38 K74ypxfrd
lrwxrwxrwx   1 root    root          15 Apr 2 13:34 S05kudzu
lrwxrwxrwx   1 root    root          18 Apr 2 13:34 S08ipchains
lrwxrwxrwx   1 root    root          18 Apr 2 13:34 S08iptables
lrwxrwxrwx   1 root    root          17 Apr 2 13:34 S10network
lrwxrwxrwx   1 root    root          16 Apr 2 13:33 S12syslog
lrwxrwxrwx   1 root    root          17 Apr 2 13:36 S13portmap
lrwxrwxrwx   1 root    root          17 Apr 2 13:37 S14nfslock
lrwxrwxrwx   1 root    root          18 Apr 2 13:33 S17keytable
lrwxrwxrwx   1 root    root          16 Apr 2 13:34 S20random
lrwxrwxrwx   1 root    root          15 Apr 2 13:34 S25netfs
lrwxrwxrwx   1 root    root          14 Apr 2 13:34 S26apmd
```

`ipchains`, `syslog` gibi programlar tanıdık gelebilir. Bu programların başlatıldığı noktayı böylece görmüş oluyoruz.

Bir kavram daha kaldı: Bazı script'lerin "`K`" ile, ötekilerinin "`S`" ile başladığını görüyoruz. Bunun sebebi nedir? `S` "`Start`" icin `K` "`Kill`" için kullanılır, yâni

başlat ve durdur komutlarıdır. Eğer Unix sistemi, `rc4.d` altındaki servisleri başlatmak istiyorsa, önce, `rc4.d` altinda "`ls S*`" benzeri bir komut işletecektir. Bu komut sadece "`S`" ile başlayan script'leri toplar. Sonra Unix sistemi, bu script'leri teker teker "`start`" kelimesini ekleyerek cağırır. Aynı şekilde sistem kapanırken, Unix `ls K*` benzeri komut işletip, durdurmak için gerekli script'leri toplar, ve onları "`stop`" kelimesini ekleyerek çağırır.

Peki niye çağırım yaparken "`start`" ve "`stop`" eklemek gerekiyor? Bunun sebebini script içeriğine baktığımızda göreceğiz.

```
..
..
start() {
       echo -n >"Kaydediciyi baslatiyoruz.. "
..
..
stop() {
       echo -n >"Kaydediciyi durduruyoruz.. "
..
..
case ">1" in
  start) ;; eger Unix start kelimesi gondermis ise
       start
       ;;
  stop)
       stop eger Unix stop kelimesi gondermis ise
       ;;
  status)
       rhstatus
       ;;
  restart|reload)
       restart
       ;;
..
..
```

10.5 Takvime Bağlı Program İşletmek

Bir zamana, takvime bağlı program işletmek için, `cron` programı kullanılır. `cron` programı, `crontab` adında bir ayar dosyası kullanır, ve bu dosya `/etc/¬ crontab` altında bulunur. Bu ayar dosyasını ya direk (`root` olarak) ya da komut satırından `crontab -e` ile edit etmeye başlayabilirsiniz. Eğer herhangi bir dosyanın `crontab` *dosyası olarak* kullanılmasını istiyorsanız, `crontab <file>` komut ile bunu yapabilirsiniz. Bu durum genellikle `crontab` dosyasını CVS gibi bir kaynak kod idare sisteminde tutmak isteyenler için gerekli olmaktadır.

Bazı `crontab` seçenekleri (ve alâkalı bir komut) şunlardır:

- `export EDITOR=vi crontab`'in hangi editör ile açılacağını kontrol eder. `vi` yerine (eğer varsa) `emacs` kullanabilirsiniz.
- `crontab -e`: Ayar dosyasını günceller
- `crontab -l`: Ayarları ekranda gösterir
- `crontab -r`: `crontab` dosyasını siler
- `crontab -v`: Dosyayı en son güncellediğimiz tarihini gösterir

`crontab` dosyasının her satırı, değişik bir programı ayarlamak için kullanılır. Bu her satırda, her kolon, zaman ayarının belli bir bölümü için kullanılır. Kolonlar sırasıyla şunlardır: Dakika, saat, ayın günü, ay, haftanın günü ve yıl. Her kolon için yıldız "`*`" işareti, o ayarın dikkate alınmadığı anlamına gelir, meselâ gün için "`*`" var ise, o program her gün işleyecektir. Tablo 10.1 üzerinde izin verilen kolon değerlerini görüyoruz.

Tablo 10.1: Crontab Kolon Değerleri

Alan	**İzin Verilen Değerler**
dakika	0-59
saat	0-23
ayın günü	0-31
ay	0-12
haftanın günü	0-7

Örnek bir `crontab` dosyası ise, altta görülmektedir.

```
SHELL=/bin/sh
MAILTO=burak
5 0 * * *      \$HOME/bin/daily.job >> \$HOME/tmp/out 2>&1
15 14 1 * *    \$HOME/bin/monthly
0 22 * * 1-5  mail -s "Saat 10pm" burak%Burak,%%Nabersin?%
```

Her bir program tanımını teker teker tarif etmek gerekirse:

1. İlk önce işletilen programların hangi shell'i kullanması gerektiğini tanımlıyoruz. Burada seçim `/bin/sh` olmuştur.
2. Bu satırda, `cron` işledikten sonra onun işlettiği dosyaların çıktısının kime mail edileceğini tanımlıyoruz. Bu kişi burada `burak` adındaki kullanıcıdır.
3. Bu satırda, ilk takvimli program tanımı yapılmıştır. Bu program, her gece yarısından beş dakika sonra her gün işletilecektir. `\$HOME` değişkeni, `crontab` dosyasının sahibidir.

4. Her ayın ilk gününde saat 2:15pm'de `\$HOME/bin/monthly` adlı program işletilecektir.

5. Her iş günü (Pazartesi ve Cuma arasındaki her gün, ve bu günler dahil olmak üzere) saat 10 pm'de bu program işletilir.

10.6 Network Durumu

`netstat`, sisteminizdeki network durumu gösterir. Genellikle kullanılma sebebi, bir port'un o anda kullanılıp kullanılmadığını anlamaktır. Sistemde kullanılan tüm port'ların listesini almak için `netstat -a` kullanılır.

10.7 Yardım Almak

`man` komutundan sonra bir program ismi belirsek, bu program hakkındaki sistemde olan tüm açıklama ekrandan verilecektir. Unix dünyasında `man` oldukça fazla kullanılır, çünkü yeni öğrendiğimiz (hâtta bazen eski bildiklerimizin bile) komutların seçeneklerini öğrenmek için, `man komut_ismi` kullanırız.

10.8 X-Windows Kullanımı

X-windows, Unix dünyasında (Java applet'lerden çok önce) görsel herhangi bir programın ekranını, penceresini, başka bir makinanın monitoründe göstermek için kullanılan tekniktir.

X-windows ile servis tarafında bir programı shell üzerinden işletip, penceresini kendi sisteminize alabilirsiniz. Bunu yapabilmek için öncelikle masaüstü (çağıran) sisteminizde bir X ortamı gerekecektir. Böyle bir ortam, eğer masaüstü Linux ya da diğer bir Unix ise kendiliğinden olacaktır (kurulum sırasında X-windows seçtiyseniz). Windows üzerinde ise, Cygwin üzerinden X-windows kullanabilirsiniz. Cygwin kurulumu için A.6 bölümüne, Cygwin üzerinde X ortamı kurulumu için A.13 bölümüne bakabiliriz.

Görsel programı işletmek için önce servis tarafına `ssh` ile bağlanılır. `ssh` üzerinden X bilgilerinin geriye alınabilmesi için, `-X` seçeneği kullanılmalıdır.

```
(desktop) \$ ssh host1 -l root -X
```

Servis sistemine girdikten sonra, servis makinasına ekranın *nerede* olduğunu söylememiz gerekiyor. Bunun için `DISPLAY` çevre değişkenine ekranın makina ismi (ya da IP numarası) sonuna `:0.0` eklenerek verilir.

```
(host1) \$ export DISPLAY=desktop:0.0
```

Son basamak, masaüstü tarafında servis tarafına görsel bilgileri göndermesi için izin vermektir:

```
(desktop) \$ xhost +
```

Bu komut tüm servis programlarına X bilgisi göndermesi için izin verecektir. Artık X programını başlatabiliriz. Meselâ `xclock`;

```
(host1) \$ xclock
```

Bu programın görüntüsü olan bir saat, masaüstü ekranımızda çıkacaktır.

Bölüm 11

Proje Yönetimi

Bu Bölümdekiler

- Karakterler
- Hata takip sistemi
- Kaynak kod idaresi
- Geliştirme ve test ortamları

Eğer tasarım disiplini bilimden ziyade sanata yakın ise, "sanata yakınlık" yarışmasında birincilik madalyasını herhalde proje yönetimi disiplinine vermek gerekirdi. Çünkü, yazılım mühendisliğinde eşya çoğunlukla insan (bazen de makinadır), ama proje yöneticiliğinde eşya *tamamen* insandır. Bu durum, proje yönetimi disiplininin anlatılması ve sistemsel bir yapıya oturtulması açısında zorluklar çıkartır, çünkü proje yönetim tekniklerini kullanan insanlar farklı olduğu gibi yönetimin hedefi olan insanlar da projeden projeye farklılık gösterirler. İnsanları hâlen matematiksel şekilde anlatamadığımıza göre, proje yönetimi hakkında verilen her tavsiyenin kulağa gayri bilimsel gelmesi normâldir. Bu konu hakkında yazan herkes tavsiyelerini sadece kendi perspektifinden ve tecrübesine dayanarak anlatmaya mecbur kalmaktadır. Objektif, deneye dayalı sonuçları henüz literatürde görememekteyiz.

Bu yüzden, bu bölümde anlatacağımız proje yönetim teknikleri, bizim çalıştığımız projelerde işlediği gözlenen subjektif teknikler olacaktır. Örnek olarak, tanıdığımız ve saygı duyduğumuz proje yöneticilerinde iyi bulduğumuz özellikleri sunacağız, ve bunu mümkün olduğu kadar projede çalışmış diğer insanlar da bu görüşe katılmış ise yapmaya gayret edeceğiz. Teknik ve teknik olmayan arkadaşlar ile iyi bir proje yönetiminin nasıl olması gerektiği hakkında uzun yıllar boyunca görüş alışverişinde bulunduk, bunları da bu bölümde paylaşmak istiyoruz.

Kişilere ek olarak, her projede kurulması ve takip edilmesi gereken mekanik süreçlerin proje yönetimi bakımından önemli olduğunu vurgulayıp, örnekler gösterek anlatmaya çalışacağız. Süreçler bize, ne yapacağımızı "hatırlatıcı" bir özellik taşırlar; Mekanik işleri sürece devrederek (ve süreçle iletişimde tarafımızı yine mekanik düşünceyi halleden beyin bölgemize devrederek) programcılığın daha yaratıcı kısmına bu şekilde korkmadan odaklanabiliriz. Mekanik ve önemli noktalar bize süreç tarafından hatırlatılacaktır. Meselâ tasarım sürecinin bir parçası olarak kodlamaya başlamadan önce "mimariyi ortaya koyan" bir belgenin yazılmasının şart konulması, bir sürece örnek olabilir; Bu belgeyi yazma gerekliliği, üzerimizde ileriyi düşünmeye yolunda bir baskı yaratır, ve bu baskı iyi ve gereklidir. Bu örnekte süreç, normâlde yapmayabileceğimiz bir eylemi üzerimizde mecbur bırakarak, bizi doğru yöne sevketmiş ve yaratıcı düşüncemizi doğru formata doğru kanalize etmiştir.

Aynı şekilde bir hata takip süreci, uygulamada test sonucu bulunan hataların bir "hata takip sistemine" girilmesini mecbur kılabilir, ve böylece hata bulma, hata düzeltme ve kapatma gibi mekanik basamaklar bize hatırlatıcı yardımcılar hâline gelirler. Biz tüm hataları hatırlayamasak ta, *süreç hatırlar*. Biz sadece hataları sisteme girmeyi hatırlamalıyız (Kural #3, #4).

O zaman bu bölümde anlatılacak teknikleri şu başlıklar altında toplayabiliriz:

- Karakterler
  - Proje yöneticileri
  - Teknik liderler

  - Programcılar
- Geliştirme Süreçleri
  - Hata takip
  - Kaynak kod idaresi
  - Kod teftişi (code review)
  - Geliştirme ve test
- Planlama yöntemi

11.1 Karakterler

Bir yazılım takımında doldurulması gereken çeşitli roller vardır; Bu roller, bizim şimdiye kadar gördüğümüz kurumsal yazılım takımları ve danışman (consulting) şirketleri baz alınarak anlatılacaktır. Her rol için ayrı bir şahsın bulunması en optimal çözümdür, fakat bâzen bir ve ya daha fazla rol aynı kişi üzerinde de olabilir. Bu karar takımın ve projenin büyüklüğüne göre verilebilir. Fakat danışman şirketleri her zaman bir rolü, bir kişiye verirler.

11.1.1 Proje Yöneticisi

Bir yazılım projesinde teknik olması gerekmeyen nadir kişilerden biri, proje yöneticisidir (project manager). Fakat genellikle, proje yöneticilerinin programcılık geçmişi vardır, sadece, bu yaptıkları iş için zorunlu değildir. Hiç teknik olmayan, fakat çok iyi proje yöneticileri ile şahsen çalıştığımızı söyleyebiliriz.

Proje yöneticisinin yaptığı işler, yazılım projenizin hangi safhasında olduğunuza göre değişir, fakat genel olarak denebilir ki proje yöneticisi takımınıza *lojistik destek* sağlamakla yükümlüdür. Tanıdığım bir teknik lider proje yöneticileri hakkında şunu söylerdi: "Programcılar kara kuvvetleri ise, proje yöneticileri **hava** kuvvetleridir. Biz yerden taarruz ederken, onlar havadan desteği sağlar". Bu çok doğru bir gözlemdir. Hakikaten de proje yöneticilerinin en iyisi, takım bir sorun bölgesine gelmeden, o sorunu önceden tahmin edip, o konu hakkında önlem alabilendir (sorunun tasarım ile ilgisi olmamalıdır). Tanıdığım en iyi yöneticilerden biri olan Arpan Sheth, bu becerinin çok güzel bir örneğini bir projemizde göstermişti.

Arpan ile beraber çalıştığımız bu projede teknik takım, projenin ürün seçme ve teknik bilgi toplama aşamasındaydı. Fakat, daha o safhaya gelmemiş olsak bile, geliştirmeye başlamak için gerekli Sun Solaris makinalarımız hâla ortalarda yoktu. Geliştirmeyi o makinalar üzerinde yapmamız gerekiyordu, ve o aşamaya gelmemiş olsak bile, o aşamaya geldiğimizde yere düştüğümüz anda koşmaya başlamamız (hit the ground running) için, makinaların bir an önce gelmesi çok önemliydi.

Ve proje yöneticimiz Arpan, geciken makinaların problem olabileceğini aylarca önceden tahmin etmişti. Kendisi hemen gerekli yerlerde düğmelere basmaya başlamış, ve geriye müşterimizden makinalar için müşteriden parayı almak kalmıştı. Son ve en kötü pürüz işte burada ortaya çıktı; Bir bütçelendirme problemi yüzünden müşteri ödemeyi istediğimiz kadar hızlı yapamayacaktı. Bu durum makinaların alınmasını aylarca geciktirebilirdi! Bunun üzerine Arpan şöyle yaratıcı bir çözüm buldu: Makinaları bizim danışman şirketimize aldırıp, müşteriye bu makinaları lease ediyormuş gibi bedelini yavaş yavaş ödetecektik. Müşteri bu ödeme şekline ikna oldu, ve geliştirme takımımız ürün seçimi ve testleri bitirdiğinde, makinalar bizi hazır bir hâlde bekliyordu. Hâtta Arpan'ın ayarladığı lease ödemeleri, faizi hesaba katılsa bile, şirketimiz için ufak bir kâr elde edecekti!

İşte iyi hava desteği budur. Kara kuvvetleri gelmeden hedef belirlenip yıkılmış, ve geriye sadece karadan yapılabilecek hamleleri yapmak kalmıştır. Proje yöneticisi tabii ki atlayıp kod yazmaya başlayamaz, ya da tasarım yapamaz, fakat projeye gereken tüm kaynakların mevcut olması için gerekli her şeyi yapar. Bu örnekten öğrenilmesi gereken diğer bir davranış, başarılı bir proje yöneticisinin en yüksek risk noktalarına nasıl hemen odaklandığını görmektir. Geliştirme makinelerin ortada olmaması, bir geliştirme takımı için *en yüksek risk* taşıyan durumlardan biridir (alternatif işletim sistemlerinde geliştirmemiz sözkonusu değildi) ve yönetici bu durumu kavramış ve işin peşini bırakmayarak tüm dikkatini ve enerjisini oraya yönelterek işi sonucunu vardırmıştır.

Gereklilik Listesi

Gereklilik listesi, yâni müşterinin yazılım takımından beklediği işlevler, proje yöneticisinin takip etmesi gereken işlerden biridir. Bu listenin ne olacağı hakkında tartışmalarda proje lideri baş rolü teknik lider ile paylaşır, çünkü gereklilik listesinde olanlar programcıların ne kodlayacağını ilgilendirir, ve programcıların ne kodlayacağı projenin ne zaman biteceğini ilgilendirir. Proje bitiş tarihine tam zamanında ve en az hata ile ulaşmak proje yöneticisinin en önemli hedefidir. Bu sebeple "gereklilik listesi" kelimelerinin duyduğu anda, proje yöneticisi kulak kabartmalıdır.

Proje yöneticisi gereklilik listesi hakkında sürekli teknik lider ile istişare durumunda olur. Yönetici hiçbir zaman tek başına bir gerekliliğin ne kadar süreceği hakkında bir kararı, teknik lider olmadan, vermemelidir. Bu kararı en iyi verebilecek kişi teknik dünyanın içinde gelen teknik liderdir.

Sabit zamanlı, sabit fiyatlı projelerde, üzerinde kaynak (değişik yeteneklerdeki programcılardan kaç tane olduğu) ve fiyat pazarlıklarının yapılabileceği bir plan dökümanının hazırlanması gerekir. Bu belgenin yazılması genellikle danışman şirketlerinde her programcıya bölüm bölüm dağıtılır. Bu belgenin bölümlerinin dağıtılması, hangi işin kimde olduğunun takip edilmesi, sonra bölümlerin herkesten alınıp birleştirilmesi ve gerekirse bir teknik yazara verilmesi proje yöneticisinin sorumluluğudur. Plan belgesindeki teknik altyapı,

ve mimari kısmı teknik lider tarafından yürütülecektir. Teknik altyapının niye plan belgesinde gereklilikler ile beraber olduğunu anlamak için 11.2 bölümüne bakabilirsiniz.

Hata Takip

Proje, kodlama safhasında ilerleyip sıra kullanıcı testleri başladıktan sonra, proje yöneticisi yeni bir rol üstlenerek *süper testçi* hâline dönüşür. Önyüzün kullanımını en iyi bilen kişi o olmalıdır, ve plan belgesinde olan herşeyin kodlanmış olup olmadığını bir yandan kontrol etmelidir. Tüm özelliklerin kodlanmış olmasından yönetici, en az teknik lider kadar sorumludur, fakat derin teknik bir alanda takılıp kalabilmesi mümkün olan teknik lidere nazaran proje yöneticinin önyüze dönük kontrolleri yapması için daha çok fırsat vardır. Bu yüzden bu görevde başrolü yönetici almalıdır.

Testçiler uygulamayı test edip hata buldukça, bunları hata takip sistemine girerler (11.3.5). Hata takip sistemindeki hataların ne kadar hızlı tamir edildiği, hangi hataların sürekli gidip geldiği, ya da bazılarının neden uzun süre tamir edilemediği gibi konuları proje yöneticisi bilir durumda olmalıdır. Önem seviyesi, aciliyeti (priority) yüksek olan hataları tamir eden programcılar ile yönetici, direk istişare hâlinde olabilir. Proje yöneticisi, önemli hatalar hakkında ilgili programcılara hatırlatmada bulunabilir, ve işin önemini onlara hissettirecek şekilde onlarla konuşabilir. Eğer ilgili programcılar moral düşüklüğü yaşıyorlar ise, belki bunun hakkında bir şeyler yapılması gerekiyordur. Hataların tamiri için teknik destek tabii ki teknik liderden gelmelidir, fakat proje yöneticisi hatırlatıcı ve kontrol edici, ve gerekirse kaynak sağlayıcı bir durumda hep hazır olmalıdır.

Tanıdığımız proje yöneticileri, önemli, ve testçiden programcıya fazlaca gidip gelen, hata no'ları ezbere bilirlerdi. Bu yöneticilerin meselâ "No 334 ne durumda?" gibi sorular sorduklarını duyardınız. Bu tavırları, projede işlerin tamamen kontrolleri altında olduğu izlenimini verirdi (ve hakikaten öyleydi), ve bu da programcıların daha dikkatli olmasını sağlardı. Psikolojik olarak birinin yaptığınız işle ilgileniyor olması, o işte daha iyi performans göstermenizi sağlar. Çünkü hata tamir edildiğinde, "hata 334 ne durumda?" sorusuna "tamir edildi" cevabı verilebilecektir. Programcı yararlılık göstermiştir ve ilgilendiği konuda bunu direk yöneticiye söyleyebilmiştir.

Sosyal Konular

Projenin sosyal atardamarı proje yöneticisidir. Geç vakte kadar çalışmaları tavsiye etmesek bile, eğer bu çalışmalar olursa çalısmadan sonra ödülleri (restoran, bar) dağıtan kişi, proje yöneticisi olmalıdır. Programcılar genç bir grup ise, bir iş gününden sonra bar'a, yemeğe götürülmelerini düzenleyecek kişi proje yöneticisi olmalıdır.

Yönetici her programcı ile iyi bir iletişimde bulunmalıdır. Onları tanımalı, ve özel hayatları hakkında çok derinliğine olmasa bile fikir sahibi olacak kadar bir yakınlıkta olmalıdır. Hiç unutmamak gerekir ki insanlar sevdikleri ve saydıkları yöneticiler için daha çok çalışırlar. Bu tekniğin tersi, müthiş bir dikta ile korku kullanarak yönetmektir, fakat korkunun sevgi/saygı kadar motive edici olmayacağını rahatlıkla söyleyebiliriz. Hiçbir insanın, korku kullanarak, o ekstra kilometreyi koşmasını sağlayamayız.

Belli bir alanda uzmanlığı olan her grup kendine has alışkanlıklar, konuşma stili ve bir alt kültür üretir. Bu kültürü tanımak proje yöneticisi için faydalı olacaktır. Meselâ benim içinde olduğum kurumsal danışman olan bizler Unix'i sayar Microsoft'u sevmeyiz. *Office Space* filmine güleriz. Resmi giyinmek bizim için en büyük cezalardan biridir. Sıkı çalışmak problem değildir, ama sıkı da eğlenilmelidir. Bu tür kültürel anektodlardan haberdar olmak proje idarecisini daha etkili yapar.

11.1.2 Teknik Lider

Projenin olmazsa olmayan kişiliği, her şeyin etrafında döndüğü, her kararın parçası olması gereken şahıs teknik liderdir. Projenin teknik sorumluluğu tamamen teknik liderin üzerindedir. Ondan başka sorumlu aranmaz.

Bu sebeple teknik lider (kısaca TL diyelim), projenin mimarisi ve kullanacağı teknolojileri seçmek ile sorumludur. *Bir projede teknik liderin kabul etmediği hiçbir teknoloji kullanılamaz.* Programcılar tavsiyeler getirebilirler ve bunlar çoğunlukla mimariyi pek ilgilendirmeyecek ufak, izole alanlar için olacaktır, fakat günün sonunda teknoloji seçimi, mimariyi etkilesin ya da etkilemesin, tamamen teknik lidere ait olan bir karardır. TL bu konuda karar verdiğinde, kararın aynen uygulanması gerekir.

Bu konuyu şöyle açıklayalım: Sorumluluk ile yetkinin aynı kişide olması gerektiği açıktır. Eğer sorumluluk var fakat yetki yoksa, bu durum teknik liderin üzerinde kontrolü dışındaki bir durumun sorumluluğunu alma stresi getirirdi ki bu hiçbir teknik lider kabul etmeyecektir. Bu yüzden bizim çalıştığımız hiçbir şirkette TL'in izni dışında teknoloji kullanılmasına izin verilmez. Programcılar seviyesinde teknolojiler hakkında bir heyecan ve "bunu da kullanalım" saflığı mevcuttur, ve bu saflığın yerine pragmatizme bırakması sadece ve sadece teknik lider seviyesine erişildiğinde kristalleşir. Bu yüzden bu karar daha tecrübeli olan teknik lidere bırakılır. Kurumsal uygulamalarda teknolojinin pür nesnesel tasarımdan bile önemli olduğunu söylemiştik, çünkü yanlış yapılan bir teknoloji seçimi, programcınıza ve projenize kayıp zaman, kalitesiz ve kötü performanslı kod anlamına gelir, ve bu sebeple bu sorumluluğu teknik liderden başkası almamalıdır. Ayrıca teknik sorunlar çıkınca yardım etmesi beklenen teknik lider olduğu için, kendi yardım edemeyeceği bir teknolojinin kullanılmasını hiçbir teknik lider istemez.

Teknolojilere karar verdikten sonra, TL kullanmasını planlandığı yeni teknolojileri testten geçirip kontrol etmeye başlayabilir. İşlerlik kontrolü (proof of con-

cept) bağlamında TL, kullanacağı teknolojiyi kontrol etmek için çalışan ufak örnekler yazabilir. Teknik lider, teknoloji ile kendini rahat hissedinceye kadar bu denemeleri yapacaktır.

Teknik lider, gerçek kodlama açısından, sadece mimariden ve (az miktarda kod içeren) geliştirme, ve deployment script'lerinden sorumlu olacaktır. Teknik liderin çok fazla gereklilik kodlaması beklenmemelidir. Çünkü teknik lider, her yere yetişebilen ve yangınları çıktığı yerde söndürebilen bir durumda olmalıdır, ama eğer kendisi çok fazla kod yazıyorsa, o zaman diğer önemli sorumluluklarını yerine getirmesi zorlaşır. Danışmanlık ağzıyla söylemek gerekirse, teknik lider kendini kritik geçiş noktasına koymamalıdır[1].

Mimari

Mimari tasarım, gereklilik listesinin hazırlandığı sırada paralelde yapılır, ve bittikten sonra gerekliliklerle beraber plan belgesinin bir bölümünde anlatılarak müşteriye sunulur. Mimari belgelendikten sonra TL, kaynak kod idare sistemini kurar, ardından planladığı mimariyi *ilk önce kendisi kullanarak* bir örnek özellik kodlar. Böylece aynı mimariyi kullanacak olan programcıların önüne bir örnek koymuş olur, aynı zamanda mimariyi hem kendisine hem de diğer programcılara ispat eder.

Açık söylemek gerekirse bir projede teknik liderin en önemli görevi, programcıları *seri üretim* aşamasına getirmektir. Seri üretimden kastımız, her gereklilik için yazılması gereken kodların kullanacağı altyapı kodlarının ve kuralların hazır olup, programcının sadece ve sadece gereklilik için kod yazacak duruma gelmiş olmasıdır. Seri üretime hazır bir takım ve programcı "şu şu nasıl yapılacak" şeklinde soruları daha az sorar, çünkü artık bir yazdığı gereklilik kodu bir önceki gerekliliğe benzer hâldedir, ve kuralları bile oradan alarak sadece işlem mantığına odaklanabilir. Seri üretim, teknoloji kullanımının tüm hatlarının ortaya konulmuş olması demektir. Teknoloji kullanımı, mimari demektir. Mimarinin tam tanımı için 8.4 bölümüne bakabilirsiniz.

Seri üretim sahfasına gelmiş olan bir takıma, ne kadar fazla programcı eklerseniz, o kadar fazla iş alırsınız. Bu açıdan bakılacak olursa Frederick Brooks'un söylediği "geç kalmış bir projeye daha fazla programcı eklemek o projeyi daha geciktirir"[11, sf. 25] sözü yanlıştır. Yanlış mimari, gereklilik kodlamasını geciktirebilir, fakat optimal bir mimari ile daha fazla programcı daha fazla iş demektir.

Geliştirme Süreçleri

Geliştirme süreçleri, kaynak kod idare sistemini nasıl kullanılacağı, hangi alt dizin yapılarının kullanılması gerektiği, ve programcıların kendi testlerini nasıl işleteceği gibi konuları kapsar, ve en az mimari kadar önemlidir.

[1] Technical lead must not be in the critical path

Kaynak kod idare (KKİ) sistemi CVS gibi bir ürün olabilir, bu ürün üzerinde tüm proje kodlarının tutulacağı bir depoyu yaratmak teknik liderin sorumluluğundadır. Depoyu yaratma işini mimariyi ve ilk gerekliliği kodlarken yapabilir. Eğer depo yarama görevini TL başka birine devretmek isterse, bunu KKİ sisteminin nasıl kurulacağını bilmediği için değil, yeni bir kişiye bu bilgiyi transfer etmek için yapmalıdır. Yâni her şartta teknik lider KKİ sistemlerinin detaylarını çok iyi bilmelidir.

Hata takip düzeni projenin başlarında kurulması mecbur değildir. Bu sistem, testçiler kurumsal uygulamayı ciddi kullanmaya başlayıncaya kadar çok gerekli olmaz, fakat o zaman gelince hata takip sistemi teknik lider tarafından hazır edilmiş olmalıdır. Hattâ şirketimizin her projesi için ortak kullandığı (ama hata verilerinin her proje için ayrı tutulduğu) bir hata takip programının olması faydalı olur. Ortak KKİ programı da aynı şekilde iyi bir fikirdir, tabii doğal olarak depo yaratma işlemi teknik KKİ bilgisi gerektirecektir.

Geliştirme ortamıyla ilgili ek bazı "davranış" kuralları yeri geldikçe teknik lider tarafından koyulabilir. Meselâ yaygın bir açık kaynak prensibi olan "her kod ekleme (check in) sonrası proje geliştirici listesine eklenen kod ile ilgili bir mesaj gönderme" kuralını teknik lider koyup (hâtta koymalıdır) takip edilmesini sağlayabilir.

Kod Kalitesi

Teknik lider, projedeki kodun kalitesinden sorumlu kişidir. Bu amaçla belli aralıklarla kod teftiş toplantıları yapılması yararlıdır. Bu toplantılardan önce TL, hangi kodların teftiş edileceğini belirler. Bu toplantılar detaylı olarak 11.3.3 bölümünde anlatılacaktır.

TL organize, gurup hâlinde teftişler yaptığı gibi bazen raslantısal ve nokta saldırısı yaparak kodun belli bölgelerine bakarak kontrol yapabilir. Bu raslantısal kontrollerden sonra eğer hoşuna gitmeyen bir nokta bulursa, programcıya bu yorumu bir tavsiye olarak götürebilir.

Kod kalitesi için önemli bir husus olan birim testlerinin kullanılmaya teşvik edilmesini teknik lider üstlenmelidir. Eğer otomize edilmiş kabul testleri var ise, bu testlerin yazılması için bir kaynak (kişi) ayırılması teknik lider proje yöneticisi ile birlikte götürür. Kabul testleri için kullanılacak programı (7.2) TL çok detaylı bir şekilde bilmelidir. Bu programın kullanılması için tüm sorular ilk başta ona gelecektir. Bu program teknik olarak kavrandıktan sonra, proje yöneticisi tüm gereken kabul testlerinin yazılma işlemini testleri yazmakla sorumlu kişiyle beraber teknik liderden bağımsız bir şekilde götürebilir.

Sosyal Konular

Programcılar ve teknik lider arasındaki iletişimin ana hatları şunlardır: Programcı, mimariye ve teknik liderin proje bazındaki koyduğu diğer genel kurallara uymaya mecburdur, ama bunun haricinde teknik lider ile programcısı

arasında bir öğrenci/öğretmen ilişkisi olacaktır. Öğrenci/öğretmen ilişkisinde öğretmen, yani teknik lider, zorlayıcı olmamalıdır. Programcı ona bir soru sorup bir teknik konu hakkında yön beklediğinde ona cevabı muhakkak verecektir, fakat eğer performans sonuçları çok fark etmiyorsa teknik lider programcıya final, "şunu yap" havasında bir yön vermekten kaçınmalıdır.

Eğer teknik lider, çok performans farkı getirmeyecek ama daha temiz bir yöntem bulmuşsa bunu programcısına "şu yaklaşımı hiç düşündün mü" diyerek getirmeli ama ısrarcı olmamalıdır. Doğru yolu programcı isterse seçmeli, istemezse kendi yolundan devam etmelidir. Mimariye bağlı kalmak çok önemlidir, teknik lider mimariden feragat etmemelidir; Bahsettiğimiz "boş alan", mimariyle gerekliliğin arasında kalan boş alandır.

Programcılar bazen, çok çetrefil bir duruma saplandılarında kendilerine güveni kaybedebilirler. Bu gibi ruhani (!) konular bizim gibi teknik konuları tercih eden insanlar için en zor konular oluyor. Fakat bu gibi durumları çok iyi idare etmemiz gerekmektedir, çünkü takımımızın motivasyonu bu idaremize bağlı olacaktır. Eğer programcı kendinden şüphe eden bir yaklaşımla lidere geliyor, ama teknik bir soruyu direk sormuyorsa, bunun tek anlamı vardır: Morale ihtiyacı vardır. Bir problemi çözmekte uzun zaman alarak takımın gözünden mi düşmektedir? Durumu iyi analiz ederek, bu soruna tek şekilde yaklaşabilirsiniz; Eğer programcı star programcılarımızdan biriyse, ona kendine güven aşılamalıyız. Bu noktada yapabileceğimiz en kötü şey, iç yapısını sadece o programcının bileceği duruma atlayıp programcıyı bir kenara iterek, problemi direk çözmeye uğraşmaktır. Çünkü 1) bunu yapamayabiliriz 2) programcımıza, ona güvenmediğimiz sinyalini göndermiş olarak motivasyonunu baltalamış oluruz.

Motivasyon faktörünü genel takım bazında da takip etmemiz ve manipüle etmemiz gerekmektedir. Takım bazında yaptığımız "burada bir sorun var", "hadi arkadaşlar" ve "işte başardık" sözlerinin takımınız üzerinde ne kadar etkili olduğunu hiç unutmamalıyız. Bu sözler ve davranışlarla, takımın ritmi tamamen teknik liderin elindedir. Çok uzun süreli "işler iyi gitmiyor" modu takım motivasyonu için faydalı değildir, bu dönemi muhakkak bir "işte başardık" zaferi izlemelidir. Bu zafer programcılara belli edilmeli ve çabalarının sonuca eriştiği ortaya çıkmalıdır. Zafer noktasına erişmek için eğer teknik lider kollarını sıvayarak programcı ile birebir oturması anlamına geliyorsa, bu yapılmalıdır. Unutmayalım ki, özellikle başlangıç seviyesi programcıları, zaten öğrenmek için oradadırlar ve çözümünüzü anlayıp öğrenmiş olmaktan, ve bir sonraki seferde kullanabilecek durumda olmaktan mutlu olacaklardır. Başarı noktasına eriştiğinizde böylece onların da motivasyonu artar.

Büyük tepe noktalarını büyük kutlama izlemelidir, kutlamaları organize etmek proje yöneticisinin görevidir, fakat takımı o noktaya getirmek (çok daha zor) teknik liderin sorumluluğudur.

11.1.3 Programcılar

Programcılar bir takımın işini üreten, onu ileri iten pistondurlar. Mimariyi hazırlamak ve programcıların önlerindeki tekniksel ve kaynaksal barikatları yıkmak her ne kadar teknik liderin ve proje yöneticisinin görevi olsa da, bu engellerden sonra nihai işi yapacak olan kişiler, programcılardan başkası değildir.

Bir programının sahip olduğu en önemli özelliklerden biri, sebat etmektir (persistence). Program yazarken problemler, teknoloji bazlı olsun olmasın her zaman ortaya çıkarlar, ve bu olduğu zaman programcı yılmayan, problemin üzerine tekrar tekrar giden, ve bâzen değişik açılardan yaklaşmayı deneyerek sonuca varabilen türden bir insan olmalıdır.

Programcının yeni teknolojileri rahat öğrenebilmesi onun esnekliğini ve seçeneklerini arttırıcı faktörlerden birisidir. Kurumsal programcılıkta ve özellikle kurumsal programcılık yapan danışmanlık şirketlerinde, her projede yeni bir teknoloji kullanılması kaçınılmazdır, bu sebeple yeni teknolojileri anlayabilme ve kullanabilme programcı için vazgeçilmez bir yetenek hâline gelir.

Aranan bir diğer özellik, enerjidir. Amerika'da birçok müstakbel programcının mülakatını yaparken baktığımız ana parametrelerden biri hep bu oluyordu. Enerjiyi anlamak için programcının nasıl davrandığına, teknoloji hakkındaki heyecanına ve problemlere yaklaşırken ki tutumuna bakabiliriz.

Programcıları her projede genellikle acemi (junior) ve usta (senior) seviyesinde iki kategoriye ayırabiliriz; Bu kategoriler programcıların çalıştığı şirket içinde HR (İnsan Kaynakları) tarafından verilmiş ve periyodik olarak bir sonraki seviyeye geçiş (yükselme) kriterleri için kontrol edilen payeler olabilirler. Bu durumda junior ya da senior olmanın *maaşsal* etkileri olacaktır. Bu işin HR tarafıdır, biz teknik olarak bu kelimelerin anlamıyla ilgileneceğiz.

Junior Programcılar

Junior, İngilizce giriş seviyesi ya da acemi anlamına gelir. Bir programcı ilk projesinde, ve ilk işinde bu seviyede olacaktır. Junior bir programcının, projede kullanılan bilgisayar dilini ve veri tabanı kavramlarını bilmesi beklenir, ama geri kalan çoğu teknolojiyi bilmiyor olabilir. Ayrıca tasarım düzenleri, mimarileri takip etmek konusunda yardıma ihtiyacı olacaktır. Junior programcı teknolojileri kendi başına öğrenirken zorluk çekebilir. Fakat enerjik ve akıllı bir programcıysa, yol gösterilince istenilen yere rahatlıkla gidebilen bir kişidir, ve teknik liderden ya da senior programcılardan yardım alınca kendi işini halledecek duruma gelir.

Junior programcılar teknolojiyi öğrenirken, bir yandan öğrenmeyi öğrenirler. Bir teknolojinin püf noktalarını kapan bir junior progamcı tipik olarak yeni teknolojileri daha rahat anlamaya başlar. Bu şekilde ilerleyerek senior düzeyine çıkma yolunda adımlar atmış olacaktır.

Projemizde bir mimarinin olması, en çok junior programcılar için yararlıdır, çünkü mimarinin yol gösterici özelliği onlar için iyi bir rehber hâline gelir.

Yaygın yapılan hatalara karşı kuralları iyi oluşturulmuş bir mimari, junior programcıyı tuzaklardan koruyacak, ve sadece işlem mantığı koduna odaklanmasına izin verecektir. Ayrıca mimariye bakarak ve onunla ilgili sorulara sorarak, teknik liderin teknoloji ve yazılım geliştirme felsefesi hakkındaki bilgisini almış olur.

Junior programcı, öğrenmek için orda olsa da, günün sonunda iş yapması beklenir, ve ona harcanandan daha fazla zamanı, yapılmış iş olarak geri vermelidir. Eğer çok fazla soru soruyorsa ve iş yaparken harcadığı zaman, ona öğretilmekte harcanan zamandan daha fazla ise, bu bir eksikliktir, ve programcı uyarılmalıdır.

Senior Programcı

Senior programcı yeni teknolojiyi öğrenme, ve bir problemi çözme konusunda en becerikli programcıdır. Senior programcının en iyi tanımı, "dışarıdan sıfır (ya da çok az) yardımla, bir problem öbeğini kendi başına çözen programcı" cümlesidir. Teknik liderden bir çıta aşağı olan bu programcının teknik liderden farkları, mimari hazırlamak, Unix ve büyük sistemlerde uygulamaların çalışmasını anlamak, performans optimizasyonu, geliştirme ortamı hazırlama, planlama, ve sadece tecrübeyle alınabilecek çok ince teknolojik detaylar konularında olabilir. Bunun haricinde senior programcı, teknoloji seçmek, kulanmak, tasarım yapmak ve nesnesel modellemede oldukça ileri bir durumdadır. Kendine verilmiş öbek içindeki kodun kalitesini üzerine alabilir, birim test tekniklerinden haberdardır. Hiçbir teknolojinin kuruluşu ve kullanımı onun için problem değildir, en yeni teknolojiyi alıp işler hâle getirebilir. Bu teknolojiyi kullanarak en temiz mimariyi kuramayacak olabilir, fakat *çalıştırmada* problem çekmez.

Senior programcı mimari takip etmekte iyidir, ve mimaride bazı boşluklar keşfederse bunları teknik lidere söylemelidir. Sonuçta senior programcının bir sonraki kariyer basamağı teknik liderlik olacaktır, bu yolda giderken kendine gereken ek yetenekleri liderden almaya uğraşmalıdır. Teknik mimarinin tasarım seçimlerini anlamak bu ek yeteneklerden biridir.

Teknik liderlik yolunda senior programcıya yardımcı olmak için takımdaki teknik lider, gelecek vaadeden senior programcıya kendi görevlerinden bazılarını yükleyebilir. Meselâ plan dokümanındaki mimarinin belgelenmesi senior programcıya aktarılabilir, ya da geliştirme ortamının kurulması (CVS), ya da hazırlanmış mimariyi kullanan "ilk gerekliliğin" kodlanması senior programcıya verilmiş bir görev olabilir.

Junior programcılar takıldıkları yerler hakkında soru sormak için senior bir kişiye gelebilirler. Bu sorular, projenin mimarisi, karşılaşılan bir problem ya da hata bulmak (troubleshooting) konularında olabilir, senior programcı kendi işleri geriye düşmediği sürece bu sorulara cevap verebilir. Ama esas yangın söndürücü teknik liderdir, belli bir süre harcandıktan sonra problem halâ çözülmediyse teknik lider konuya dahil olmalıdır, çünkü üstüne en az kodlama sorumluluğunu almış olan teknik liderdir (mimari kodlama projenin

başında yapılmış ve bitmiştir) bu yüzden yardım etmek için en fazla zamanı olan tek kişi o olacaktır.

11.2 Planlama

Bir işe başlamadan önce plan yapmanın önemini general ve eski ABD Başkanı olan Eisenhower çok güzel tarif eder: "Bir savaş sırasında planların genellikle işe yaramadığını gördüm, ama yine de planlamak, bir savaş için en vazgeçilmez eylemdir"[13]. İnsanoğlunun evrimi sırasında sürekli olarak bilgi işleyebilme kapasitesi artmıştır[12, sf. 138] ve bilgi işleme, onu kaydetme, bilgiden öğrenmek anlamına geldiğine göre, bu yeni kapasite öğrenim, tecrübe ve öngörü olarak insana ileriyi tahmin yeteneğini arttırmasını sağlamıştır. Bu artış, *planlarımızın* daha doğru olacağı anlamına gelmiş ve direk sonuç olarak hayatta kalma şansımızı arttırmıştır.

Planlar, tecrübeden destek alan doğru öngörüler üzerine kurulursa çok başarılı olurlar. Fakat düzgün öngörüler üzerine kurulmamışlar bile olsalar, planlar başarısız olacakları ana kadar bir gidiş yönü sağlarlar, ve işlememeye başladıkları anda plan ile gerçek durumun farkını iyice belirgenleştirecekleri için, Eisenhower'ın dediği gibi, ortada olmaları vazgeçilmez kaynaklar hâlini alırlar.

%100 öngörü, engin teknik ve yönetim tecrübesi gerektirir, ve bu tecrübeyi bu kitaptaki değişik bölümlerde aktarmaya çalıştık. Geriye tek bir konu kaldı: *Planlama planını*, yâni bir planlama sürecinin takip edeceği bazı faydalı kuralları vermemiz gerekiyor. Böylece planlayıcılar üzerinde hatırlatıcı noktalar yaratarak, planlamada yapılabilecek mekanik hataları en aza indirgemeyi amaçlıyoruz, ayrıca, bu plandan çıkması gereken belgeleri ortaya koyarak, profosyonel bir yazılım projesinin müşterisi ile arasında önemli olan iletişim noktalarını göstermeyi amaçlıyoruz.

Planlamanın kaç ay ileriye doğru yapılması gerektiği, üzerinde oldukça düşünülmüş ve tartışma yapılmış diğer bir konudur; Bu konu hakkında sektörün seçimini sunmaya çalışacağız.

11.2.1 Ne Kadar İleri Görelim?

Bu sorunun müşterimiz (iş sahibi, projeyi siparış vermek isteyen kişi) açısından, ve yazılım ekibi için ayrı ayrı cevaplanması gereken iki bacağı vardır. Yazılım ekibi için bu cevap "görebileceğimiz kadar ileriye" olmalı, müşteri için ise, "en çok kâr getirmesi mümkün olan an az yazılıma yetecek kadar" olmalıdır. Bu iki nokta arasında hangi tarafın daha ağır basacağını yazılım takımının istediği fiyat belirleyecektir. Fiyat, (*projedeki eleman kalitesi ve bunların sayıları*) x (*kalite birim fiyatı*) x (*projede harcanan gün*) denkleminden çıkacaktır. Eğer müşteri kârını mümkün olduğu kadar arttırıp tam özellikli bir yazılım istiyorsa ve bunun bedelini ödemeye hazırsa, yazılım takımı görebileceği kadar ileriye giden bir proje kapsamını müşteriye hazırlayıp sunmakta serbesttir.

Yazılım takımı niye "görebileceği kadar ileriyi" görmeye uğraşmalıdır? Buna karşı bir soruyla cevap verelim: Eğer bunu yapabiliyorsak, niye yapmayalım? Bir işi daha iyi yapması beklenen bir yazılım, kendi içinde bir bütündür ve sonradan başımıza dert olabilecek durumları öngörebiliyorsak, bu bütün içinde olabilecekleri öngörebilmemiz gerekir, hâtta öngörmeliyiz. Çünkü yazılımın kalitesi ve kodun sonradan değişmesini engellemek bunun faydası olacaktır.

Bazı planlama süreçlerine göre, "uzağı görebiliyorsak bile, görmemek daha iyidir", çünkü "o zamana kadar müşterinin fikri de, etraftaki başka şartlarda değişmiş olacaktır". Onlara göre "zâten tasarımı düzgün yapmamız mümkün değildir". Ayrıca "müşteriye ne yapacağını söylememeliyiz, çünkü o işini en iyi bilendir".

Bu düşüncelere katılmıyoruz, çünkü cevap vermeye sonran başlamak gerekirse, danışmanlık dünyasının önemli bir kanununu şöyle der: *Müşteri ne istediğini bilmez.* Yâni, bir projenin başında en parlak, renkli, acaip önyüz ve veri yapısı tasarımlarını gerektiren gereklilikleri önünüze getiren müşteri olacaktır, fakat teknolojiyi, hâtta bâzen kendi iş yapılarını ve o yapının ne olması gerektiğini çoğunlukla anlamadıkları için, gerçekçi seçimler yapamazlar. O zaman, bir danışman müşteriye ileriye dönük, sürekli değişmesi gerekmeyecek (sağlam) bir iş süreci dahilinde bir IT sistemini tavsiye edecek durumdaki en uygun kişidir. Müşteriniz, bu tavsiyeyi alınca mutlu da olacaktır.

Tasarımı düzgün yapamama, sadece kullanması gereken teknolojileri iyi bilmeyen takımlar ve teknik liderler için geçerlidir, bu yüzden pire için yorgan yakılmaması en iyi seçimdir. Teknik mimar zâten üzerinden optimal bir mimari kuramayacağı bir işe kalkışmamalıdır, bunun tersi intihardır. Teknik lider böyle bir işe girişirse, projenin bitiş tarihini tehlikeye atmış olur, ama bu sebeple her teknik lideri bilgi seviyesi ne olursa olsun mimari tasarlamaktan men etmek, %99 şartlarda kaliteye sebebiyet verebilecek planlama aşamasından bizi mahrum bırakmek anlamına gelecektir. Eğer teknolojiyi anlamayan teknik lider ve ona bu görevi vermekte sakınca görmeyen danışman şirketinin en yüksek teknik görevlisi (CTO) içine girdikleri riskin farkında değillerse, aldıkları risk geri tepecektir, ve tepmelidir; Tahmin edilen sonuç, daralıp işten çıkan programcılar, müşteriye zamanında yazılım verememekten lekelenen bir ün ve kayıp iş ve zaman olacaktır. Kısacası piyasa kanunları durumu hâlledecektir.

Müşterinin fikrinin değişme argümanını herhalde (aynı anda cevaplamış) olduk, çünkü müşteriye, değişmeyecek fikirleri göstermek projenizin planlama takımının görevidir. Müşteri tabii ki planlama dökümanı imzalandıktan sonra çok gerekli değişimleri yapabilmelidir, ama bunun cevabı ayrı bir sipariş değişim belgesi (change order document) ve sürecidir. Bu değişim bir mini planlama evresinden geçer, ve sonucunda ayrı fiyatlandırması gereken bir belgeye sebebiyet verecektir.

Konjektürden kaynaklanan değişimler hakkında ise yapacak bir şey zaten yoktur. Eğer sırf bunun için rüzgar her değiştiğinde başka yöne gidebilen bir planlama süreci kullanırsak, o zaman habire projeye yeniden başlamış oluyoruz demektir, ve bu, uzun vadeli planlamadan elimize geçecek hiçbir kazanımdan

istifade edemiyoruz demektir. Burada esas sorulması gereken, eğer konjektür değişmiş ise, niye belli bir yönde gidecek bir yazılımın ve iş stratejisinin daha baştan ele alınmış olduğudur. Bir yerde bir hata yapılmıştır, ve bu hata yazılım projesinin çok üzerinde, stratejik bağlamda yapılmıştır. Eğer bu hata yapılmışsa, o ana kadar yazılmış kod atmaktan başka çare yoktur. Bu kod kullanılabildiği kadarıyla yeni proje yönünde kullanılmaya uğraşılabilir. Ama şundan emin olmamız gerekiyor ki, konjektür çok fazla değişmiş ve bu iş planı üzerinde çok büyük değişikliklere sebebiyet verdiyse, üstünde çalışacağımız şey *yeni bir projedir*. Bu, gereklilik toplama, mimariyi tasarlama, vs. gibi tüm basamakların silbaştan tekrarlanması anlamına gelir.

11.2.2 UGB

Kitabımızda, bir yazılımın planlama odağı olarak tek bir döküman önereceğiz: Uygulama Geliştirme Belgesi (UGB). Bu belgenin isminin "gereklilik belgesi" olmamasının sebebi, içinde gerekliliklerin listesine ek olarak teknik mimarinin de işlenecek olmasıdır. UGB pür tasarım, ya da pür gereklilik belgesi değildir, her ikisinin birleşimidir. Hem üzerinden ödeme şartlarını tanımlayan bir kontrakt imzalanabilecek bir belge, hem de programcıların alıp (mimariyi gözönüne alarak) gereklilikleri geliştirmeye başlayabilecekleri bir kayıttır. UGB kalemlerini şöyle sıralayabiliriz:

- Gereklilikler
  - Gri ekranlar
  - Ekranlar arası geçiş diyagramı
  - Düzyazı olarak detaylanmış gereklilikler
- Teknik mimari
  - Yazılım mimarisi
  - Sonuç donanım mimarisi
  - Veri tabanı şeması

UGB öncelikle bir giriş bölümü içermelidir. Bu girişte, projenin misyonu ve çözmeye çalıştığı sorunlar özet olarak anlatılmalıdır. Projenin teknik önkabulleri paylaşılmalı, ve sistem hakkında genel bir fikir verilmelidir.

Gri ekranları bölümünde, form'ları (veri giriş ekranlarını) detaylı alanlarıyla beraber belgelememiz gerekecektir. Gri ekranlar, isimlerini renksiz ve sıkıcı olmaktan alırlar, çünkü tek amaçları sistemizin veri alışveriş gerekliliklerini belgelemektir. Gri ekranlara bakarak tüm veri giriş gerekliliklerini görmek mümkün olmalıdır. Unutmayalım ki bu gri ekranlarının, veri taban şeması tasarımına etkileri olacaktır.

Gri ekranları kağıda (ekrana) dökebilmek için, diyagram üretebilen bir program kullanmamız gerekir, piyasada birçok gri ekran hazırlayabilen ürün mevcuttur. Bu ürünler ile ekranları ekran alanlarını gösterip, belgemizin içine şekilleri grafik olarak ekleyebiliriz.

Ekranlar teker teker tamamlandıktan sonra, kuşbakışı seviyesinde (alan detaylarını göstermeden) sadece hangi ekrandan hangisine geçilebileceğini gösteren bir büyük diyagram hazırlanmalıdır. Bu diyagram, bir ekrandan diğerlerine yapılması mümkün olan tüm geçişleri göstermelidir, ve bütün olarak bakıldığında bir ağ gibi gözükecek diyagram, tüm geçişleri göstermiş olacaktır. Böylece üst seviyeden bakarak uygulamada nereden nereye gidebileceğimizi anlamamız rahat olur.

Ardından, düzyazı olarak anlatılacak gereklilikler bölümü gelmelidir. Her gereklilik ayrı bir bölüm (section) içinde, düzyazı olarak anlatılmalıdır, ve bu anlatım mümkün olduğu kadar detay içermelidir. Bu belge üzerinden kodlama yapacağımızı unutmamamız gerekir. Belge hazırlanırken birçok git/gel, istişare, yeniden konuşma ve al/ver (tradeoff) kararı yaşanacaktır, çünkü müşteri kendi için iyi olacak gerekliliklere karar verme basamaklarından geçmektedir, ve bu sağlıklıdır. Belgeleme süreci, yapması istenen şeyi, yâni iletişimi faydalı yönde kanalize edici, ve en sonunda kaydedici rolünü oynamaktadır. Planlama takımı bu noktada müşteri için uygun tavsiyeleri ortaya koymalı, ve gereklilikleri müşteri için yararlı ve teknik olarak yapılabilir (feasible) yöne doğru itmelidir.

Gereklilikleri toplarken, düzyazı ya da ekranlar formatında, planlama takımının fiyatlandırma düşünmemesi iyi olur. Unutmayalım ki müşteri, eğer kendi önceliklerini iyi anlamışsa, "en çok kâr getirecek en az yazılımı" düşünüyordur, ve bu yönde açık tavsiyeler almak onun için faydalıdır, ve fiyat ancak bu gereklilik listelemesi bittiğinde ve teknik mimari ortaya çıktığında o projenin ne kadar süreceği (programcı sayısını ve kalitesine oranla tabii) belli olacaktır. Bu noktadan önce, uygun gerekliliklerin listeye eklenmesi zaman argümanıyla engellenmemelidir. Bir özelliğin dahil edilmesi ya da edilmemesi, yapılabilirlik, ve işe faydası çerçevesinde olmalıdır. Müşteri nasıl olsa bir fiyat gördüğünde, ve o fiyat yüksek gelirse kendiliğinden gereklilik traşlamaya başlayarak, daha az önemli özellikleri atmak isteyecektir.

Teknik mimari, gereklilikler toplanırken arka planda teknik lider tarafından yavaş yavaş pişirilecektir. TL eğer bir karara yaklaştıysa, mimariyi paralelde, gereklilikler toplanırken bile belgelemeye başlayabilir. UGB'nin mimari alâkalı bölümünde hem yazılım mimarisi hem de sonuç ortamının yapısı (donanım) ortaya çıkmalıdır, yâni ölçekleme (scaling) bu aşamada yapılacaktır. Bu amaçla teknik lider müşteriye, sistemin kullanım seviyesi hakkında sorular sormalıdır. Sistem, eşzamanlı kaç kişi, günde toplam kaç ziyaret için hazırlanmaktadır? Kullanım düzenleri (access patterns) nasıldır; Her dakika, saat, gün için eşit midir, yoksa belli bazı zıplama noktaları (peak) olacak mıdır (meselâ bir çiçekçi sitesinin sevgililer gününden bir gün önce yüksek zıplama yapması, ama diğer günler oldukça boş kalması gibi).

Veri tabanı şeması ise, gri ekranların bir sonucu olarak ortaya çıkacaktır.

Gereklilikler ve Tasarım

Gereklilikleri düz yazı olarak yazmanın önemini bir kere daha vurgulamak istiyoruz, çünkü gerekliliklerin en önemli görevi müşterimiz ile aramızda bir iletişim formatı olmaktır, ikincil olarak, geliştirme takımına ne yapacağını anlatmaktır.

Bazı planlama yaklaşımlarında, gereklilik belgesini, koda yakın bir şekilde ve kodun ne yapacağını birebir tarif etmeye uğraşılması tavsiye edilir. Bu tamamen gereksiz, ve zaman israfı sayılabilecek bir işlemdir, çünkü kodlama, o işi yapan programcıların işidir ve kod miktarı bakımından en fazla yer tutan bölüm olan gereklilik kodlarının iki kere yazılmış olması (birincisi belgede diğeri kodda), hem uyum açısından hem de efor açısından zarar verecektir. "Analizcinin programcıya gerekliliği hata bırakmayacak şekilde târif etmesi" iyi bir şey olarak pazarlanabilir, fakat kaynak açısından analizci, en fazla birkaç kişi olabilir, ve bu birkaç kişi neredeyse uygulamayı belge içine *yazarken*, programcıların kenarda durup bunun bitmesini beklemeleri gerekecektir. Hız, günümüz iş dünyasında en önemli faktör olduğu için, programcıların kodlamaya gereklilikleri okuyarak ve mimariyi dikkate alarak başlaması, en uygun seçim olacaktır.

11.2.3 Kaynaklar, Zaman, Kalite, Fiyat

UGB hazırlandıktan sonra yazılım takımının yönetim grubu, teknik liderin kontrolünde, kaynak ayırımı, zaman kararı ve fiyatlandırma işlerine başlayabilir. Kaynak ayırımı şirket içinden programcı (kaynak) bulmak, fiyatlandırma ise müşteriden istenecek meblağa karar vermek için yapılacaktır. Danışman şirketleri için kaynak ayırımının fiyat üzerinde etkileri vardır, çünkü danışman şirketleri aslında kendileri için çalışan programcıların *zamanını* müşterilerine satmaktadırlar. Eğer bir yazılım projesi müşterinin şirketindeki programcılara yaptırılacaksa (in-house development) o zaman bu aşamada tek yapılması gereken kaynak ve zamana bakmaktır, çünkü programcıların maaşı nasıl olsa müşterinin kendisi tarafından verilmektedir.

Danışman şirketlerde, kaynak planlaması açısından yapılması gereken tek şey teknik liderin kendi şirketi içinde bir kaynak kapma savaşı vermesidir. Eğer danışman şirkette birden fazla proje üzerinde çalışılmakta ise, normâl olarak herkes en iyilerle ve en yeterli sayıda insanla çalışmak istediği için, kaynaklar üzerinde diğer projelerle girişilen kıran kırana bir kapışma savaşı yaşanabilecektir. Dot com yıllarında danışmanlık yapan bir şirketimizde görevlendirme toplantıları (staffing meeting) kanın gövdeyi götürdüğü (!) toplantılar hâline gelmekle ünlüydü.

Bu kapışma sonucu teknik lider belli sayıda adamları projesine katar. Şimdi bu adamların sayısı ve yetenek seviyeleri üzerinden, bir zaman ve ona göre bir fiyat kararı verilebilir. Bu irdeleme "programcı X, Y, Z arasından, X senior ve Y, Z junior ise, A gereklilik listesi 4 ayda biter" gibi bir çizgiyi izler. Danışman şirketi her programcı tipi için bir fiyat çizelgesi hazırlamıştır, ve X, Y, Z'nin harcayacağı 4 ay için müşteriye bir fiyat biçilir.

Ve bu, planlamanın *olması gereken* şeklidir. Bu yaklaşımın tersi, yazılım üst şirket yönetiminin bir projeye çok muhtaç olup imkansız bir bitiş zamanı için müşteriye direk söz vermesi, ve bir de ek kaynak vermeden teknik lidere projeyi bir şekilde bitirmesini istemektir. Kaynaklar ve zaman yakın ilişkide olan iki vanadırlar; Birinin değişimi ötekini etkiler. Ama bu iki vana değişmiyorsa, üçüncü bir vana olan *kod kalitesi* aşağı düşecektir. Test edilmeden bittiği deklare edilen kod parçaları, test evresine girildiğinde hâla geliştirmesi devam eden yazılım parçaları bu garabetin baş göstergeleri sayılırlar. Bunun yapıldığını sektörde ne yazık ki pek çok kez gördük. Umarız bu alanda iyiye doğru bir gidiş yaşanır.

Bilinçli bir şekilde bir vana olarak göstermediğimiz diğer bir oynama noktası (dokunulmamasını umarak), programcının iş saatinin arttırılma vanasıdır. Vana olarak gösterilmemiştir çünkü programcı saatinde oynama yapmak, kalitesizlik ile toplanarak programcının stresini arttırıcı bir rol oynayabilir. Psikolojik olarak programcılar proje daha başlamadan geç kaldıklarını hissederler ve sürekli kovalama oyunu oynadıkları bilinciyle tüm potansiyelleri ile çalışamazlar. Belli bir yaş seviyesindeki programcılarla bu oyunu oynamak mümkün olabilir, fakat bir süre sonra bu oyunu oynayacak programcı bulmakta zorlanabiliriz. Talebin olduğu bir piyasada, programcılar daha az stresli çalışabilecekleri yerlere geçeceklerdir.

11.3 Geliştirme Ortamı

Geliştirme ortamı, kaynak kod deposu, test makinaları, test veri tabanı, hata takip sistemi ve programcıların makinalarını içeren sistemler ve ürünler topluluğudur. Bu bölümde, geliştirme ortamını oluşturan sistemlerin kurulması ve aralarındaki ilişkinin nasıl olması gerektiğini işleyeceğiz.

Her projede, üzerinde kaynak kod deposu (meselâ CVS'in) çalışacağı bir makina gereklidir. Bu makina, herkesin geliştirme ortamının bağlanıp en son kodu alıp geri verebileceği bir merkezi yer olmalıdır. Genellikle bu makinanın uygulama servisi ve veri tabanı test makinalarından ayrı olması tavsiye edilir. Bu makinanın üzerindeki kod depolarının yedeklenme işi düzenli olarak yapılmalıdır.

Programcıların geliştirme makinaları (hesapları), merkezi depo'ya işaret eder, ve tüm kod alma/verme işlemleri, bu depo üzerinden yapılır. Test için ortaya çıkartılacak sürüm, müşteriye projenin sonunda gönderilecek final ürün, hep bu merkezi depodan çekilen kodlar baz alınarak yapılmalıdır. Tüm kodun en son hâli, sadece kod deposunda olmalıdır.

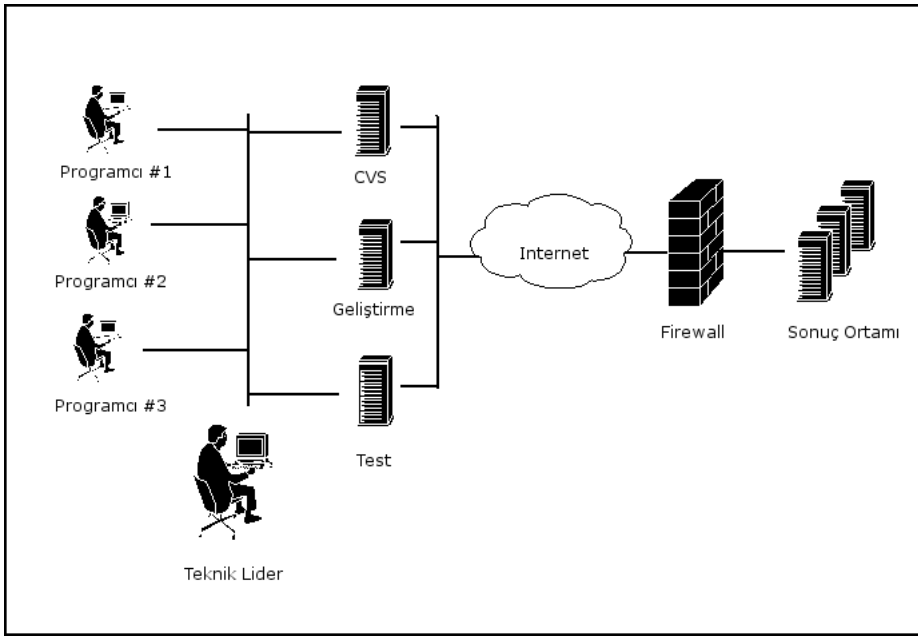
Programcıların hususi geliştirme ve test ortamı için iki türlü fiziksel yapı mümkündür:

- Unix makinalara login ederek, onlar üzerinden geliştirme ve test
- Windows makinalarda geliştirme ve Unix üzerinde test

Bu anlatım, dolaylı olarak şu önkabulu içerir: Her programcıya ayrı bir test ortamı gereklidir. Bu test ortamı, genellikle, Unix üzerinde olur, çünkü sonuç (production) makinası bir Unix makinasıdır. Test ortamının sonuç ortamına mümkün olduğu kadar benzemesi, üzerinde kodun işleyeceği final ortamda çıkabilecek hataları önceden yakalama bakımından çok önemlidir. Projemizde işletim sisteminden işlerkod seviyesinde bağımsız olan Java dilini kullanıyorsak bile, bu durum böyledir, çünkü baytkod aynı bile olsa, JVM'ler ayrı işletim sistemlerinde çalışmaktadır. Bir Murphy kanununa göre "ters gidebilecek herşey ters gidecekse", o zaman tüm potansiyel terslikler (bir JVM'de çalışan kodun öteki işletim sistemindeki JVM'de çalışmaması ihtimali) önceden test edilmelidir.

Eğer Unix üzerinde geliştirme yapıyorsak, bu birden fazla programcı aynı makinada kodlama yapıyor demektir, ve bu sebeple A.4.2 bölümünde anlatıldığı gibi herkesin uygulama servisi (meselâ JBoss) için ayrı ayrı port'lar kullanmamız gerekir. Programcılar ayrı Unix hesapları ile merkezi geliştirme makinasına girecekler, ve oradaki ortamları (`javac`, editör, vs) ile geliştirmeyi yapıp, depoya o ortamdan son stabil kodu göndereceklerdir.

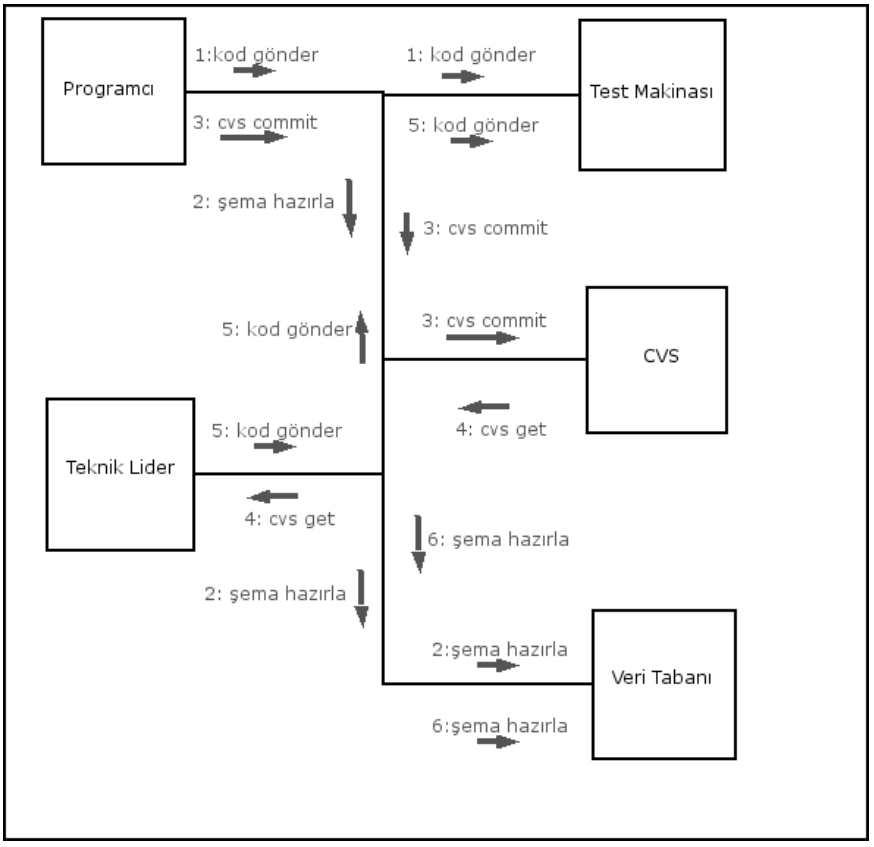
Eğer geliştirme Windows, test ortamı Unix ise (yâni potansiyel JVM farklılıklarından çıkabilecek hataları göze aldıysak), geliştirme ortamında port ayrımı yapmamız



Şekil 11.1: Geliştirme Ortamı Fiziksel Yapısı

gerekmez çünkü herkesin makinası ay**rı**dır ve bu yüzden bu makinalar aynı port'ları kullanabilirler. Fakat test makinası merkezi olacaktır, ve bu sebeple A.4.2 bölümündeki port ayar değişikliğini test makinası için yapmamız gerekir.

Unix üzerinde, geliştirme ya da test amaçlı birçok kullanıcıyı aynı makinada desteklemek için teknik lider şu prensibi takip etmelidir. Geliştirme ve test amaçları için *her programcıya*



Şekil 11.2: Geliştirme ve Test Sırası

- Bir Unix kullanıcısı (kendi adında)
- O kullanıcı üzerinde işleyen bir JBoss sürecine ayrı bir port
- Uygulamanın kullanacağı ayrı isimli bir veri tabanı şeması (kullanıcısı)

tahsis edilmelidir. Tüm bu birimlerin herkes için **ayrı** olması çok önemlidir! Yoksa, aynı anda aynı makinada geliştirme ya da test yapmanın imkanı yoktur.

Ayrı port/kullanıcı/süreç sistemi, tüm programcılara ve testçilere birbirlerinden izolasyon sağlayarak, herkese kendi geliştirme/test ortamını verir. İnsanların bir test yapmak için birbirinden izin istediği, ya da istemeden direk kullanmaya başlayarak birbirlerinin sonucunu etkilediği bir sistem optimal-altı, hatalara açık ve hiçbir şirketin takip etmeyeceği bir yöntemdir.

Veri tabanı, fiziksel olarak, uygulama servislerinin işlediği test Unix makinasında olabilir.

11.3.1 Test Ortamına Deployment

Geliştirme süreci içinde uygulama belli bir olgunluğa eriştiği zaman test için en son kodu test makinasına gönderme işi teknik lider tarafından özel script'ler kullanılarak yapılır (bölüm 6.1). Eğer programcılar Windows'da çalışıyor ve hepsinin ayrı bir Unix makinasında test hesapları var ise, bu script'ler zaten programcıların kendi test'leri için sürekli kullanılmaktadır, teknik lider sadece bu kod gönderim işlemini *testçi için ayrı* bir Unix hesabına, JBoss port'una ve veri tabanı kullanıcısına doğru yapacaktır. Kodların geliştirme ve diğer ortamları arasındaki hareketi, Şekil 11.2 üzerinde gösterilmiştir.

Ek olarak teknik lider, kod depo sisteminde gönderilen kodun gününü belirten bir etiket atılmalıdır, çünkü bu sürüme sonradan dönmek gerekebilir.

Bu yapıldıktan sonra, artık en son kodları içeren test hesabında üzerinden test servisi başlatılır, ve kabul testlerini işletmekle görevli kişi bu servis üzerinde testlerini yapmaya başlar. Bulunan hatalar hata takip sistemine girilecektir.

Projenin kaynak kod dizin ağacında bulunması gereken önemli script'ler,

- Kod göndermek
- Veri tabanını sıfırdan kurmak
- Test verisi yüklemek

için yazılmış script'lerdir. Kitabımızın örnek projelerinde veri tabanı script'leri `src/sql` altındadır. Bu script'ler `.sql` dosyalarında bulunurlar ve test makinasına her kod gönderildiğinde test makinasında işletilerek en son kodun en son şema ile çalışması sağlanmalıdır. Bu beyanın bir yan sonucu olarak, şemanın en son hâlinin her zaman `sql` dosyalarına yazılması gerekliliği sonucuna varırız; Şema ile kodun arasında olması gereken birebir ilişki barizdir, ve bu iki tarafı senkron halde tutmaya dikkat ederek, en son kodun *eski bir şema* üzerinde test edilmesini engellemiş oluruz. Hattâ teknik liderin koyacağı önemli davranış kurallarından biri bu olmalıdır: "Her şema değişikliği ilk önce `src/sql` altındaki script'ler içinde yapılmalı ve bu değişikliği kullanan kodlar ile birlikte KKİ'a aynı anda eklenmelidir (check-in)". Şema değişiği *hiçbir zaman* test veri tabanı üzerinde direk olarak yapılıp, orada bırakılmamalıdır. Şema, kodumuzun ayrılmaz bir parçasıdır.

Kod deposu, uygulamanın çalışması için gereken her türlü metni içeren bir merkez olmalıdır.

Kod gönderme için gereken script, meselâ `deploy.sh` adında ve `src/¬ scripts` dizini altındaki olabilir. Bu script parametre olarak deploy edilecek makinanın ismini, ve Unix hesap ismini alabilir. Bunları kullanarak, önce kod deposundan en kodu alacak, Ant ile derleyecek, `scp` ile derlenen kodları test ortamına gönderecek, ve test veri tabanındaki şemayı silbaştan yaratıp (`CREATE TABLE` komutları ile) test verisini ekleyecektir.

11.3.2 Kaynak Kod İdaresi

Kaynak kod idaresi (KKİ) sistemi bir projede çalışan tüm programcıların yazdığı kodun ve script'lerin metninin muhafaza edildiği yerdir. Bu merkezi yere olan gereksinimimiz, birçok programcının aynı anda aynı yerdeki kod üzerinde yaptığı değişiklikleri gönderebilme ihtimalinden ortaya çıkmaktadır; Eğer aynı dosyayı birden fazla kişi değiştirmişse, KKİ bu çakışmaları çözecek teknolojiye sahiptir.

Ayrıca KKİ tarihi bir arşiv niteliğini taşır. Bu akıllı depo, bir kod parçasının üzerinde yapılmış olan tüm değişiklikleri hatırlayabilir. Eğer kod parçası X'in 2, 3, 10 gün önceki hâline dönmek istiyorsak, bu KKİ sistemleri için çok basit bir işlemdir. Bunu yapmak projelerde çok sık olmasa da, arada yapılması gerekebilir ve gerektiği zaman KKİ tarafından desteklenmesi istenir.

Diğer aranan KKİ özellikleri kaynak kodları yedekleme, belli bir tarihteki tüm kodlara işaret atarak o günün kodunu dondurma (labeling) ve dallanmak (branching) gibi özelliklerdir. Bunların CVS ile nasıl yapılacağını ileri bölümlerde işleyeceğiz.

CVS, ürün olarak piyasadaki en rahat erişilebilir, fiyatı uygun (bedava) ve hakkında yaygın kaynak olan KKİ sistemidir. Son zamanlarda Subversion adlı açık yazılım ürün de popülerlik kazanmaya başlamıştır. Açık yazılım projelerinin neredeyse tamamı CVS kullanır, bunda CVS'in programcılara kopuk (disconnected) şekilde çalışmasına verdiği destek büyük rol oynamıştır (fiyat ikinci bir etkendir).

Kopuk çalısma, sadece İnternet üzerinden iletişim kuran ve değişik bölgelerden programcıların beraber çalışmasını çağrıştırır, fakat kopuk çalışma aynı bölgede, aynı odada çalışan programcılar için bile istenen bir özelliktir. CVS ile bir dosyayı değiştirmek için kitlememiz gerekmez, ve başkası ile aynı anda yaptığımız değişikliğimizi CVS'e aynı anda geri koymaya uğraşırsak CVS bir çakışma (merge conflict) mesajı alırız. Böylece CVS ikinci değişiklığı gönderen kişinin çakışmayı çözmesini bekler. Bu durum, ayrı bir dünya için olsa da, 3.8.3 bölümünde uygulama ve veri tabanı arasındaki çakısma çözme dinamiğine çok benzer. Ve uygun çözüm ne kadar ilginç ki "tekrar" iyimser kitleme (optimistic locking) yönünde olmuştur, yâni burada da çözüm çakışmayı sonradan çözmek yönündedir. İyi fikirlerin sürekli ortaya çıkma gibi bir adetleri vardır.

İlk Kod Deposunu Hazırlamak

CVS kurmak için A.10 bölümüne başvurunuz. KKİ sistemi için önce bir depo yaratmak gerekir. Bu depo, bütün kaynak kodun saklandığı yer olur. Projenin ilk kodunu sisteme eklemek için, CVS'in olduğu makinaya bağlanıp (yâni `CVSROOT` değişkenini o makinadaki depoya set edip), `cvs import` komutunu kullanmalıyız. Bu komut, sadece projenin başında gereklidir. Teknik lider projenin dizin yapısını, Ant script'lerini, mimari için gereken Java dosyalarını, ve jar kütüphanelerini kendi yerel makinasında hazırladıktan sonra bu yapının tamamını CVS'e koymak isteyecektir. O anda CVS bomboştur, ve ilk giren kod bu kodlar olacaktır.

Bu durum tipik bir `cvs import` gerektiren bir durumdur. Teknik lider, eğer projeyi `/usr/local/proje1/` altında kurdu ise, önce o dizine gider, ve şu komutu işletir.

```
cvs import -m "Proje Ekleniyor" havuz_ismi vendortag releasetag
```

Bu komut, `havuz_ismi` için ne kullanıldıysa, o isimde bir kod havuzu CVS'te yaratacaktır, ve `/usr/local/proje1` altındaki bütün kodları oraya koyacaktır.

Bu komutu kullanırken, CVSROOT değişkeninin doğru ayarlanmış olduğunu farzediyoruz. CVS kurmak ile ilgili A.10 bölümünde, havuzun fiziksel adresini taşımak için `-d` yaklaşımı yerine `CVSROOT`'un daha iyi olacağını belirtmiştik.

Bir önemli not daha: `cvs import`, (teknik lider için) `/usr/local/proje¬ 1` dizinini geliştirme yapmaya hazırlamaz, sadece o kodu içeri koyar. Yani, o kodu içeri koyan teknik lider cvs commit, cvs update gibi komutları kullanmak istiyorsa, önce `cvs co havuz_ismi` komutunu işletmeli ve kodu aynen öteki programcıların yapacağı gibi dışarı çekmelidir. Bunu ya aynı dizinde, ya da başka bir dizinde yapabilir.

Dallanma (Branching)

ClearCase kullanılan bir projede çalışmış olanlar, branch kullanımının ne kadar fazla olduğunu gözlemlemiştir. ClearCase, branch yaratmayı ve kullanımını büyük ölçüde rahatlatır. ABD'de 'büyük beşli' diye bilinen danışman şirketlerinden (Deloitte & Touche, Ernst & Young, KPMG, PriceWaterhouseCoopers and Accenture) gelen arkadaşlarda bu derin ClearCase etkisi ile bir 'branch' kültürü hissedersiniz. Bu şirketlerdeki ClearCase kullanımının sebebi, büyük projeleri idare etmek için eskiden ClearCase'den daha profosyonel bir aracın olmamasıdır. Bu şirketler de yazılımda eski ve köklü şirketlerden sayılırlar.

ClearCase'çilerin de çok branch kullanmalarının sebebi de aslında şudur: ClearCase de çalışırken, bir dosya üzerinde çalışmak için önce onu kitlemeniz gerekir. Eğer branch'ler olmasaydı, aynı kod üzerinde çalışan programcılar *aynı anda*, *aynı dosya* üzerinde çalışamaz olurlardı. Bu da kabul edilmez bir durumdur. İşte bu sebeple de aynı dosya üzerinde çalışması muhtemel olabilecek aynı takımın programcıları, kendileri için ayrı birer branch yaratma yoluna gider

olmuşlardır. Yâni kişi başına branch yaratmak, ClearCase kodcuları için bir lüks değil, aslında bir *zorunluluktur*.

Fakat açık yazılım, kendisi ve kültürü ile bu yaklaşımı değiştirmeye başladı. Öncelikle, CVS kopuk (disconnected) çalışmaya uygun bir araç olduğu için, bu stil, dünyanın herhangi bir yerinde olabilecek açık yazılımcı için daha uygun bir model olmuştur. CVS'te bir dosya üzerinde çalışmak için onu kitlemeniz gerekmiyor. CVS, herkesin aynı dosya üzerinde değişiklik yapmasına izin verir, ve bu sebeple olabilecek, yâni aynı dosyayı birkaç kişinin değiştirmesinden ortaya çıkabilecek, çakışmaları (merge conflicts) önceden engellemek değil, *sonradan, commit anında kontrol etmeyi* seçer. Çakışmaları çözmek, biraz CVS'in yardımıyla ikinci değişikliği yapan programcıya kalır.

Örnek: A ve B tüm kodu checkout ederler. A dosya X'i değiştirir. B X'i değiştir. A commit eder. B commit eder, ve B çakışma hatası görür. Bu çakışmaları, `cvs up` komutu otomatik olarak çözebilir, ve *lokal* dosyasında, güzelce formatlanmış şekilde hangi parçanın hangi kodcudan geldiğini gösteren bir şekilde bir otomatik birleştirim (auto merge) sonucu görecektir. Ama iş bitmiş değildir, final bir versiyon için bu parçalardan birinin ya da ötekin seçilmesi, ve bu final versiyonun CVS'e commit edilmesi gerekir. Ancak bu yapılınca iş tamamlanacaktır.

CVS ClearCase'den kitleme bağlamında değişik olmasına karşın, bu, CVS'te hiç branch kullanılmayacağı anlamına gelmez. Evet, CVS projelerinde branch kullanımı daha azdır, ama *gereksiz* değildir. O zaman şu soruların cevabını verelim:

- CVS'te ne zaman branch yaratmak gerekir, ya da daha genel anlamında 'bir yazılım projesi ne zaman branch yaratmalıdır?'.
- Ne kadar sıklıkla branch yaratılmalıdır?
- Branch'lerle alâkalı hangi disiplin takip edilmelidir; Yâni branching yaplarımız, yapmalarımız nelerdir?

İlk soruya cevap verelim. Bazı şartlar vardır ki, branch kullanımı kaçınılmazdır. Meselâ, 2004-10-02 tarihinde HEAD üzerinden bir sürüm yaptınız ve yazılımın bir sonraki özelliklerini eklemeye devam ettiniz, haftalar geçti ve kodcularınız tam gaz gidiyorlar. CVS'i kullanma stiliniz şöyle: Kodun son hâlini HEAD üzerinde tutuyorsunuz, programcılar kodlarını son hâlini buraya commit ediyorlar.

Fakat, birden bir kullanıcınız (user) haftalar önceki 2004-10-02 sürümünde bir hata buldu. Fakat öyle bir şey ki, siz yeni versiyon içinde zâten 'bu hatanın olduğu modülü' tamamen değiştirmektesiniz, yâni hata veren kod ortalıkta yok! Bu hatanın tamir edilmesi ve kullanıcıya yeni bir sürüm verilmesi gerekiyor. Ne yapacaksınız? Tamir edilecek kod artık HEAD'de değil, ya da orada, ama üzerinde başka bir şey eklenmekte, yâni çalışma hâlinde.

İşte bir branch, bu klasik derde bir devadır. Prensip olarak, sürüm yapılınca (örneğimizdeki 2004-10-02 tarihi) kodun o andaki haline CVS'te bir tag (etiket) atılır. Ve bundan sonra geliştirmeye HEAD üzerinde devam edersiniz. Bu sayede hata raporu gelince, tag koyulan yere geri dönebilirsiniz. Hatayı tamir için, HEAD'i etkilemeyecek bir yan dal açmanız gerekir, yâni bir branch. Şimdi hatayı, *bu branch üzerinde* tamir edip, commit etmeniz mümkündür. Bu değişiklik HEAD tarafından görülmeyecektir. Bu durum HEAD üzerinde yeni özellikleri ekleyen arkadaşları mutlu eder, çünkü kendi hâllerinde işlerine devam etmek isterler.

İşte branch'in faydası budur. Bir branch, herhangi bir zamanda HEAD'in sanal bir kopyasıdır, ve o andan itibaren ondan bağımsızdır. Branch'ler sayesinde HEAD'i öteki branch'lerden izole edebiliyoruz. Yan detay olarak: bir branch HEAD'den olduğu gibi başka bir branch'ten de açılabilir, CVS buna izin verir, ama bu yaklaşımı tavsiye etmiyoruz. Bir branch'i sadece HEAD'den açmalıyız.

Hata tamir senaryomuza dönersek; Diyelim ki, 2004-10-02 branch'i üzerinde yapılan değişiklik projenin en son hâlinde isteyeceğimiz bir özellik. Yâni eğer yaptığınız düzeltmeleri HEAD üzerine de koymak istiyorsanız, bunu merge işlemi ile gerçekleştirebilirsiniz. Bu işlem, cvs up yaptığımızda olan merge işleminin branch-lerarası bir karşılığıdır. Aynen normal CVS kullanımında olduğu gibi eğer çakışmalar olursa, CVS sizi uyaracaktır. Bu işlem için gereken komutları daha sonra göreceğiz.

Branch'ler ile çalışmaya başladığınızda, özellikle Kaynak İdare Lideri (KİL) olan siz, birden fazla olacak branch'leri hatırlamak ve ismine bakarak ne için yaratılmış olduğunu çabukça anlabilmek için bir *isimlendirme stratejisi* geliştirmek zorundasınız.

Tavsiyemiz bu ismin, aşağıdaki her kavram için diğerlerinden _ ile ayrılan bir kelime kullanmasıdır. Bu kavramlar

- **Sabit bir önek**: İsmin branch olduğunu belirten B harfi, ya da tag için T.
- **Proje safhası**: (ALPHA, BETA, RELEASE)
- **Bir ayraç**: KKİ tarafından ya da kullanıcı tarafından kararlaştırılır. 1,2,3 ya da ÖNCE, SONRA, kullanıcı ismi, vs.
- **Tarih**: (YIL-AY-GÜN olarak, örnek: 2004-10-20)

Örnek olarak, BETA safhasında yarattığımız bir branch ismine bakalım: `B_BETA_1_2004_10_22`.

Bir branch yaratmak için, `cvs tag` komutunu kullanmamız gerekiyor. Bu durum, `cvs tag` komutunu daha önce salt etiketlemek için kullanlar için biraz garip gelebilir. Fakat tarihi bir takım sebepler yüzünden durum böyledir.

Branch'e dönelim: Başta bahsettiğimiz senaryo bağlamında tag koyma işi (pür etiket), branch'in kendisinden muhakkak daha önce gelecektir. Her release'den sonra, muhakkak tag atmayı unutmayın! Eğer dönecek işaretiniz

yoksa, o noktadan branch açmanız imkansız hâle gelir. Projenin önemli kilometre taşlarını bir tag ile işaretlemeyi alışkanlık hâline getirin.

Örneğimize dönelim: Bu şekilde atılmış olan tag isminin `T_RELEASE_1_2¬ 004_10_22` olduğunu farzedersek, önce o tag'e dönüp, yeni bir çalışma dizinini sadece o tag için checkout etmemiz gerekecektir.

```
cd /vs/vs/proje1/

cvs co -d eski_release -r T_RELEASE_1_2004_10_22
```

Şimdi bu çalışma dizini içinden bir branch oluşturabiliriz.

```
cd /vs/vs/proje1/eski_release/ModulIsmi

cvs tag -b B_RELEASE_1_2004_10_22
```

Böylece sürümün yapıldığı anda tanımlanmış tag üzerinden artık bir branch yaratmış oluyoruz. Artık değişikliklerimizi bu branch üzerinden yapabiliriz.

Fakat dikkat! Hâlen branch üzerindeki kodları gösteren bir çalışma dizini içinde değiliz. Bu çalışma dizinini yaratmak için, cvs co komutunu kullanmamız gerekiyor.

```
cd /vs/vs/proje1/

cvs co -d Relase1Branch -r B_RELEASE_1_2004_10_22
```

Tamam. Artık `/vs/vs/proje1/Relase1Branch` altında branch kodları üzerinde istediğimiz düzeltmeyi yapabiliriz. Geliştirme sırasında artık tamamen ayrı bir branch'te olduğunuza göre, istediğiniz kadar checkin yapabilirsiniz. Bu commit'ler sadece branch'inize gidecektir, HEAD etkilenmeyecektir. Değişim bittikten sonra, önce update sonra commit yapmalısınız. Her commit'ten önce update yapmayı da alışkanlık hâline getirin. Bu sayede aynı branch üzerinde olabilecek çakışmaları görebilirsiniz.

```
cd /vs/vs/proje1/Relase1Branch/ModulIsmi

cvs -q update
```

Merge çakışması gelmedi ise, commit edebiliriz.

```
cvs ci -m "Hata no 3342 tamir edildi"
...
...
(commit çıktısı)
...
...
```

Örneğimizin başında, tamir edilmesi gereken kodun artık ortada bile olmayabileceğini söylemiştik. Bu durumda HEAD branch herhalde yapılan tamire ihtiyaç duymaz. Siz de tamiri branch içinde bırakır ve bir daha onu kullanmazsınız (branch'i silmeyin). Fakat durum öyle olabilir ki (ve çoğunlukla böyle

olacaktır) yapılan tamir HEAD üzerinde de yapılmalıdır. Hata raporu release tarihine oldukça yakın olabilir ve gelen rapor HEAD üzerindeki yeni özellikleri bile etkiliyordur. Bu durum, branch'ten HEAD'e merge yapmamızı gerektiren bir durumdur.

Bu noktada, 'eğer aynı değişikliği HEAD üzerine nasıl olsa geçireceksek, niye bu iş için bir branch açtım. Release tag'deki hâlimize dönüp, orada tamir yapabilirdim' diye düşünebilirsiniz. Bu düşünce şu sebeple geçersizdir.

1. Release tag üzerinde kod değiştirirseniz, tag, hareketli bir tag olacaktır. Halbuki release tag'i, kodun belli bir anda taşa kazınmış hâli olmalıdır. Siz değişiklikleri release tag üstünde commit ederseniz, kodun ilk release'deki hâline bir daha asla dönemezsiniz. 'Zaten ihtiyacım olmaz' demeyin. Gerekebilir.

2. Branch üzerinde tamir yapmakla, istediğiniz kadar commit etme, ve günlerce bu branch'te kalma lüksüne kavuşmuş oluyorsunuz. Ne siz HEAD'i, ne HEAD sizi etkiliyor.

Tamam. Şimdi HEAD'e merge tekniğini görelim. Biz de bir geliştirici olduğumuza göre, HEAD'e işaret eden yâni en son kodları içeren bir çalışma dizinimiz olacaktır. Yoksa yaratalım. Merge'e başlamadan önce de `cvs up -d` ile güncelleyelim. Bu dizin içine gidelim, ve önce "update" ile çakışmaları görelim. Çakışmaları çözdükten sonra da commit (`ci`) ile işi bitirelim.

```
cd /vs/vs/proje1/HEAD/ModulIsmi

cvs -q update -j B_RELEASE_1_2004_10_22

cvs -q ci -m "B_RELEASE_1_2004_10_22 Branchinden Merge Edildi"
```

Bravo. İlk merge'ü gerçekleştirdiniz.

Bu merge bittikten sonra, eğer aynı branch'i ileride tekrar kullanmak istiyorsanız, önemli bir uyarıda bulunmak isterim. Aynı branch'ten HEAD'e *birden fazla* merge yaparsanız (tabii ki yeni değişiklikler ve yeni bir commit yaptıktan sonra), bir sürprizle karşılaşacaksınız. CVS, daha önce yaptığınız değişiklikleri tekrar merge etmeye çalısacak! Ve bu sebeple "warning: conflicts during merge" mesajını görürsünüz. Bunun sebebi, CVS'in "son yaptığınız commit'lerin kümesi" diye bir anlayışının olmamasıdır. CVS'e göre, "beraber commit olmuş" dosya A,B,C arasında hiçbir bağlantı yoktur. Merge işleminin tarifi de branch'in üst noktası ve HEAD arasındaki farkın uygulanması olduğuna göre, üst nokta, aynı değişiklikleri içerecektir, ve çakışma ortaya çıkacaktır.

Bu problemin üstesinden gelmek için, CVS'e biraz yardım etmemiz lâzım. Branch üzerinde ilk tamiri gerçekleştirdikten sonra branch üzerine bir tag koyun. İsmi, meselâ, `B_RELEASE_1_2004_10_22_tamir_1` olsun.

Böylece İkinci tamiri yaptıktan (ve commit ettikten sonra), CVS'e 'bu tag'ten önce yaptıklarımı istemiyorum' diyebilirsiniz. Bunun için (HEAD çalışma dizini içinde) şunları kullanmanız lâzım.

```
cd /vs/vs/proje1/HEAD/ModulIsmi

cvs -q update -j B_RELEASE_1_2004_10_22_tamir_1 B_RELEASE_1_2004_10_22

cvs -q ci -m "B_RELEASE_1_2004_10_22 tamir 2 Branchinden Merge Edildi"
```

İlk -j başlangıç tamir 1'den sonra konulan tag'dir, ikinci -j normal branch ismidir.

Herhalde burada alınacak ders şu olmalıdır: CVS'te sık sık tag atın! Bir release olayı, bir branch'ten HEAD'e merge olayı, ya da bir hata tamiri CVS dünyasında *önemli* olaylardandır, ve günün anlam ve önemini belirten bir tag ile kutlanmaları (!) gerekir.

Artık seçtiğimiz yolu özetlemeye hazırız.

- CVS'te en rahat geliştirme yolu, programcıların en sonu kodu HEAD üzerinde geliştirdikleri yoldur. Zâten progamcıların çalışma dizinleri bir nevi branch gibi de görülebilir. Ama bu yöntemi verimli çalışır hâlde tutmak için, şu kuralları sıkı sıkı takip edin:
  - Programcılarınıza sık sık commit etmesini söyleyin (her gün).
  - Programcılarınıza derlenmemiş ve testleri geçmeyen kodu commit etmemesini tembih edin.
  - Herkesin CVS'ten en son kodu sık sık almalarını söyleyin.
- Branch'leri özel durumlar (hata tamiri) için kullanın.
- Önemli olaylarda tag atın (hem HEAD üzerinde, hem de branch'ler üzerinde). Meselâ bir release, ya da hata tamiri muhakkak tag gerektiren durumlar olmalıdır.
- İşinizin bittiği branch'i kendi hâlinde bırakın. Çalışma dizinini silebilirsiniz (tabii içindeki herşeyin commit edildiğine emin olduktan sonra). Bu branch'i CVS'ten silmeye uğraşmayın. Tekrar işinize yarayabilir.

Kaynak Kodu Etiketlemek

Teknik lider, ya da onun eğittiği ve görevlendirdiği projede idari işlere bakan arkadaş, ne zaman test için önemli bir sürüm yapılmışsa o sürümün referans aldığı o anki kod durumunu "dondurmak" için, CVS'te bir etiket atmalıdır. Etiket atmak, bir nevi koda işaret bırakmaktır. Bu işarete sonradan dönülebilir, hattâ sadece işarete yönelik `cvs commit` işlemleri bile yapabilirsiniz. Fakat geri dönmek, veya hatırlamak için bu işareti bırakmak daha yaygın bir yaklaşımdır. Etiketlemek için,

```
cvs tag <<etiket_ismi>>
```

komutunu kullanabilirsiniz.

Teknik lider, ne zaman müşteriye ve ya büyük bir test yönelik bir sürüm yaparsa, bu etiketleme işlemini gerçekleştirmelidir. Etiket isimleri, `release_1` (sürüm 1), `release_2`, gibi isimler, ya da sadece bir numara olabilir. Tarih içeren sürüm isimleri de görmüştük.

Daha büyük projelerde, etiketleme süreci ile *hata takip programı* arasında ilişkinin kurulması gerekecektir, çünkü sonuçta sürüm yapılmıştır, test makinasına konulmuştur, ve bir süre sonra testçi, hata raporları göndermeye başlayacaktır. Programcı bu hataları, hangi sürümde tamir etmeli, ya da tekrar ortaya çıkarmalıdır? Bu gibi durumlarda, ITracker programının da desteklendiği gibi, hataların içinde "hangi sürümde test edileceği" bilgisi kaydedilmelidir. Bu sürüm no'su, teknik liderin CVS'te attığı etiket numaralarını baz alarak girilen bir numara olacaktır.

Ayrıca diğer yönde, testçilerin o anda "hangi sürümde olan bir programa" baktıklarını anlayabilmeleri için (önyüze bakarak sürümü anlayabilmeleri her zaman mümkün değildir), teknik liderin etiket değerini, uygulamanın parçası olan bir statik HTML sayfasına koyması uygun olabilir (meselâ `src/pages/version.html` gibi. Bu değer gönderimi, etiketleme script'inin yapacağı yan bir işlem olabilir (en uygun yer aslında orasıdır). Sürüm HTML sayfası, hep aynı yerde, hep aynı isimde olacak bir sayfa olmalıdır, ve sayfanın tek içeriği, test edilen o anki kodların sürüm numarasından ibarettir. Böylece gerektiğinde "ismi belli" bu HTML sayfasını ziyaret eden testçi, test ettiği uygulamanın hangi sürümde olduğunu rahatça anlayabilmiş olur.

11.3.3 Kod Gözden Geçirme Toplantıları

Proje takımında her programcının belli bazı kodlarını gözden geçirmek, ve daha genel amaçlı olarak takımın kullandığı teknolojiler hakkında bilgi istişaresinde bulunmak için, arada sırada resmi bir kod gözden geçirme (code review) toplantısının yapılması gereklidir. Bu toplantıdan önce teknik lider hangi programcının kodunun gözden geçirileceğine karar verir, ve bu toplantıdan önce programı kodunu yazıcıdan basarak herkese dağıtır. Kodun her yazıcı sayfasına iki kod sayfası düşecek şekilde ve yatay olarak basılması iyi olur. Bu şekilde kod basmanın tekniklerini A.11 bölümünde görebilirsiniz.

Bu gözden geçirme toplantılarına herkes baktığı kodun üzerine notlar almış olarak gelir. Toplantı sırasında herkes sırayla yazdığı yorumları kodun sahibi ile paylaşır. Programcı konu hakkında soru sorar, eğer katılmıyorsa, cevap verir. Teknik lider de konu hakkında yorumlarını ekleyecektir, böylece tüm takım birbirinden yeni bilgiler edinmiş olur.

Ruh hâli olarak gözden geçirme toplantıları programcılar üzerinde stres yaratabilir, çünkü herkesin önünde kodları hakkında yorum yapılacaktır, ve görülen eksiklikler söylenecektir. Tavsiyemiz, programcıların bu toplantıyı "idam mangası önüne çıkıyor" gibi görmemesidir. Ayrıca kod hakkında yorum verenler de yapıcı eleştiriler ile toplantıya gelmelidirler. İnsanlar mekanik hatalar

yapabilirler (bkz. Kural #4), bunlar zâten çok önemli değildir; Ama prensip hataları, takip edilmesi gereken mimari kuralların kontrolü kod gözden geçirme toplantısında yapılmalıdır. Kodlama standartlarının takibi de, aynı şekilde, bu toplantıda kontrol edilmelidir.

11.3.4 Kodlama Standartları

Kodlama standartları, pür dil seviyesinde konan ve stil ile alâkalı kurallardır. Meselâ, `if` komutlarında "`{`" işaretinin `if` kelimesi ile aynı satırda mı, bir sonraki satırda mı olması gerektiği, bir kodlama standartı kararıdır. Kozmetik bir karar gibi gözükse de, kodlama standartları kod bakımı için önemlidir, çünkü kodun birçok değişik yerine bakarken insanın aradığını bulma kabiliyetini arttırır. Proje sonunda tüm kod sanki tek kişi tarafından yazılmış gibi gözükmelidir.

Kodlama standartlarını otomatik olarak kontrol etmek istiyorsanız, `http¬://checkstyle.sourceforge.net/` adresinden Checkstyle adlı açık yazılım projesini alıp kullanabilirsiniz. Checkstyle kullanmak için kuralların tanımlandığı bir ayar dosyası hazırlamak gerekir, ve daha sonra (meselâ bir Ant target'inden çağırarak) işlettiğiniz Checkstyle, tüm kodu stil kontrolünden geçirecektir. Bulunan stil hataları, aynen derleme hataları gibi bir hata log'undan ekrana basılacaktır. Teknik lider kodlama standartları gerçekten takip edilmesini istiyorsa, projenin Ant script'i `build.xml` içinde derleme target'ini Checkstyle target'i ile bir `depends` ilişkisine sokabilir. Böylece derleme yapmak için stil kontrolü kendiliğinden olmuş olacaktır.

Teknik liderler kodlama standartlarını oluştururken, programcıları çok sıkacak kurallar getirmemelidirler; Genellikle çoğunluğun takip ettigi yöntemi kural hâline getirmek en iyisidir. Amacımız kodun aynı gözükmesini sağlamaktır, kodculara hiç normâl gelmeyen katı kurallar koymak değil.

11.3.5 Projelerde Hata Takip Düzeni

Bilgi işlem projelerinde hata takip için bir sistem kurmamız gereken zaman, stabil bir kod bazının ortaya çıkmaya başladığı ve sistemi müşterilere göstermeye başladığımız zamana yakın bir yerde olmalıdır. Hatalar, herhangi bir programcının herhangi bir kod parçasında (JSP sayfası, Java kodları, vs.) ortaya çıkmış olabilir, ve bu hatalar bulunduğu zaman bir yerlere kayıt edilip oradan takip edilmelidir. Hatalar (ve bilahere onların tamir edilme isteği) zamanla biriktikçe, hangisinin tamir edilmiş olduğu, test etmeye hazır olduğu gibi konular idare açısından saç yolduracak seviyeye gelebilirler. 20-30 tane hata bile, bir hata takip sistemi kurmamızı gerektirecektir.

Bu hata takip sisteminin ana özellikleri şunlar olmalıdır. Sistem,

- Programcılara atanan/ait hataları sadece onlara gösterebilmeli
- Hataların tamir durumunu muhafaza edebilmeli

- Tamirden sonra tekrar test edilecek hataların hangileri olduğunu testçilere gösterebilmeli
- Hataların testçiden programcıya, ve tamir edildikten sonra programcıdan testçiye geri olan **iş akışını** destekleyebilmeli.
- Hataların değişim tarihçesini, herkesin düştüğü notları kayıtlı tutabilmeli ve gösterebilmelidir

Bu ana hatları destekleyecek en iyi uygulama türü, portal şeklinde olan bir hata takip sistemidir. Buna karşılık, Excel tablolarında hata takip yapmak çok külfetli olacaktır, çünkü bu şekilde tablolarda iş akışı, tarihçe tutmak imkansızdır. Tek bir Excel tablosunda birkaç programcının hatalarının en son durumunu kaydedilip birleştirilmesi çok zordur. Portal bazlı hata takip programı, her programcıya ve testçiye bir giriş ismi verecek, testçilerin yarattığı hataları programcılara göndermesini rahatlatacaktır.

İş Akışı

İş akışının genel hatları şöyle olur.

1. Testçi, hata portal'ına kendi kullanıcı ismini kullanarak girer ve projenin uygulamasında gördüğü bir hatayı sisteme ekler. Bu hatayı, bir programcıya atar.
2. Hatayı alan programcı, o hatanın çıktığı kodu yazan, ya da o kodu iyi bilen bir başka programcı olabilir. Bu programcı, günün herhangi bir saatinde portal'e girip hataları listesini kontrol edecektir (ya da hata portal'inden otomatik e-mail almıştır, portal hata atanır atanmaz programcıya e-mail atmak için ayarlanmış olabilir), ve listesindeki kendine atanmış hatayı görür. Herkesin `benimPortal` listesi, kendine atanan hataları göstermesi için ayarlanmıştır (buna testçi de dahil, çünkü ona da hatalar atanabilir)
3. Programcı hatanın detayına tıklar, ve tanımı okur. Kendi geliştirme ortamında hatayı tekrar ettirerek (duplicate) kendi de görür, ve tamir etmek için kolları sıvar.
4. Programcı hatayı tamir edince, yeni kodu kaynak kontrol sistemine ekler. Projenin teknik lideri her gün başında test makinasına (bkz. 11.3 bölümü) zaten sürüm yapıyordur, ya da acil bir sürüm yapılarak (o hata çok önemli ise) test makinasına en son kodlar atılır.
5. Programcı, portal'den tamir edilmiş hatanın detayına tekrar inerek, bu sefer bir değişiklik yapar. Hatanın konumunu "çözüldü" olarak değiştirir, ve hatanın yeni sahibi olarak "testçi" arkadaşını seçer. Hatayı kaydeder. Şimdi programcı kendi "benimHataPortal" ekranına döndüğünde, tamir

etmiş olduğu hatanın kaybolduğunu görecektir (çok güzel, yapacak iş azaldı).

6. Bu hata tabii kaybolmadı. Testçi kimse, "benimHataPortal"'ına girdiğinde (ya da aynı şekilde e-mail aldığında) portal'a girecek, ve listesindeki hatayı görecektir. Testçilerin hata alması garip olmamalı, testçinin görevi sadece test etmek olduğu için bu hatanın onun listesinde olmasının tek bir anlamı vardır: Hata tamir edilmiştir, ve tekrar test edilmeye hazırdır.

7. Test makinasında da en son kodlar olduğuna göre, testçi arkadaş bu makinaya bağlanarak hatanın olduğu kısma/bölüme giderek kontrolünü yapar.

8. Eğer hata gerçekten tamir olduysa, testçi portal'a dönerek o hatanın detayına gelecek, ve "kapandı" statüsüne getirecektir. Başka yapması gereken bir şey yoktur.

9. Fakat hata hâlâ ortaya çıkıyor ise, o zaman hatanın statüsünü "açık" olarak değiştirmeli, ve sahibi olarak (tamir ettiğini zanneden) programcıyı atamalı, yani, hatayı programcıya geri göndermelidir! Ve bu şekilde iş akışı tekrar başlamış olur.

10. Eğer hata tamir olmuş ise, kapandı statüsünü alan hata herkezin listesinden kaybolacak, yani tamir edilen hataların arasına karışmış olacaktır.

    Not: Konu hakkında eklememiz gereken iki nokta şöyledir:

    - Dikkat edilmesi gereken bir nokta, bir hatanın tamir olmuş ama, o hatadan bağımsız başka bir hata aynı ekranda/bölümde bulunmasında ortaya çıkar. **Bu yeni hatanın eski hata üzerine not düşülmemesi gerekir**. Çünkü yeni hata değişik bir hatadır. Ayrı şekilde takip edilmesi gerekecektir.
    - Üstte tarif edilen düzende şöyle bir nüans farki uygulanabilir: Testçi hatayı atar (programcıyı seçerek). Ama programcı, tamirden sonra tesçiye geri atama yapmaz. Sadece hata konumunu "çözüldü" ye getirir. Bu durumda testçinin görevi, periyodik bir şekilde portal üzerinde statüsü "çözüldü" durumunda olan hataları kontrol etmektir çünkü hatayı geri kendi `benimPortal` listesinde görmeyecektir.

ITracker

Evet, genel hatları ile tarif ettiğimiz bu iş akışını ITracker programı ile gerçekleştirmeye çalışacağız. ITracker'in nasıl kurulacağını A.9 bölümünda bulabilirsiniz. Programa admin olarak ilk girdiğinizde bu ekranı göreceksiniz. (Eğer bu ekran

İngilizce geldiyse, My Preferences altından Turkish seçerek Türkçe ekrana gelebiliriz) [2]

Yukarıdaki ekranda, bekleneceği gibi, hiçbir proje yok. İlk yapmamız gereken, bir proje yaratmak. İlk önce Sistem Bakımı seçeneğine tıklayın.

"Proje Bakımı | Bakım Yap" seçeneğinden "Proje Listesi" ekranına gelin.

Şimdi üst sağda bulunan sayfa ikonuna basarak yeni proje yaratma sayfasına gelebilirsiniz.

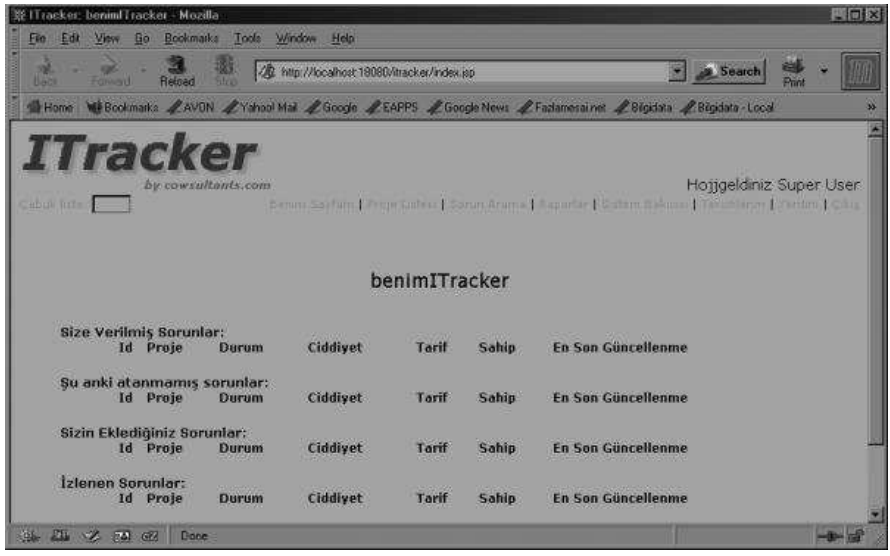
Üstteki örnek bilgileri girerek hemen bir proje yaratmamız mümkün. Proje ismi "deneme" olarak seçildi.

Kullanıcılar

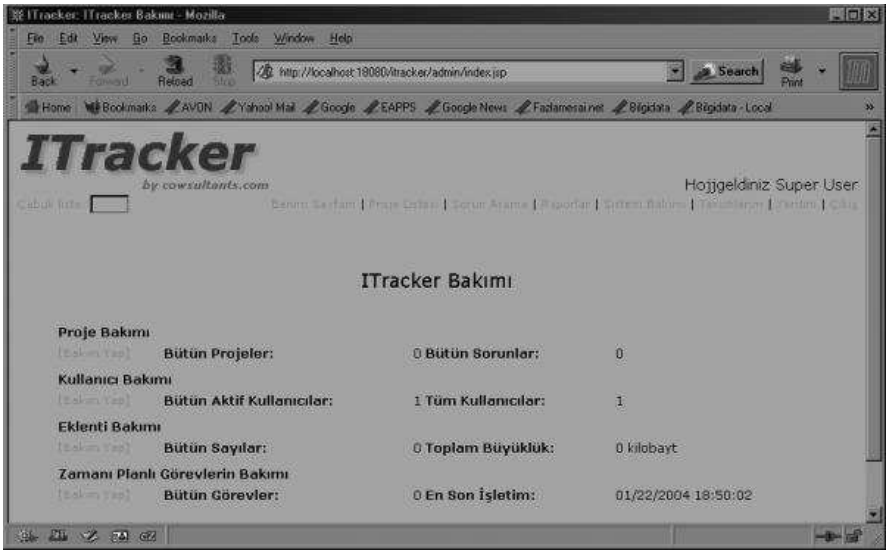
Şimdi de kullanıcılar yaratmamız gerekiyor. Bu kullanıcılar, projede görev yapan programcılar, proje müdürü gibi kimseler olacaklar. ITracker'ı kullanan herkese ayrı bir kullanıcı ismi gerekecek, çünkü herkes kendine ait olan sorunlari (hataları) kendi kullanıcısı üzerinden görecek.

Şekil 11.8 üzerinde görüldüğü gibi ilk kullanıcıyı yaratıyoruz. Kullanıcının ismi Ali, ve testçilik görevini yürütecek. Ali, uygulamayı test edip çıkan hataları programcılara atamak ile görevli.

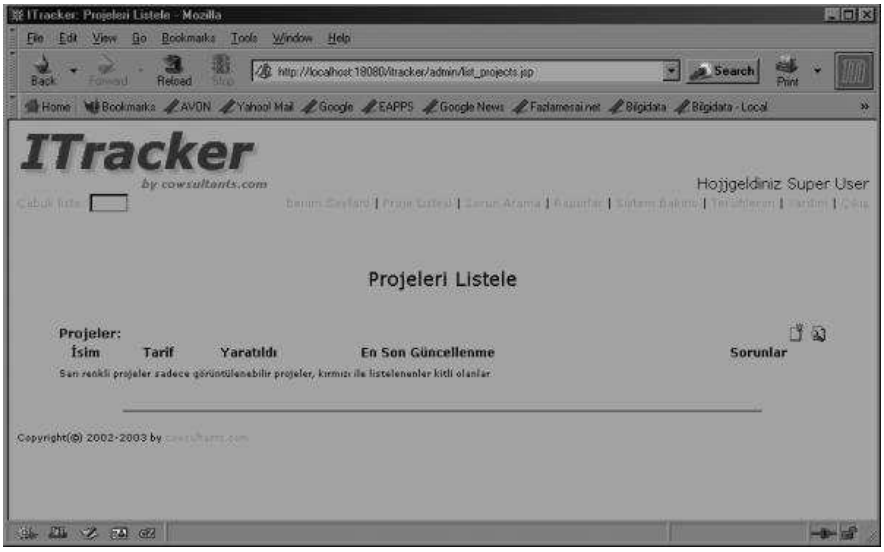
[2]ITracker programı bu satırların yazarı tarafından Türkçeleştirilmiştir



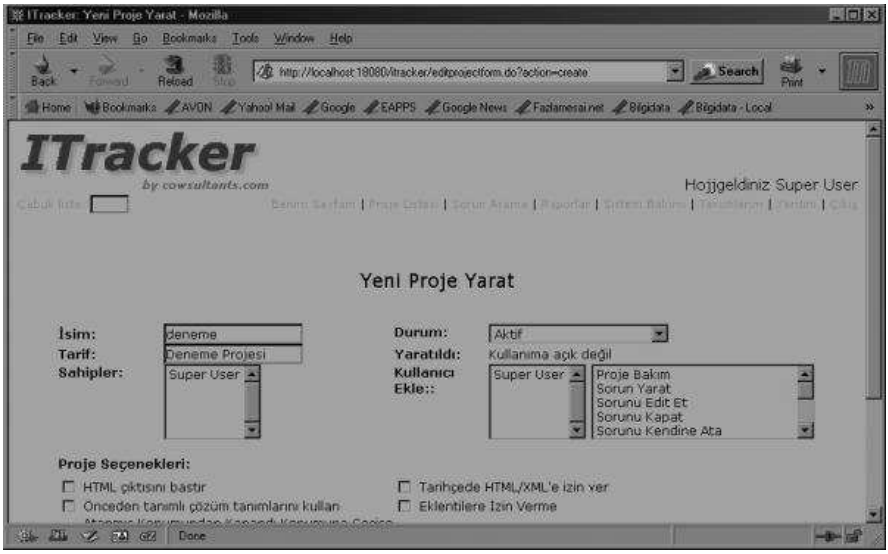
Şekil 11.3:



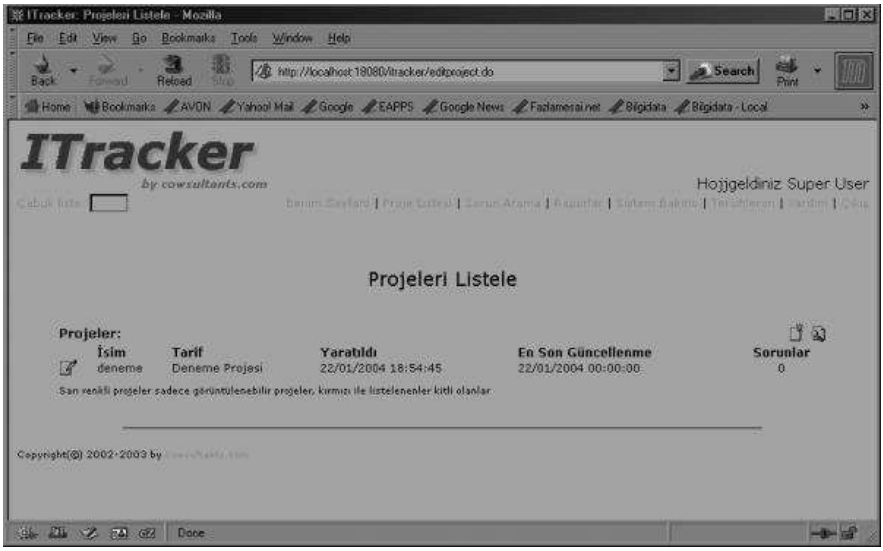
Şekil 11.4:



Şekil 11.5:



Şekil 11.6:



Şekil 11.7:

Ali'yi eklediğimiz ekrana dikkat ederseniz, ona birçok hak verdiğimizi görüyoruz. Bunlardan en önemlisi "Proje Bakım" ve "Sorunu Kapat" haklarıdır. Proje müdürü (ya da testçisi) olarak Ali, sorunları tamir edildikten sonra tekrar test edip, sorunun çözülüp çözülmeyeceğine karar verebilecek tek kişi olmalıdır. Programcılar sorunları kapatmamalı, Ali'ye test edilmesi için **yollamalıdır**.

Şekil 11.8:

Aynen Ali'yi yarattığımız gibi, Veli'yi de benzer şekilde ekleyebiliriz. Tabii Veli'de testçilere özel haklar olmayacak.

Hata Eklemek

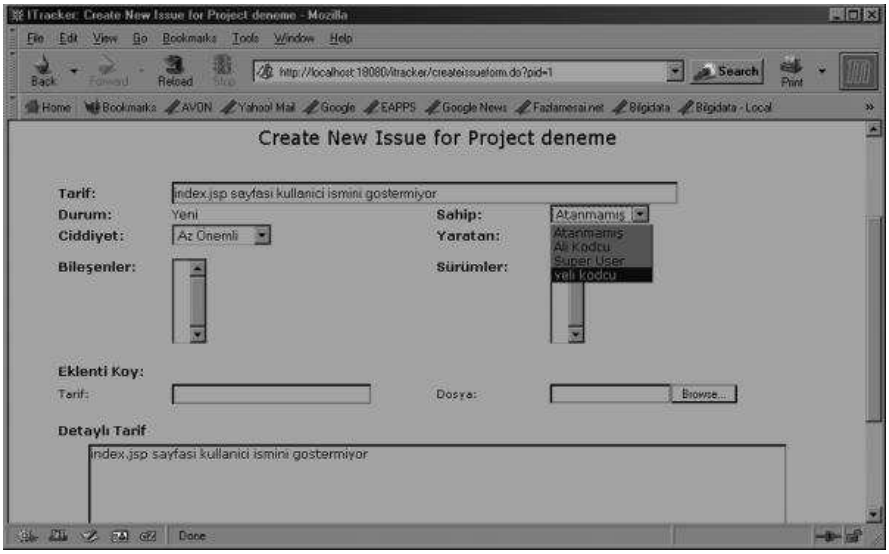
Bu noktada, Ali ve Veli sistemi kullanmaya hazırlar. Ali testçi, Veli programcı. Şimdi, Ali'nin sisteme nasıl sorun eklediğini görelim.

Ali, ilk giriş yaptığında benimITracker ekranını görecek. Buradan, üst kısımdaki Proje Listesi seçeneğinden "deneme" projesini görebilir. Ali, en solda bulunan üçlü ikon gurubundan ortadakini seçerek, yeni bir sorun ekleyecek ve Veli'ye atayacak. Görelim.

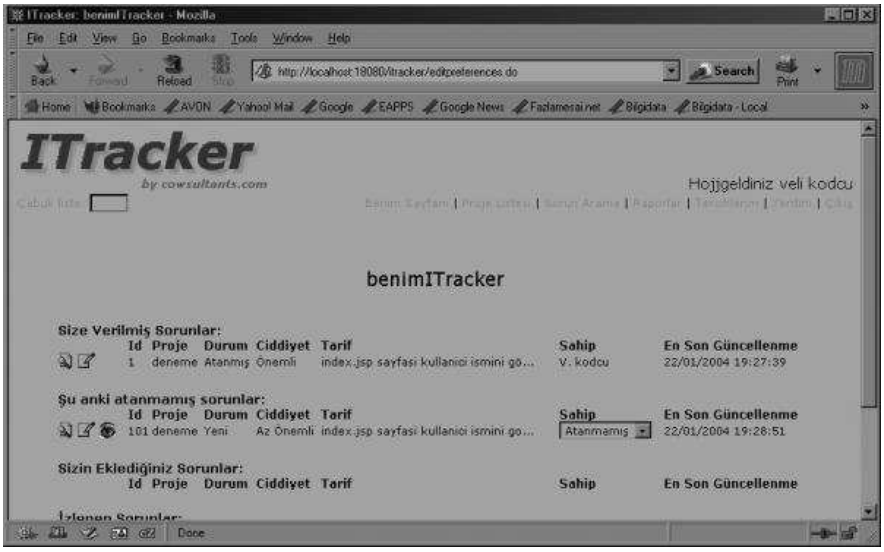
Atanma yapıldıktan sonra, görmek için Veli olarak sisteme girebilirsiniz.

Şekil 11.11 üzerinde görüldüğü gibi bir sorun Veli'ye atanmış.

Not: Eğer Veli, "benimITracker"'ında çıkan listesini, sadece ona atanmış sorunlara indirmek istiyorsa, bunun ayarını rahatlıkla yapabilir. En üstteki "Tercihlerim" seçeneğinden,



Şekil 11.9:



Şekil 11.10:

"benimITracker Bölümlerimi Sakla" kısmına gidip, orada "Atanmış Sorunlar" hariç bütün diğer listeleri kapatabilir.

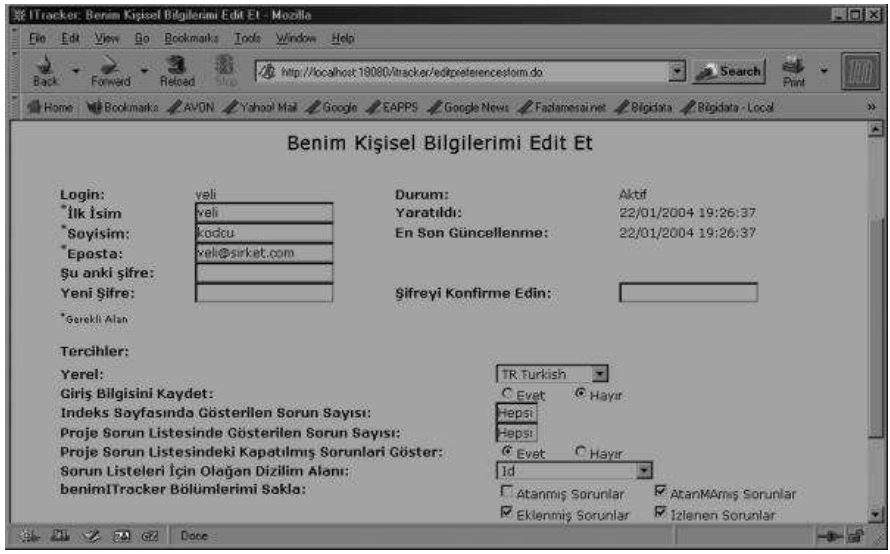
Şimdi gelelim iş akışımızda önemli bir bölüme: Veli, uygulama kodlarını hatasını tamir etmek üzere değiştirdi, kendi geliştirme makinasında test etti ve tamir edildiğine karar kıldı. Şimdi, bu hatanın "tekrar test" için Ali'ye geri gönderilmesi gerekiyor.

Hata listesinden bu hatayı değiştirmeyi seçip, aşağıdaki gibi bir giriş yaparsa, hatayı Ali'ye geri göndermiş olacaktır (kırmızı ile işaretli alanlardan)

Bu sayede Ali, kendi "benimITracker" sayfasından, tamirini istediği hatanın geri gelmiş olduğunu görecek, bu hatayı test ederek eğer tamir olmuşsa Kapat seçeneği ile hatayı kapatacaktır. Kapanmış statüsündeki hatalar kimsenin "benimITracker" listesinde gözükmezler. Onlar tamir edilmiştir!

11.4 Kullanım Kılavuzu

Kullanım kılavuzu (turnover document) belgesi, bir projeyi geliştirmiş programcılar projeden çıktıktan sonra, onlardan sonra kod üzerinde bakım yapması gerekecek programcılara bırakılmış bir mesaj niteliğini taşır. Bu dokümanda uygulamaya yeni bir kod parçası eklemek için yapılması gerekenler, geliştirme ortamının nasıl kullanılacağı, en son kodu sonuç ortamına göndermek için yapılması gerekenler gibi bakım/idare detayları anlatılır.

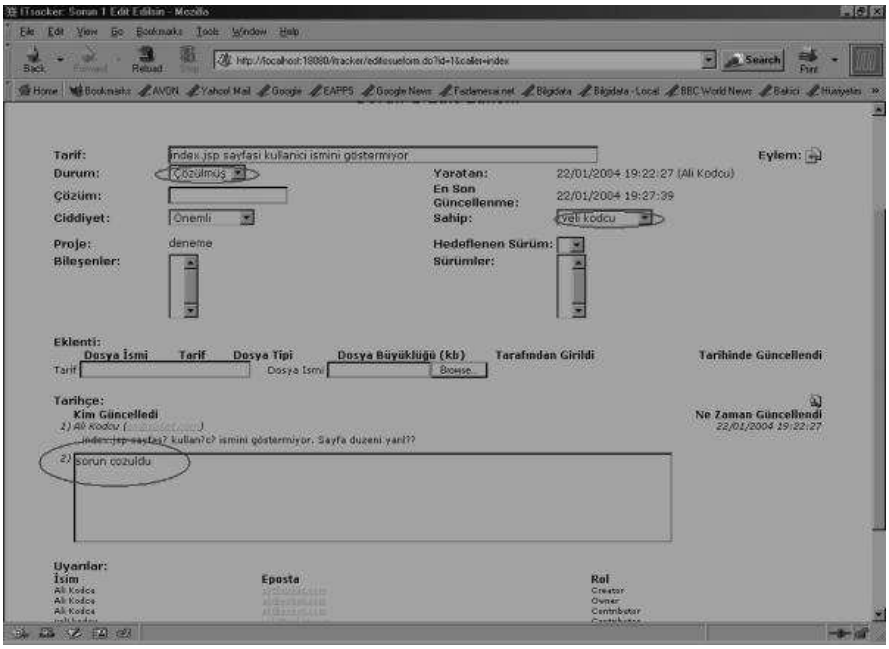


Şekil 11.11:

Kullanım kılavuzu, danışman şirketlerinin her proje sonunda ürettiği standart dökümanlardan biridir, çünkü proje bittikten sonra takım elemanları genelde başka projelerde ve başka müşteriler için görevlendirilirler, ve bu orijinal takım projeden ayrılacağı için onların tüm bilgilerinin bir yerde belgelenmesi şarttır. Fakat kullanım kılavuzu, şirket içi (in-house) yazılım takımları, ya da ürün şirketleri için bile faydalıdır. Takıma yeni katılan programcılar bu belgeleri kullanarak geliştirme sürecine daha rahat bir şekilde dahil olabilirler.

Web uygulamaları dünyasında, eğer uygulamayı işletmekle ve ayakta tutmakla sorumlu bir ASP (Application Service Provider - Uygulama Servis Sağlayıcısı) şirketi var ise, kullanım kılavuzunun sonuç ortamı ile alâkalı bölüm en çok onlara hitaben yazılacaktır. Sonuçta uygulamayı sürekli işletmekle görevli olanlar onlardır, ve sonuç ortamı bölümü "servis nasıl başlatılır", "veri tabanına bağlanmak için gerekenler", "sonuç ortamına kod gönderme" gibi detayları içerdiği için ASP'ler için okunması gereken bölümler hâline gelirler.

Teknik lider kullanım kılavuzu belgesinin en önemli yazarıdır, çünkü sistemsel detaylara hâkim olan kişi o olacaktır. Fakat bazen bu görev, kariyeri teknik liderlik yönünde ilerleyen senior bir programcıya da verilebilir.



Şekil 11.12:

Ekler

Ek A

Araçlar

Bu kitapta anlatılan kavramları evde uygulayabilmeniz için, bazı programların kurulması gerekiyor. Bu programlar sırasıyla `javac` derleyicisi, `java` yorumlayıcısı, Ant derleme sistemi, JBoss Uygulama Servisidir. Geliştirme aracı olarak Emacs editörü ve IDE ortamı anlatılacaktır, fakat herkes kendi ortamını kullanabilir, örnekler hiçbir IDE'ye bağlı değildir. Bu programların hepsinin nasıl kurulacağını ve kullanılacağını bu bölümde anlatacağız.

Kitapta referans edilen tüm örnek kodlar ve araçlar

`http://sourceforge.net/project/showfiles.php?group_id=135492`

ya da

`http://www.mycompany.com/kurumsaljava`

adresinden bulunabilir.

A.1 Örnek Kodlar

Projede referans edilen örnek kodları kurmak ve işletmek üstteki bağlantıdan `kitap-code-xx.xx.zip` dosyasını indirin ve açın. Dosyanın açılmış hâli, aşağıdaki gibi olacaktır.

```
+- DistObjs
| +- EJB
| | +- CarsEJB
| | +- CounterStateful
| +- JMS
| | +- CarsJms
| | +- Filtered
```

```
| | +- SimpleListenerServer
| | +- SimpleMdbServer
| +- RMI
| | +- CarsRMI
+- Hibernate
| +- HibernateComposite
| +- HibernateManyToMany
| +- HibernateQueries
| +- HibernateRelFetchSelect
| +- HibernateSimpleRelations
| +- SimpleHibernate
+- ProfileSample
+- Struts
| +- StrutsHibAdv
| +- StrutsHibLogin
| +- StrutsHibPerformance
| +- StrutsHibSimple
| +- StrutsHibTags
| +- StrutsTiles
| +- StrutsUpload
+- lib
| +- commons
| | +- commons-beanutils.jar
| | +- commons-collections.jar
| | ...
| +- hibernate
| | +- c3p0-0.8.4.5.jar
| | +- classes12.zip
| | +- ehcache-1.1.jar
| | +- hibernate3.jar
|  ...
| +- web
| | +- jstl.jar
| | +- struts.jar
+- tools
+- README.english
+- README.turkce
+- build.xml
+- clean.sh
+- lgpl.txt
```

Projelerin hepsi üst dizin `lib` altındaki jar dosyalarını kullanmaktadırlar. Pür Hibernate için gereken jar'lar `lib/hibernate`, pür Web/Struts/JSTL odaklı jar'lar `lib/web` ve Apache Commons jar'larına ihtiyaç duyan projeler, `lib/¬ commons` dizinleri altından Ant derleme sistemi tarafından alınıp kullanılır.

HEr projenin derlenmesi o projenin altındaki `build.xml` aracılığı ile yapılmaktadır. Hibernate üzerinden veri tabanı etkileşimi gerektiren her proje `./resources/¬`

`hibernate.cfg.xml` dosyasında tanımlı veri tabanına gitmek üzere ayarlanmıştır. Bu veri tabanı üzerinde gereken şemayı her proje içinde mevcut (ve ötekilerden değişik olabilecek) `./src/sql/tables_mysql.sql` dosyası içinde bulabilirsiniz.

A.1.1 Hibernate

Pür Hibernate ile alâkalı örnek kodları `Hibernate` üst dizini altındaki projelerde bulacaksınız. Bu kodlar yanlızca komut satırından işleyebilecek şekilde hazırlanmıştır; Hiçbir görsel birim mevcut değildir. Her Hibernate projesi `ant` komutu ile derlenebilir ve JUnit birim testleri `ant test` ile işletilebilir. Birim testleri, `hibernate.cfg.xml` üzerinde tanımlı olan tabana gitmek üzere ayarlanmıştır.

A.1.2 Web

`Struts` üst dizini altındaki her proje bir Web uygulamasıdır. Çoğunluğu JSP/JSTL/Struts/Hibernate teknolojisini kullanır. Web projelerini JBoss üzerinde deploy etmek için `build.properties` içindeki JBoss dizinini tanımlamamız gerekiyor. Örnek bir `build.properties` aşağıda gözükmektedir.

Liste A.1: build.properties

```
project.title=Jakarta Struts Blank
project.distname=kitapdemo
project.version=1.1
doc.src=./WEB-INF/src/java
jboss.home=c:/devprogs/jboss-4.0.1
distpath.project=\${jboss.home}/server/default/deploy
```

- Bu tanımlardan, kendi özel JBoss'umuz için değişmesi gereken `jboss.home` değişkenidir. Bu değişkenin programcının kendi makinasında kurulmuş olan JBoss'un yerini göstermesi gerekir.
- `distpath.project`, SAR paketinin hangi dizine gideceğini gösterir.
- `project.distname`, JBoss altında gidece SAR paketinin ismini oluşturacak baz kelimedir. Tüm örneklerde bu `kitapdemo` kelimesidir, yâni Web projeleri için SAR dosyasının ismi `kitapdemo.sar` olacaktır.

Web projemizi derlemek için geliştirme dizininde

```
\$ ant
```

komutunu kullanırız. Birim testleri işletmek için ise

```
\$ ant test
```

yeterlidir.

A.1.3 Dağıtık Nesneler

JMS, EJB, RMI teknolojilerini kullanan dağıtık nesne mimari örnekleri, `DistObjs` dizini altında bulunabilir. Her ana teknolojinin örnekleri `DistObjs` altında alt dizinler olarak bulunacaktır.

Her dağıtık teknoloji örnek dizini içinde, dağıtık teknolojiyi kullanan bir Web projesi de mevcut olacaktır. Bu projenin yapabildikleri (functionality) `StrutsHibAdv` projesi ile aynıdır. Sadece, tüm Hibernate erişim kodları *ikinci bir servis katmanı* üzerinde, ve dağıtık nesne teknolojisi üzerinden erişilerek yapılmaktadır. O zaman `DistObjs` altındaki Web kodlarını test ederken, servis tarafında bir değil, iki JBoss JVM'i başlatmamız gerekiyor.

İki JBoss demek, iki geliştirme projesi demektir. `DistObjs` altındaki Web projelerinin dizinlerine girerseniz, en üst seviyede `Server` ve `Webclient` dizinlerine ayrılmış olduğunu göreceksiniz. Bu iki proje, iki değişik JBoss `deploy` dizinine gitmesi gereken iki değişik projedir. Hangi projenin hangi JBoss'a gideceğini ayarlamak için `build.properties` dosyasında gerekli değişikliği yapabilirsiniz. Aynı makinada iki JBoss başlatmak için gerekli değişiklikleri A.4.2 bölümünde anlatılmaktadır.

Her iki projeyi bir kerede derlemek için, teknoloji dizinin (JMS, EJB, RMI) altında bir `build.xml` bulacaksınız. Bu `build.xml`'in amacı, önce `Server` dizini, sonra `Webclient` dizinine teker teker girip iki kere `ant` komutu vermekten kurtulmaktır. Böylece bir üst seviyeden tek `ant` ile işimizi halletmiş oluyoruz. Ayrıca, `Server` projesindeki POJO tanımlarının `WebClient`'a gönderilmesi için bir kopyalama işlemi gerekir, bunun için `Server` dizinindeki derleme, `WebClient` derlemesinden önce yapılmalıdır; Üst seviyedeki `build.xml` bu sırayı hatırlayacak şekilde hazırlanmıştır.

A.2 Java

Java derleyicisini (`javac`) ve yorumlayıcısını (`java`) kurmak için, `http://¬ java.sun.com/j2se/1.4.2/download.html` sitesine girin ve J2SE 1.4.2 (ya da en son) sürümü tıklayın. Tercihen içinde NetBeans programı dahil edilmemiş olan paketi kurmanız daha iyi olacaktır.

Windows

Windows için sonu exe ile biten kuruluş programını alın. İndirim bitince exe dosyasına tıklayın, ve yönlendirici (wizard) ekranlarını takip edin. JDK'yi nereye kurmak istediğiniz size sorulacaktır.

Unix

İndirmiş olduğunuz `j2sdk-1_4_2_08-linux-i586.bin` adındaki bir dosyayı, komut satırından `root` ya da yeterli hakları olan bir kullanıcı altından

```
\$ j2sdk-1_4_2_08-linux-i586.bin
```

şeklinde işletin. Bu dosya, kendi kendini açan (self inflating executable) türünden bir işler dosyadır, yâni bir tür zip dosyasıdır. Dosya açılınca, `j2sdk-1_4_2_08-linux-i586` adında yeni bir dizin göreceksiniz. Bu dizini alıp, herhangi bir diğer dizine (genelde `/usr/local/`) altına koyarsanız, Java kuruluş işlemini gerçekleştirmiş olursunuz.

A.3 Ant

Ant, Java için vazgeçilmez derleme sistemidir. Herhangi bir görsel IDE'nin sağlayabileceği derlemeden çok daha güçlü ek özellikler taşır. Ant ile her türlü derleme ve alâkalı ek işleri yerine getirebiliriz.

Mesela Ant, shell'den komut çağırma, dosya ve dizin kopyalama, jar oluşturma gibi birçok özelliği `build.xml` içinden kullanamanıza izin verir. Bunlar neden yapılmıştır? Çünkü bir derleme süreci, yanlızca `.java` dosyalarını `.class` dosyalarına çevirmekten ibaret değildir. Bazı ayar dosyalarının bir yerden ötekine kopyalanması, bazı dizinlerin silinmesi (meselâ JBoss altından) ve bir XYZ programının çağırılması gerekebilir. Tüm bunları derleme dilimiz içinde ihtiyaç oldukça yapabiliyor durumda olmalıyız.

A.3.1 Kurmak

Ant'in kuruluş programını `http://ant.apache.org/bindownload.cgi` adresinden indirebilirsiniz. İndirdiğiniz dosya, bir zip ya da tar.gz dosyası olacaktır. Bu dosyayı Ant'i kurmak istediğiniz dizin üzerinde açın. Biz geliştirme ortamı olarak `c:/devprogs` adlı bir dizin altında tüm geliştirme odaklı programları koyuyoruz.

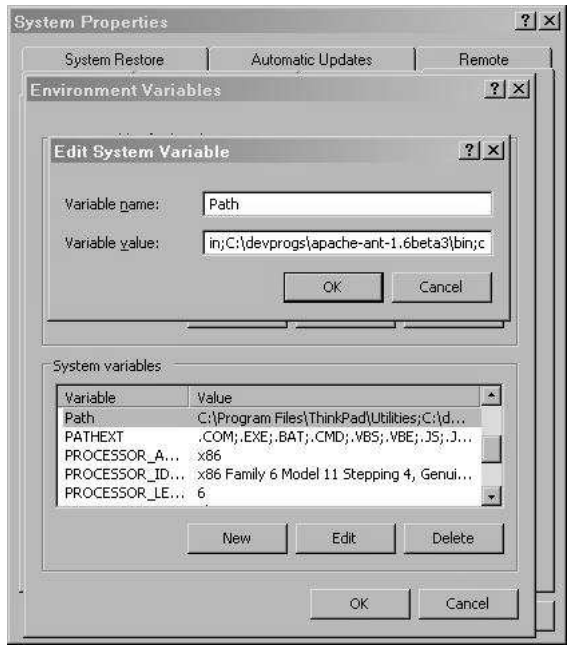
Zip'in açılması bittikten sonra, mesela `c:/devprogs/apache-ant-1.6` gibi bir dizin yaratılmış olur. Komut satırından `ant` komutunu uygulayabilmek için, `c:/devprogs/apache-ant-1.6/bin` dizininin sistem `PATH`'ine eklenmiş olması gerekiyor. Bunun için Windows'da `Control Panel | System | Advanced | Environment Variables` listesindeki `PATH` değişkenine Ant'in `bin` dizinini vermeniz gerekiyor. Şekil A.1 üzerinde bu işlemi görüyoruz.

Aynı şekilde, `JAVA_HOME` değişkeninin de aynı panelden sisteme bildirilmesi gerekiyor. `JAVA_HOME` için JDK'nin kurulmuş olduğu dizini veriniz. Java kuruluşu için A.2 bölümüne bakınız.

A.3.2 Kullanmak

Ant ile bir projeyi derlemek istiyorsanız, o projenin (genelde en üst) dizini içinde `build.xml` adında bir dosya bulmanız gerekir. Bu dosya içinde Ant için lazım olan tüm derleme komutları mevcuttur. Derleme işlemini başlatmak için

```
ant
```



Şekil A.1: Ant Dizinini PATH'e Eklemek

komutunu, `build.xml`'in olduğu aynı dizinden başlatırsanız, Ant otomatik olarak o dizindeki `build.xml` dosyasını kullanacağını bilecektir. Onu okur ve içindeki derleme işlemlerini başlatır.

Eğer geliştirme sırasında herhangi bir sebebten ötürü komut satırından (`cd` ile) daha alt dizinlere indiyseniz, `ant` komutunun alt dizinden başlayıp üst dizinlere çıkarak en yakındaki `build.xml`'i bulup işletmesi için, `ant -find` komutunu kullanabilirsiniz.

Derleme komutlarımızı, ismi `build.xml` *olmayan* bir dosyada tutmak istersek (bu pek nadir yapılır), o zaman, meselâ `my-build.xml` adında bir dosya için,

```
ant -f ./my-build.xml
```

komutunu kullanabiliriz.

A.3.3 Build.xml

Ant kullanmak, `build.xml` içinde Ant *task*'leri kullanarak *target*'ler yaratmak demektir. Task, daha önceden yazılmış ve etiketi sağlanmış Ant komutudur. Meselâ Java kodu derlemek için kullanılan `javac` bir Ant task'idir. Bu task, ismi tipik olarak `compile` olan bir target içinde kullanmak gerekir.

Bir target'i, komut satırından ismini belirterek direk çağırabiliriz.

Bir target, `build.xml` içindeki diğer bir target'i çağırabilir, ya da bir target ötekine "dayanıyor" (depends) olabilir; O zaman "dayanılan" target, ilk çağırılan target işletilmeden *önce* işletilecektir.

Başlangıç

Her `build.xml`, `<project>` etiketi ile başlar.

```
<project name="blank" basedir="." default="all">
...
</project>
```

Burada proje ismi bir tanımdan ibarettir, `basedir` projenin ana dizininin neresi olduğunu belirtir ("`.`" içinde bulunan dizin demektir), ve `default` ise Ant işletilirken target belirtilmezse, hangi target'in farz edilmesinin gerektiğini (default) belirtir.

Sabit Değerler

```
<property file="build.properties"/>
```

şeklinde bir kullanım ile, sabit değişkenleri alışık olduğumuz bir `.properties` dosyasından alabiliyoruz. Bu dosya içinde tanımlanan her `variable=vsvs` şeklindeki değişkene, `build.xml` içinden `\${variable}` şeklinde erişebiliriz.

Classpath

`javac` ve `java` Ant task'lerine bir `CLASSPATH` gerekir (ayen komut satırında `java` ve `javac`'nin aynı bilgilere ihtiyaç duyduğu gibi).

Ayrıca, aynı classpath, hem derleme hem işletmek amaçlı gerektiği için, tek bir kez tanımlanması faydalıdır. Bu tanımı merkezileştirmek için `<path>` etiketi kullanılır.

```
<path id="compile.classpath">
  <pathelement location ="build/WEB-INF/classes"/>
  <pathelement location="‘\{jboss.home}/lib/jboss-jmx.jar"/>
  <fileset dir="lib">
    <include name="**/*.jar"/>
    <include name="**/*.zip"/>
  </fileset>
  ....
</path>
```

Bir `<path>` tanımlarken, hem `jar`, hem de direk dizin ismi kullanılabilir. Ayrıca bir dizin altında belli isim düzenine uyan bir dosya grubu da kullanılabilir. Bu `<fileset>` task'i ile başarılmıştır.

Tanım yapıldıktan sonra, bu dizin, dosya topluluğuna sadece `compile.¬ classpath` ismi ile referans edilebilecektir.

Derlemek

Derlemek için (meselâ `compile` adında bir target içinde) `javac` task'i şöyle kullanılır.

```
<target name="compile" depends="prepare,resources">
  <javac srcdir="src/java" destdir="./build/classes" debug="true">
    <classpath refid="compile.classpath"/>
  </javac>
</target>
```

- `srcdir`, tüm Java kaynaklarının başladığı üst dizini belirtmektedir. Ant, buradan başlayarak tüm alt dizinlerdeki tüm `.java` dosyalarını bularak derleme işlemini yapacaktır.
- `destdir` ise, sonuç dosyaları olan `.class` dosyalarının nereye konulacağını belirtir.
- `debug` ise, bir Exception atıldığında bu hata içinde hatanın hangi Java satırından geldiği bilgisinin hataya dahil edilip edilmemesini kontrol eder. Geliştirme sırasında bu seçeneğin `true` olmasını şiddetle tavsiye ediyoruz.
- `<classpath refid>` ise, daha önce tanımladığımız classpath dizini, sadece ismini kullanarak referans ederek kullanmaktadır.

Bir Java Class'ı İşletmek

Bir Java programını çağırmak için komut satırında işletilecek Java class'ın paketiyle beraber ismi, ve bulunabileceği bir `CLASSPATH` yeterli olurdu (tabii bir de Java class'ının içinde bir `main()` metotu gerekir). Bu işi Ant ile yapmak istiyorsak, şöyle bir kullanım gerekir.

```
<target name="xxx" depends="clean,compile">
  <java fork="yes" classname="org.mycompany.kitapdemo.vs.MyClass"
       taskname="xxx" failonerror="true">
    <classpath refid="compile.classpath"/>
  </java>
</target>
```

- `fork`, Java sürecinin çağıran sürecin parçası olup olmayacağını kontrol eder.
- `classname` paket ismiyle beraber class ismini alır.
- `failonerror` hata olursa durulup durulmayacağıdır.

Burada en ilginç gelecek kullanım, ilk kez bir "dayanma" (depends) ilişkisinin gösterilmesidir. Bu target'in tanımına göre, `depends="clean, compile"` ile `xxx` target'i işlemeden *önce* `clean` ve `compile` target'lerinin işlemesi şart konulmuştur. Bu da normâldir, bir class'i işletmeden önce derleme yapmak isteriz.

Ant, ayrıca sadece değişen dosyaları derlemeyi bilecek kadar akıllıdır. Yâni, eğer bir önceki derleme ile o an arasında hiç Java dosyası değiştirmediyseniz, derleme işleme hiçbir şey yapmadan geri dönecektir.

Komut Satırına Çıkma

Ant, normâlde shell üzerinden çağırabileceğiniz her komutu, `build.xml` içinden çağırabilir. Bunun için `<exec>` task'i kullanılır.

```
<exec executable="command">
  <arg line="-filan"/>
  <arg line="param1"/>
  <arg line="-falan"/>
  <arg line="param2"/>
</exec>
```

Bu komut, eğer komut satırında verilseydi, şöyle gözükecekti:

```
\$ command -filan param1 -falan param2
```

Dosyayı Dosyaya Kopyalamak

Bir dosyayı diğer bir dosya ismiyle (hâtta başka bir dizine) kopyalamak için, şu komut kullanılır.

```
<copy overwrite="true"
      file="./from/directory/myFileName.txt"
      tofile="./to/directory/otherFileName.txt"/>
```

Dosyayı Dizine Kopyalamak

Bir dosyayı diğer bir dizine aynı isimle kopyalamak için, şu komut kullanılır.

```
<copy overwrite="true"
      file="./from/directory/myFileName.txt"
      toDir="./to/directory"/>
```

Birçok Dosyaları Kopyalamak

Bunun için, classpath tekniğinden tanıdık gelebilecek `<fileset>` kavramını kullanıyoruz.

```
<copy todir="/to/dir" includeEmptyDirs="yes">
  <fileset dir="src">
    <patternset>
      <include name="**/*.xml"/>
    </patternset>
  </fileset>
</copy>
```

Bu komutla, `build.xml`'in olduğu dizinin altındaki `src` dizini içindeki, her `.xml` ile biten dosya, dizin yapısıyla beraber, `/to/dir` adlı dizine postalanacaktır.

Dizin Silmek ve Yaratmak

Dizin silmek için (`dir` isminde bir dizin için meselâ)

```
<delete dir="dir"/>
```

eklemek için ise,

```
<mkdir dir="dir"/>
```

kullanılır.

A.4 JBoss

Kurulması en rahat uygulamalardan biri herhalde JBoss'tur. Tek yapmanız gereken, `http://www.jboss.com/products/jbossas/downloads` adresinden en son JBoss zip dosyasını indirmektir. Kurmak için şunları yapın.

Windows

Dosyayı bir dizinde açın. JBoss'un kurulmuş olduğu dizine, `JBOSS_HOME` diyelim. JBoss servisini başlatmak için, `JBOSS_HOME/bin/run.bat` dosyasına tıklarsanız, servis başlayacaktır. Windows üzerinde `JAVA_HOME` değişkeni tanımlanmiş olduğu için, daha fazla bir değişiklik yapmanız gerekmemiştir.

Linux

JBoss kodları pür Java kodu oldukları için hem Windows hem de Unix ortamında hiç değişiklik gerektirmeden çalısabilirler. İndirdiğimiz zip dosyasını Unix'e kopyalayıp orada açarsak, JBoss'u Unix ortamında hiç değişiklik gerektirmeden işletebiliriz. Unix'te bir zip dosyasını açmak için, `unzip` komutu kullanılır. JBoss zip'ini açmak için

```
unzip jboss-4.0.1.zip
```

komutunu kullanın. Şu anda içinde bulunduğunuz dizinde `jboss-4.0.1` adında bir ek dizin oluşturulmuş olması gerekiyor. Bu dizin altında `./bin/run.conf` dosyasına girin. Bu dosya içinde

```
#JAVA_HOME="/opt/java/jdk"
```

şeklinde comment edilmiş bir satır bulacaksınız. Bu satırı comment-out edip, `JAVA_HOME` değişkenine A.2 bölümünde JDK'yi kurmuş olduğu dizini eşitleyin. Artık JBoss'u başlatmak için

```
sh JBOSS_HOME/bin/run.sh
```

komutunu kullanabilirsiniz.

A.4.1 Deploy Dizinleri

Tabii ilk kurulduğu haliyle JBoss'un içinde sadece örnek uygulamalar mevcuttur. Eğer kendi uygulamanızı JBoss'a dahil edip çalıştırtmak istiyorsanız, uygulamanızın nereye gideceğini bilmeniz gerekiyor.

Bir JBoss uygulaması, bir WAR, EAR ya da SAR paketleri içinde olabilir. Bir JBoss paketi, aslında uygulamanın çalışması için gereken tüm dosyaları içeren bir dizin ya da bir dosyadır. Eğer paketi tek bir dosya olarak görmek istiyorsanız, meselâ `myapp.ear` adında bir dosya üretebilirsiniz. Bu üretimi her zaman Ant script'lerimize yaptırırız, çünkü belli dosyaların paket içinde belli dizinler altında olması gerekmektedir (J2EE standartı bunu belirler) ve Ant bu işlemleri kodlayabileceğimiz uygun yerdir.

Tek bir dosya içinde bir EAR, WAR ya da SAR yapısını koymak, *kapalı dosya* kullanımıdır. Açık dosya kullanımı ise, biraz önce paketin içine (bir sıkıştırma programı ile) koyduğumuz tüm dosyalara ve dizinlere hiç dokunmadan, ama aynı sonek ile (meselâ `.ear`) olduğu gibi bir *dizini* JBoss'a göndermektir. Yâni `myapp.ear` dosyası yerine `myapp.ear/` gibi bir dizin ismini JBoss'a verirsek, JBoss hiç bir fark gözetmeden bu dizini de işleme koyacaktır.

Tercihimiz hangisi? Biz açık paket şeklini tercih ediyoruz. Bunun sebepleri için 3.3 bölümündeki anlatıma bakabilirsiniz.

Uygulama paketimizi oluşturduktan sonra göndermemiz gereken yer, çoğu zaman ve geliştirme amaçlı olarak `JBOSS_HOME/server/default/deploy` dizini olacaktır. Bu dizin, JBoss için özel bir dizindir; JBoss, başlar başlamaz bu dizinde ne olup olmadığına bakar. Bu dizinde olan ve sonu `sar`, `ear` ya da `war` ile biten dizinlerin ya da dosyaların bir kurumsal uygulama olduğuna kanaat getirir, ve onların içindeki ayar dosyalarına giderek (J2EE standartına göre nerede oldukları bellidir) uygulamanızı işleme koyar.

Peki neden `JBOSS_HOME/server/default` altındaki `deploy` dizinine bakılmaktadır? Başka bir dizin altındaki `deploy` dizinine bakılmaz mi? Bu kesinlikle yapılabilir. Eğer JBoss'u başlatırken komut satırında

```
bin/run.bat -c newdirectory
```

gibi bir komut verirseniz, JBoss uygulamaları `JBOSS_HOME/server/newdirectory/deploy` altında arayacaktır. Fakat genelde `-c` seçeneği ile *hiçbir dizin verilmediği için* JBoss, `default` dizininin istendiğini farz eder ve uygulamaları orada arar. Bu kullanım, küme ile çalısmamız gereken ortamlarda değişecektir.

A.4.2 Geliştirme Amaçlı Port Değiştirmek

Bazen birden fazla kullanıcının ortak bir Unix makinasında geliştirme yapması ya da test programlarını işletmesi gerekebilir. Birden fazla geliştiriciyi tek bir JBoss kuruluşundan desteklemek için, `JBOSS_HOME/server/default` dizininin bir kopyasını çıkartarak, meselâ `JBOSS_HOME/server/user1` gibi bir dizin yarabilirsiniz. İkinci bir kullanici için `JBOSS_HOME/server/user2` olabilir, vs. Böylece meselâ birinci kullanıcı, kendi JBoss'unu başlatmak için

```
sh run.sh -c user1
```

komutunu kullanabilir. Fakat işimiz daha bitmedi: Bu düzende eğer hiçbir değişiklik yapılmazsa ve kullanıcı ötekinden habersiz bir şekilde `sh run.sh` ile birden fazla JBoss servisini aynı makine üzerinde çalıştırmaya kalkışırsa, ikinci JBoss'u başlatan kullanıcı, kullandığı portların "başkası tarafından kullanılmakta olduğuna" dair bir mesaj görecektir. Bunun sebebi de basittir. JBoss içindeki HTTP, RMI, JNDI gibi J2EE servisleri, kendilerini dış dünyaya afişe etmek bir port'a ihtiyaç duyarlar. `sh run.sh` kullanan herkes `JBOSS_HOME/server/default` dizinin bir kopyasını kullandığı için, o dizin altında tanımlanmış ve *aynı olan* port değerlerini kullanıyor olacaktır. Bu sebeple ikinci servisi başlatan kullanıcının port çakışma hatası görmesi çok normâldir.

Geliştirme ve test amaçlı olarak bu hatadan kurtulmak için, tarafımızdan `JBOSS_HOME/server/default` altında port tanımı yapan her dosya çıkartılıp, port değerlerinin baş tarafına 1'den 6'ya kadar olan sayılar eklenerek, ayrı ayrı dizinlerde yeni ve birbirinden değişik port tanım dosyaları yaratılmıştır.

Bu yeni tanım dosyalarını almak örnek kodlar ve araçların adresinden `kitap-tools-jboss-4.0.1-portlar.zip` adlı dosyayı indirin. Bu dosyayı geçici bir dizin altında açın.

```
+- jboss-4.0.1-portlar
| +- 1-port
| | +- conf
| | | +- jboss-service.xml
| | +- deploy
| | | +- jbossweb-tomcat50.sar
| | | | +- server.xml
| | | | +- jms
| | | | | +- uil2-service.xml
| +- 2-port
| | +- conf
| | | +- jboss-service.xml
| | +- deploy
| | | +- jbossweb-tomcat50.sar
| | | | +- server.xml
| | | | +- jms
| | | | | +- uil2-service.xml
...
```

Görüldüğü gibi port dizinleri, `1-port` ile başlayıp `6-port`'a kadar devam edecektir. JBoss'unuzun hangi port başlangıç değeri ile başlamasını istiyorsanız, o başlangıç değerine ait olan alt dizinin altındaki herşeyi olduğu gibi alın, ve kendi ayrı JBoss kuruluşunuzun (meseâ `JBOSS_HOME/server/user1`) dizini altına bırakın. Aynı şeyi ikinci kullanıcı `user2` altında yapacaktır. Böylece `sh run.sh -c user1` ve `sh run.sh -c user2` komutları birbiri ile çakışmamış olur.

A.4.3 Küme Ortamında Port Değiştirmek

Küme ortamında ve yine test amaçlı olarak, eğer aynı makinada bir küme oluşturmak istiyorsak, yine port değişikliği yapmamız gerekiyor (kümeler hakkında detayları 5.4.8 bölümünde bulabilirsiniz).

Ama dikkat: JBoss, bir küme işletebilmek için gerekli olan ayarları `JBOSS_¬ HOME/server/default` altında değil, `JBOSS_HOME/server/all` altında tutmaktadır. Eğer test hattâ sonuç ortamınız her makinada bir JBoss JVM'i olmak üzere planlanmışsa, o zaman her makinada

```
sh run.sh -c all
```

komutunu kullanarak kümenizi başlatabilirsiniz. JBoss, içindeki JGroups kütüphanesinin yardımıyla multicast protokolu üzerinden network üzerindeki diğer küme birimlerini otomatik olarak bulacaktır.

Fakat elinizde tek bir makina var ise, ve bu tek makinada bir küme testi yapmak istiyorsak, yine port değişikliği yapmamız gerekecek. Ve A.4.2 bölümünde anlatılan tekniği burada kullanamayız. Ama aslında küme şartlarında port değiştirme işlemi daha da basit olacak.

Aynı JBOSS kuruluşu altında (ve aynı makinada) yeni bir küme birimi yaratmak için, `JBOSS_HOME/server/all` dizinini `JBOSS_HOME/server` altında `node1` ve `node2` olarak olarak kopyalayalım. Daha sonra meselâ `node1` altındaki `/conf/jboss-service.xml` dosyasına girelim, ve burada "ports" kelimesini arayalım. Şu şekilde bir kod tanımını bulacaksınız:

```
<!--
   | ...
<mbean code="org.jboss.services.binding.ServiceBindingManager"
  name="jboss.system:service=ServiceBindingManager">
  <attribute name="ServerName">ports-01</attribute>
  <attribute name="StoreURL">
     ../docs/examples/binding-manager/sample-bindings.xml
  </attribute>
  <attribute name="StoreFactoryClassName">
    org.jboss.services.binding.XMLServicesStoreFactory
  </attribute>
</mbean>
-->
```

Bu tanım parçasınının, <–! ve --> işaretleri ile sarılmış olduğuna dikkat edelim; XML tanımlarında bu işaretler *comment* için kullanılır, yâni üstte gördüğümüz tanım, aslında *iptal edilmiş* bir tanımdır. Küme ortamında port değişikliği için, bu tanımı önce aktif hâle getirmemiz gerekiyor. Eğer <–! ve --> işaretlerini doğru yerlere koyarsak, yâni üstteki tanımı

```
<!--
   | ...
-->
<mbean code="org.jboss.services.binding.ServiceBindingManager"
  name="jboss.system:service=ServiceBindingManager">
  <attribute name="ServerName">ports-01</attribute>
  <attribute name="StoreURL">
     ../docs/examples/binding-manager/sample-bindings.xml
  </attribute>
  <attribute name="StoreFactoryClassName">
    org.jboss.services.binding.XMLServicesStoreFactory
  </attribute>
</mbean>
```

durumuna getirirsek, o zaman tanım aktif hâle gelmiş olur (sadece --> işaretini alttan, tanım bloğununun üstüne taşımış olmakla) ve JBoss'umuz `ports-01` kimlikli port tanımını kullanmaya başlar. `ports-01`, üstte belirtilen `../docs¬ /examples/binding-manager/sample-bindings.xml` dosyasındaki bir kimlik değeridir.

İkinci küme birimi içinse, `JBOSS_HOME/server/node2` altındaki `/conf/¬ jboss-service.xml` dosyasına girip, aynı comment değişikliğini yapıp, bu sefer `ports-01` yerine `ports-02` kullanmamız gerekecektir. Bu son değişikliği yapmakla, `sh run.sh -c node1` ve `sh run.sh -c node2` ile başlatılacak JBoss'ların birbirleri ile çakışması engellenmiş olur.

A.5 Linux

Örnek programları kodlarken onları işletebilmek için bir servis ortamına ihtiyacımız vardı: Test makinamızda Linux işletim sistemini kullandık. İstanbul Mecidiyeköy'de mbSan şirketinde toplattırdığımız Intel bazlı, network ve grafik kartı ana kart üzerinde dahil olan ve 1 GB hafıza, 2 Ghz işlemci, 40 GB disk kapasitesine sahip olan bir servis makinasında Suse Linux kuruldu (bu donanıma 600 YTL ödenmiştir).

Dizüstü geliştirme bilgisayarından kod gönderimi, ve `ssh` ile sisteme girebilmek için, makinayı yerel bir ağa dahil etmek gerekti. Bunun için her iki bilgisayar bir hub'a bağlandı (bir RJ-45 kablosu ile iki bilgisayar arasında direk kablo çekilmesi bir işe yaramayacaktır). Linux kurulumu, paketten çıktığı hâliyle network'e hazırdır, fakat buna ek olarak servis makinasına bir IP adresi verilmesi gerekecek. En basit yöntem olarak bir statik bir IP vermek için Eth-

ernet kartı `eth0` için (eğer tek kart var ise) `/etc/sysconfig/network/ifcfg-eth0` dosyasında

```
STARTMODE="onboot"
BOOTPROTO="static"
BROADCAST="10.10.255.255"
IPADDR="10.10.11.184"
NETMASK="255.255.0.0"
```

tanımları yaparsanız bilgisayarınız başladığında yeni bir IP adresi olacaktır. Bu ayarlara göre, bilgisayarımızın IP adresi `10.10.11.184` olarak seçilmiştir. Bundan sonra eğer bu makinayı görmek istiyorsanız, dizüstü bilgisayarınıza aynı alandan (`10.10.11.` ile başlayan) bir IP adresi verebiliriz. Bundan sonra `telnet` ve `ssh` ile Linux makinanızı erişilebilir hâle gelecektir.

Linux'u reboot etmek için, `root` olarak

```
\$ shutdown -r now
```

komutunu, sadece durdurmak için

```
\$ shutdown -h now
```

komutlarını kullanabilirsiniz. Seçenek `-r` reboot, `-h` ise halt kelimeleri için birer kısaltmadır.

A.6 Cygwin

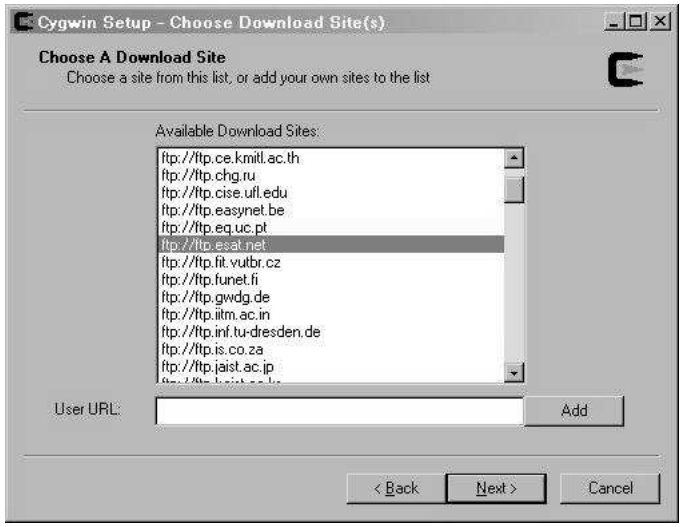
Cygwin programımı kurmak için, `http://www.cygwin.com` sitesini ziyaret edelim. İlk sayfa yüklendikten sonra, Şekil A.6 üzerinde gözüken sağ taraftaki `Install Cygwin Now` ikonu üzerine tıklamamız gerekiyor. Bunun sonucunda bize `setup.exe` programını indireceğimiz bir yer sorulacaktır. Bir yer seçelim, ve program indirildikten sonra üzerinde tıklayarak kuruluş işlemini başlatalım.

Kuruluşun ilk ekranında hemen `Next` tuşuna basabiliriz. Bundan sonra, çıkan üçlü seçenekten `Install from Internet` seçeneğini seçelim. `Root Directory` olarak `c:/cygwin` dizinini seçebiliriz. Tekrar `Next` tuşundan sonra, kuruluş dosyalarını nereye koyacağını soracaktır. Burada `c:/installs` gibi bir dizin ismi verebiliriz. `Select Your Internet Connection` seçeneği ile, kuruluş işleminin Internet'e nasıl bağlacağını tanımlıyoruz. `Direct Connection` uygundur. Şekil A.3 üzerinde gösterilen bir sonraki ekran, Cygwin'in hangi siteden indirileceğini sorar.

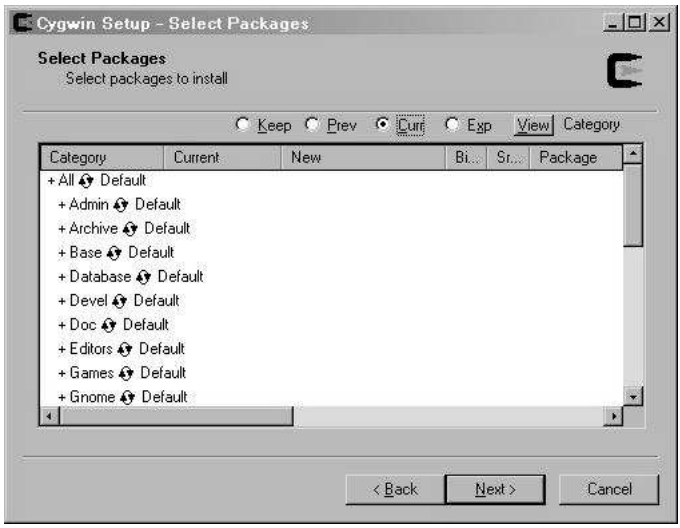
Bir sonraki ekran, Cygwin'in indirilecebileceği mevcut sitelerin bir listesidir. Buradan bir site ismini seçebiliriz, ve `Next` tuşuna basarız. Bu aşamada, kodların indirileceği siteden mevcut paketlerin bir listesi alınacaktır. Bu yükleme aşaması 10-15 saniye sürebilir. Cevap gelince, kurulacak Cygwin paketlerin listesi alttaki Şekil A.4 üzerindeki gibi gözükecektir.

Dikkat: `Next` tuşuna basıp kuruluşu başlatmadan önce, bir işlem daha yapmamız gerekiyor. En son listede bize lazım olacak tüm programların hepsi halen

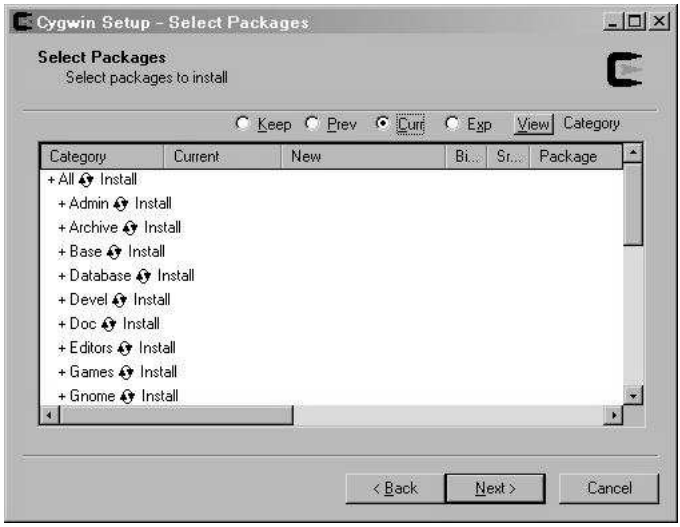
Şekil A.2: Cygwin Sitesi



Şekil A.3:



Şekil A.4:



Şekil A.5:

seçilmiş değildir. Bu seçimi yapmak için, en üstteki `All` yazısının yanındaki yuvarlak işaretinin üzerinde "sadece bir kez" tıklamamız gerekiyor. Bunu yaptıktan sonra bekleyin, ve ekran Şekil A.5 üzerinde görülen hâle dönüşecek.

Gördüğümüz gibi `Default` yazıları `Install` yazılarına dönüştü. Artık bir

daha `Next` tuşuna bastığınızda, Cygwin altındaki tüm paketler kurulmaya başlanacaktır. Bu kuruluş aşaması çok uzun zaman sürebilir! Tavsiyemiz, bu kuruluş işlemini makinanın başka bir işi olmadığı zaman yapmanızdır. Fakat Cygwin paketinin tüm bekleyişinize değen bir paket olduğunu söylememiz gerekiyor; Ne de olsa Windows üzerinde Unix komut satırına ve komutlarına sahip olmak, yabana atılır bir avantaj değildir.

A.7 MySQL Front

MySQL-Front, MySQL veri tabanı için en popüler önyüzlerden biridir. Bu programı kullanarak veri tabanlarını görebilir, sorgu yapabilir, görsel olarak tablo, kolon ekleyebilir ve bir tablo içindeki veri satırlarının (belli bir limit dahilinde) listesini görebilirsiniz.

Programı indirmek için `http://www.mysqlfront.de/download.html` adresinden en son sürümü alın. Kurulması, standart bir Windows programı gibidir ve oldukça basittir.

A.8 OpenSSH

`ssh` ve `scp` programlarını kurmayı, sadece Windows ortamı ve Cygwin (A.6) kurulmadığı şartlar için anlatacağız çünkü eğer geliştirme ortamınız Linux ise `ssh` zaten işletim sisteminde kurulmuş olarak gelecektir. Solaris ortamında ise eğer `ssh` yoksa (çoğunlukla vardır) makinadan sorumlu admin'e bu programı kurdurtabilirsiniz. Hattâ Windows ortamında da, eğer Cygwin kurmuşsanız `ssh`, Cygwin paketinin içinden kullanıma hazır bir hâlde çıkacaktır.

Windows'da `ssh`'i ve `scp` programlarını kurmak için, önce `http://sshwindows.sourceforge.net/download/` adresinden `Binary Installer Release`'i indirin. Kuruluş programını başlatın ve sadece `Client` için kuruluşu seçin. Kurulum yaptığınız makina, üzerinde geliştirme yaptığınız makina olduğu için, sadece kod *göndermekle* yükümlü olacaktır, bu yüzden bir `ssh` müşterisi (client) olması yeterlidir.

Kuruluş ekranlarını takip edin, istediğiniz bir kuruluş dizini seçin, ve kuruluşu tamamlayın. Şimdi yeni açtığınız bir komut satırı ekranından `ssh` ya da `scp` komutlarını işletebilirsiniz.

A.9 ITracker

`http://sourceforge.net/project/showfiles.php?group_id=54141`

adresinden `Download` bağlantısını takip ederek oradan `itracker_xxx.express¬ .zip` dosyasını bulup indirin. Bu zip tamamen kendi kendine yeterli (self-contained) bir JBoss da içermektedir, yâni hazırlıklar tamamlandıktan sonra

tek yapmanız meselâ `ITRACKER` dizini altındaki kurulumu `ITRACKER/jboss-3.2.5/bin` altındaki `run.bat` ya da `run.sh` ile başlatmaktır.

İlk önce ITracker'ın verilerini tutacak MySQL'de çalışan yeni bir veri tabanına ihtiyacımız var. Bu veri tabanının ismi `itracker` olsun. Veri tabanını MySQLFront (A.7) ile yaratabiliriz. Taban hazır olunca, tabanı doğru tablolarla doldurmak için `ITRACKER/sql/mysql/install` altındaki `create_itracker_core.sql` ve `create_mysql_user.sql` dosyalarını işletmemiz gerekecektir.

Bunlardan sonra, JBoss ITracker'ın veri tabanından haberdar olması için, JBoss üzerinde `ITrackerDS` adında bir J2EE veri kaynağı (data source) yaratmamız lazım. Bunun için bildiğimiz gibi `deploy` dizini altında sonu `-ds.xml` ile biten bir dosya yeterli oluyor. Biz ITracker için `itracker-ds.xml` adında bir dosya yarattık. Bu örnek dosyayı kitap kodları altındaki `conf` dizininde bulabilirsiniz. Ayrıca `ITRACKER/jboss-3.2.5/server/default/deploy/it-hsqldb-ds.xml` adındaki dosyayı silin, çünkü paketten çıktığı haliyle ITracker HSQLDB kullanmak üzere ayarlanmıştır. `ITrackerDS`'in tanımını altta veriyoruz.

```
<?xml version="1.0" encoding="UTF-8"?>
<datasources>
  <local-tx-datasource>
    <jndi-name>ITrackerDS</jndi-name>
    <connection-url>
      jdbc:mysql://localhost:3306/itracker
    </connection-url>
    <driver-class>org.gjt.mm.mysql.Driver</driver-class>
    <user-name>root</user-name>
    <password></password>
    <exception-sorter-class-name>
        org.jboss.resource.adapter.jdbc.vendor.MySQLExceptionSorter
    </exception-sorter-class-name>
    <metadata>
       <type-mapping>mySQL</type-mapping>
    </metadata>
  </local-tx-datasource>
</datasources>
```

Son bir basamak, MySQL için JDBC sürücüsü (driver) jar dosyasını JBoss'a vermek, bu da yine kitap kodları altında bulabileceğiniz `lib/hibernate/mysql-connector-java-3.1.11-bin.jar` dosyasını `ITRACKER/jboss-3.2.5/server/default/lib` dizini altına atmaktır.

Artık `localhost:8080/itracker` adresinden programı kullanmaya başlayabilirsiniz. Program otomatik olarak ilk kullanıcıyı yaratıyor, kullanıcı: `admin`, şifre: `admin` ile programa girip projeniz için gerekli bilgileri girmeye başlayabilirsiniz.

ITracker resmi sürümü, Türkçeleştirilmiştir durumdadır. Kullanıcı olarak sisteme girdikten sonra, `Tercihlerim` (My Preferences) seçeneği altından Türkçe dilini seçerek uygulamanın dilini tamamen değiştirmeniz mümkündür.

A.10 Linux Üzerinde CVS

CVS'i kurmak için öncelikle

```
ftp://ftp.gnu.org/gnu/non-gnu/cvs/
```

adresinden en son cvs kaynak sürümünü indirin. Bu dizin altında bugün için en son sürüm `cvs-1.11.tar.gz` dosyasıdır. Bu dosyayı Unix makinanıza kopyalayıp `tar xvzf` ile açabilirsiniz. Açılan yeni dizin içinde

```
./configure
make
make install
```

komutlarını sırasıyla uygulayın. Bu komutlar CVS işler kodlarını gerekli yerlere koyacaktır.

CVS servisini başlatabilmek için `cvs` adında bir kullanıcı ve `cvs` adında bir grup yaratmanız gerekiyor. Şu komutları sırasıyla uygulayın.

```
groupadd cvs

mkdir /home/cvs

chgrp cvs /home/cvs

useradd cvs -d /home/cvs -g cvs

chown cvs /home/cvs
```

Bu satırlar ile `cvs` Unix grubu altında olan bir `cvs` kullanıcısı yarattık. Bunları yaptıktan sonra `/etc/inetd.conf` dosyası içine şu satırları eklemelisiniz.

```
cvspserver stream tcp nowait root /usr/bin/cvs cvs
          --allow-root=/home/cvs pserver
```

Ek olarak `/etc/services` dosyası içinde şu satırların olmasına dikkat etmeliyiz:

```
cvspserver 2401/tcp
```

Artık `inetd` servisini tekrar başlatırsak (`killall -HUP inetd`), yeni ayarlar devreye girmiş olacaktır. Eğer Unix sistemimiz xinetd kullanıyorsa, ek bir aşama olarak `/etc/xinet.d/cvspserver` dosyasına şunları yazmalıyız

```
service cvspserver
{
  socket_type = stream
  wait    = no
  user    = cvs
  group   = cvs
  env   = HOME=/home/cvs # Fixes RHL 7.0 problem!
  server    = /usr/bin/cvs
  server_args = -f --allow-root=/home/cvs pserver
```

```
  disable  = no
}
```

Bu ayarları `xinitd` ile devreye sokmak için, `root` kullanıcısından

```
/etc/init.d/xinetd restart
```

komutunu çağırmalıyız. Bunlar yapıldıktan sonra, sisteme yeni bir CVS kullanıcı eklemek, o kullanıcıyı yeni `cvs` Unix grubuna eklemek kadar basittir. Bunu

```
/usr/sbin/usermod -G cvs user123
```

sözdizimini kullanarak yapabiliriz.

A.10.1 Kullanmak

CVS'e bağlanan sistemler için bazı ayarlamalar gerecektir. Her CVS kullanıcısı, öncelikle hangi havuzda işlem yapacağını belirtmelidir. `cvs -d havuz_ismi` seçeneği ile bunu rahatça yapabiliriz, ya da `CVSROOT` çevre degişkeninde havuz ismini tanımlayabiliriz; Böylece sürekli `-d` kullanılmasına gerek kalmaz.Eğer kod havuzu yerel ise (aynı makina üzerinde yani), o zaman

```
export CVSROOT=/tmp/deneme1
```

kullanılabilir. Fakat havuz ile kullanıcı ayrı sistemlerde ise (ki genelde böyle olur), o zaman

```
export CVSROOT=:pserver:remoteuser@hostname:/tmp/deneme1
```

Şimdi bu ayarları kullanarak CVS'e giriş yapabiliriz.

```
cvs login
```

Cevap olarak alttaki çıktı gelecektir.

```
(Logging in to hostname)
CVS password: ...
```

Şifrenizi girdikten sonra, `cvs co MODULE` komutunu kullanarak (`MODULE` yerine kendi kod havuz ismimizi koymayı unutmayalım) depolanmış kodları kendi yerel diskimize alabiliriz.

Kodlar yerel dizine geldikten sonra, istediğimiz dosyada değişliklik yapmakta serbestiz. Yaptığımız değişikliklerden tatmin olduysak, o zaman kodu geri koyabiliriz. En üst dizin seviyesinden,

```
cvs commit
```

komutunu verince, bu komut en baştan başlayıp her alt dizini ziyaret ederek orada bulduğu değişiklikleri havuza gönderecektir.

Eğer biz başkalarının yaptığı değişiklikleri görmek istiyorsak, yine en üst seviyeden,

```
cvs up -d
```

komutunu verebiliriz. Bu komut bizim dokunmadığımız dosyaların en son hâlini alacak, fakat hem bizim hem de başka bir programcının yaptığı değişiklikleri birleştirim (auto-merge) bize sunacaktır. Bu dosyaların hangileri olduğu bize ekrandan bildirilecektir.

Eğer `cvs commit` komutunu gönderdiğimizde güncelleme çakışması var ise, `cvs commit` hata verip geri döner. Bunun anlamı, yukarıda gösterildiği gibi `cvs up -d` ile auto-merge yaparak çakışmaları çözmeye mecbur olduğumuz anlamına gelir.

Çakışma Örneği

Programcılar, projenin son hâlini almak ve kendileri kodlamaya başlamak için depodan dosyaları kendi ortamlarına indirirler. Dosya üzerinde ekleme, çıkarma işini kendi ortamlarında yaparlar. İşleri bitince KKI sistemine "geri" verirler. KKİ sistemi, eğer "aynı anda" iki kişinin değistirdiği bir dosya varolduğunu bulursa, değişim "çarpışması" olduğunu haber verir. Bu haber aynı dosyayı ikinci geri veren programcıya gösterilir. Böyle bir durumda programcının, değişmiş dosyayı depodan çıkartıp, kendi sürümü ile 'birleştirmesi' (merge) gerekir. CVS programı işimizi rahatlatmak için otomatik bir birleştirici sunmaktadır.

Meselâ, `BeniDegistir.c` dosyasını iki kişi aynı anda değiştirmiştir. İşleri bittikten sonra, belli aralar ile şu işlemı yapmıslardır;

```
cvs commit BeniDegistir.c
```

Bu komutu veren ikinci programcı bir hata mesajı görecektir, ve depo ikinci sürümü kabul etmeyecektir. Çünkü ikinci değişiklikten önce "başkası" dosyayı değiştirmiş, ve ikinci kişi eklememimizi ondan bir önceki sürüme göre yapmıştır. CVS şunu demeye çalışmaktadır; "Bir de kodun son haline bir bakın, eğer değişikliğiniz hala geçerliyse, bana birinci değişiklikle beraber tekrar geri vermeniz gerekiyor. Eski değişikliğin üzerinde yazıp onu kaybetmek istemiyorum". Bu durumda, en yeni (birinci değişiklikten sonraki) sürümü depodan alıp, kendi dosyamız ile "birleştirmemiz" gerekecektir. Dikkat edelim, bu birleştirme hala merkezi depoda değil, bizim şahsi dosyamızda olacaktır.

```
cvs -q co -P BeniDegistir.c
```

Bu komuttan sonra, elinizde şöyle bir kayıt gececek..

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc,
         char **argv)
{
    init_arayici();
    ara();
```

```
    if (argc != 1)
    {
        fprintf(stderr, "tc: Hic Oge Gondermeye Gerek Yok.\\n");
        exit(1);
    }
    if (nerr == 0)
        KodYarat();
    else
        fprintf(stderr, "Kod Yaratilmadi.\\n");
<<<<<<< BeniDegistir.c
    exit(nerr == 0 ? CIKIS_BASARILI : CIKIS_BASARISIZ);
=======
    exit(!!nerr);
>>>>>>> 1.6
}
```

Karakterler `<<<<<<<` ve `=======` arasına gelen satırlar, bizim eklememiz. `====¬ ===` ve `>>>>>>> 1.6` arasına gelenler depoda bizden önce yapılan değişikliklerdir. Bu birleşmiş dosyaya bakarak, bizim eklememizin geçerli olup olmadığına bakmalıyız, ve dosyayı son haline getirmeliyiz. Bunu yaparken `====` işaretlerini çıkartartmamız gerekecektir, ve öteki değişikliği yapan programcıya değişiklikleri hakkında soru sormamız gerekebilir. Bundan sonra, en son formu bulup, şu komutu tekrar işletmemiz gerekecektir.

```
cvs commit BeniDegistir.c
```

Fakat artık cvs komutu başarıyla tamamlanacak. Depoda artık güncel dosyanız bulunuyor!

A.10.2 CVS ve Binary Dosyalar

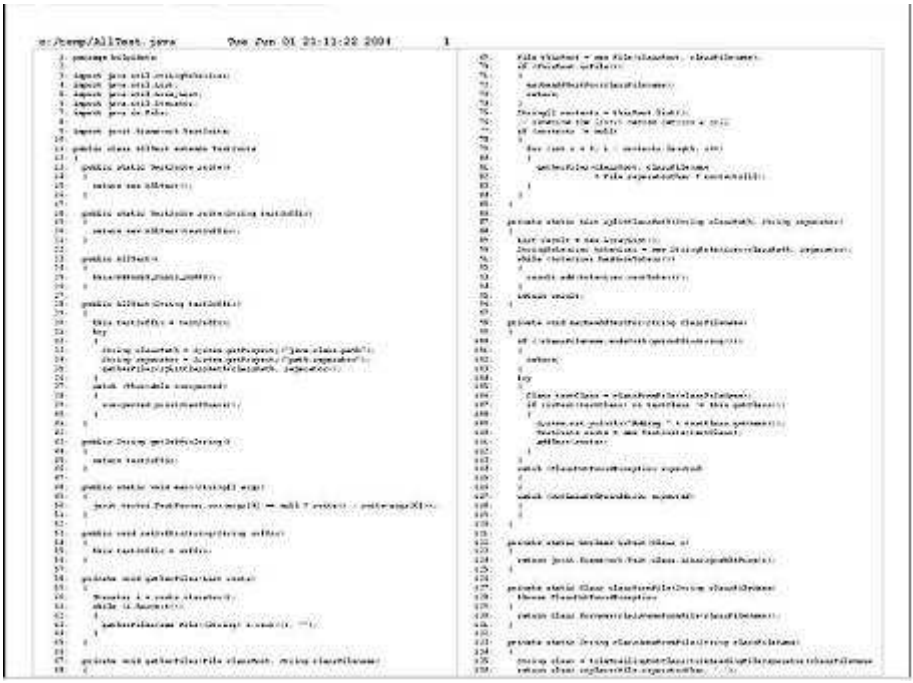
CVS'te en sık karşılanan problemlerden biri, `gif`, `jpg` ve `jar` gibi ikisel (binary) dosyaları CVS'e ekleyip, geri alınca bu dosyaların bozulduğunu görmektir. Bunun sebebi, CVS'in bu dosyaları metin bazlı dosyalar gibi görmesi ve üzerinde değişim yaparken dosyayı bozmasıdır.

Bu hatayı düzetlmek için, CVS'i kurduğunuz makine ve dizine girip şu değişikliği yapın. `CVS_DIZIN/CVSROOT/cvswrappers` adlı bir dosya bulacaksınız. Metin bazlı dosyadan daha değişik muamele görmesini istediğiniz dosyaları, şu şekilde cvswrappers içine ekleyin.

```
*.gif -k 'b'
*.jpg -k 'b'
*.jar -k 'b'
..
```

Bu yeterli olacaktır. CVS'i tekrar başlatmanıza bile gerek yoktur. Şimdi yeni ekleyeceğiniz dosyalar doğru muamele ediliyor olacaklar. Eğer daha önceden eklenmiş dosyaları düzeltmek istiyorsanız, `cvs remove`, `cvs commit` ve arkasından `cvs add` ve `cvs commit` ile bu dosyaların düzgün hâlini tekrar ekleyin.

A.11 Enscript



Kod gözden geçirmek için en rahat yol, kodu yazıcıdan basmak, ve üzerine yorumlar yazarak kodu bu şekilde gözden geçirmektir. Kod gözden geçirme (code review) toplantıları için kodu kağıda basmak, ayrıca önem taşımaktadır. Kodu gözden geçirilen programcı da toplantı sırasında kendisine verilen yorumları, düzeltmeleri kendi yazıcı çıktısı üzerine yazarsa, kendi için yararlı olacaktır.

Yazıcıdan alınacak çıktıdaki kodların "2 kod sayfası bir yazıcı sayfasına" gelecek, ve her satırın numaralı olacak şekilde alınması en iyisidir. Bu sayede, kod gözden geçirme toplantısı sırasında "..şu, ve şu hata, şu no'lu satırda" gibi sözler söylenebilecektir. Bu türden bir çıktının alınması için kullanılan program, `enscript` programıdır.

Enscript'i Windows'da kurmak için kitap kodlarımız `kitap-tools-enscript.zip` dosyasını açtıktan sonra en üst dizin `enscript` dizinini olduğu gibi alıp `c:` seviyesine kopyalayın. Bu kopyalama işleminden sonra `PATH` değişeninize `c:/enscript` dizinini ekleyebilirsiniz. Cygwin'i A.6 bölümünde anlatıldığı gibi kurmuş olanlar için hiçbir ek işleme gerek yok, `enscript` zaten kurulmuş olarak gelecektir. Artık komut satırında herhangi bir yerden `enscript` komutunu kullanabilirsiniz.

Şimdi sıra Ghostview ve Ghostscript kurmaya geldi. Postscript dosyaları Unix'çilerin uzun süredir bildiği bir formattır. Enscript programı formatladığı kaynak kodları görsel olarak biçimlendirip, çıktıyı postscript olarak yazdığı için, Ghostview adında bir gösterici programa ihtiyacımız var.

`ftp://mirror.cs.wisc.edu/pub/mirrors/ghost/ghostgum/gsv46w32.¬ exe`) ve `ftp://mirror.cs.wisc.edu/pub/mirrors/ghost/AFPL/gs814/gs8¬ 14w32.exe` bağlantılarından Windows için Ghostview ve Ghostscript'i indirebilirsiniz.

Bu kadar! Artık istediğiniz kodu güzel basmak için aşağıdaki şekilde bir komut verebilirsiniz. Enscript'in birçok seçeneği vardır, bunlari `enscript -¬ -help` ile öğrenebilirsiniz. En standart kullanım, 2 kod sayfası bir yazıcı sayfası, satırlarda numaralar, en başta kod isminin ismi başlık olarak ve Java sözdiziminde önemli olan anahtar kelimelerin (keyword) koyu olarak basıldığı formattır.

```
enscript -pcikti.ps -r -c -C -2 -j --pretty-print=java
        -r -v --lines-per-page=90 CLASS.java
```

A.12 Emacs

Emacs dünyada en çok fazla dili destekleyen editör'lerden biridir. Teknik lider olarak işiniz shell script'lerden Java'ya, oradan XML'e oradan da Perl'e atlamanızı gerektireceği için, her türlü ihtiyacınızı karşılayabilecek bir editör'e ihtiyacınız olacak.

Kurmak için `http://ftp.gnu.org/gnu/windows/emacs/latest/` adresinden Emacs'in son versiyonun indirin. Bu zip dosyasını bir dizinde açın (windows üzerinde genellikle `c:/` Unix'de genellikle `/usr/local/`). Kurduğunuz dizinin yerine `EMACS_DIR` olarak referans edersek, Windows'da kuruluşu tamamlamak için `EMACS_DIR/bin/addpm.exe` dosyasına çift tıklamanız gerekmektedir.

Bizim kullandığımız ayarları kullanmak/almak için, kitap kodlarımız arasında bulunan `kitap-tools-emacs-xxx.zip` dosyasını indirin ve Windows'da `c:/` üzerinde açın. Bu açılım, `c:/` altına bazı `.el` dosyaları ve `c:/emacs-21.4` dizini altına bir takım ek dosyalar bırakacaktır. Bizim kullandığımız Java geliştirme ortamı ve bir süredir toplamış olduğumuz, beğendiğimiz ayarlar bu dosyalar içindedir.

A.12.1 Emacs Özellikleri

Mod'lar

Emacs, her dosya için belli bir mod açabilir. Mod'lar o anda edit etmekte olduğunuz dil için özel bazı yetenekler sağlarlar. O dile özel girintilendirme (indentation), anahtar kelime renklendirmesi (syntax coloring), yardımcı tuşlar

bu yardımlardan bazılarıdır. Hangi dosya için hangi mod kullanılması gerektiği `_emacs.el` içinde tanımlanabilir.

```
(setq auto-mode-alist
      (append '(("\\.C\$" . c++-mode)
                ("\\.xml\$" . nxml-mode)
                ("\\.java" . jde-mode)
```

Görüldüğü gibi Emasc XML dosyalarını sonek `.xml` sayesinde anlamaktadır, ve bu dosyaları `nxml-mode` açmaktadır.

Her mod kendi dili için girintileme (indentation) yapabilir. Indent için istediğiniz bölgeyi mouse ile seçin, ve `MyJDE | Indent`, ya da `\C-x\=` tuşlarını kullanın. Ayrıca herhangi bir satır üzerindeyken (satırın neresinde olursanız olun) TAB'e basarak girintileme yapabilirsiniz. Bu özellik çoğu editör'de mevcut değildir.

`M-` sembolü, Emacs dünyasında Escape tuşu olarak bilinir. Yerine `ALT` tuşu basık tutularak ta kullanılabilir.

Dosya açmak için `File | Open File`, ya da `\C-x\C-f`, kaydetmek için `File | Save`, ya da `\C-x\C-s` kullanılır.

Dired

Dizin gezmek için Dired mod'u kullanılabilir. Dired kelimesi, **Dir**ectory **ed**itor kelimelerinden gelir. `File | Open Directory` ya da `\C-x\d <ENTER>` ile girebilirsiniz. Dired size o anki dizinin listesini verir. Bu listede

- Aşağı yukarı gitmek için "`n`" ve "`p`" tuşlarını
- Dosyaya girmek için `f` tuşunu
- Ya da mouse ile dosyaya işaret ederek sol düğmeyi

kullanabilirsiniz. Eğer Dired'de bakmakta olduğumuz dizin üzerinden bir Explorer penceresi açmak istiyorsak, bunun için `MyJDE | Open Explorer In Current Dir` menü seçeneğini kullanabiliriz (ya da `F10` tuşu). Aynı şekilde `MyJDE | Open Cmd In Current Dir` Dired'deki dizinde bir komut satırı (shell) açacaktır.

JDE

Çok kuvvetli bir Java mod'udur. Herhangi bir `.java` dosyasına girdiğiniz anda aktif olur. Projenizin en üst seviyesinde `prj.el` varsa, size daha yardımcı olacaktır. `prj.el` şöyle gözükebilir;

```
(custom-set-variables
 '(jde-built-class-path
   (quote ("./lib/hibernate3.jar"
     ...
     )))
```

```
'(jde-complete-function (quote jde-complete-menu))
'(jde-global-classpath
  (quote ("./lib/hibernate3.jar"
    ....
    )))
'(jde-sourcepath
  (quote ("./src/java"
    )) )
)
```

Üstteki ayarlar üzerinden, bir metotu çağıranları bulmak için `F4`, `MyJDE | Call Tree (Usage)`, işleç üzerindeyken bir class tanımına gitmek için `F3`, `MyJDE | Goto` kullanılır.

Bir class'ı otomatik import etmek için, class'ın üzerine gidip `MyJDE | Import Class`, ya da `\M-uv` kullanabilirsiniz. Bir referansın metotlarını listelemek için, noktadan sonra `F7` ya da `MyJDE | Complete` kullanılabilir.

Tüm bu Java yardımcıları, `prj.el` içinde jar'larını koyduğunuz dosyalar ve dizinler üzerinde çalışır.

Derlemek

Bizim ayarlarımızı kurduysanız, `MyJDE | Compile` ya da `\C-x\c`, otomatik olarak `ant -emacs -find` komutunu işletecektir. Çıktı, Emacs tarafından taranır. Hatalara gitmek için `\C-x\~`. Ya da hatanın üstüne gidip `\C-c\C-c` kullanabilirsiniz.

Kopyalama, Yapıştırma, Silme

- Kopyalama için `\C-c`
- Yapıştırma için `\C-y`
- Geri tek karakter silmek için `\C-k`
- Satır silmek için `\C-t`

Hareket Tuşları

- Yukarı: `\C-p`
- Aşağı: `\C-n`
- Sola: `\C-j`
- Sağa: `\C-l`
- Kelime sağa: `\C-d`
- Kelime sola: `\C-w`

- Kelime sola silmek: `\C-f`
- Belli bir satıra gitmek: `\C-x\g`

Buffer'lar

Emacs'de her dosya bir buffer içinde tutulur. Buffer'lar arasında gezinmek için `Buffers` menü seçeneğinden (ya da `\C-x\C-b` ile) ya da `CTRL` artı sol mouse düğmesine tıklayarak tüm mevcut buffer'ları görebilirsiniz

A.12.2 Emacs ve CTRL tuşu

Emacs'de CTRL tuşu çok kullanılır. Normal klavyelerde, CTRL çok erişilmez bir yerde olduğu için çözüm `CAPS LOCK` tuşunu `CTRL`'a çevirmektir. Alttaki kodu `ctrl.reg` olarak kaydedip, ve üzerine çift tıklarsanız, bilgisayarı kapatıp açtıktan sonra `CAPS LOCK` tuşunun `CTRL` tuşuna dönüşmüş olduğunu göreceksiniz.

```
REGEDIT4
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Keyboard Layout]
"Scancode Map"=hex:00,00,00,00,00,00,00,00,02,00,00,00,1d,00,3a,00,00,\
00,00,00
```

Not: \ işareti satırın bitmediğini göstermek için bizim tarafımızdan koyulmuştur, normâl dosyanın içine bu karakteri koymaya gerek yoktur.

A.12.3 Ayar Değişiklikleri

Ayar değişiklikleri için `_emacs.el` dosyasını istediğiniz gibi değiştirebilirsiniz. Sonra `\C-x\C-b` ile derleyin ya da `\M-x byte-me` LISP komutunu işletin. Yeni değişiklikler anında etkili olacaktır.

Değişik renkler kullanmak istiyorsanız, `_emacs.el` ile aynı seviyede olan `_colors-public.el` dosyasında gereken değişiklikleri yapabilirsiniz.

A.13 Cygwin Üzerinde X Windows

Cygwin/X kurmadan önce Cygwin'i kurmuş olmanız gerekecektir. Eğer bu kurulumu yaptıysanız, `http://x.cygwin.com/` adresinden `Install Cygwin/¬ X Now` bağlantısını tıklayın. Bir `setup.exe` programı ortaya çıkacaktır, bunu sabit diskinizde bir yere kaydedin. Bu programın üzerine tıklayıp, `Next`'e basın, ve "Install from Internet" seçimini yapın. Cywgin `Root Directory` olarak Cygwin'inizin nerede olduğu algılacacaktır, eğer algılanmadıysa doğru dizin yerini siz girin. Sonra iki kere `Next` ile kurulumu yapın.

Herşey doğru kurulduktan sonra, komut satırından

```
startx
```

ile X altyapınızı başlatabilirsiniz. Bu altyapı başladıktan sonra, bir (üzerinde X servisi işleyen) bir Unix makinasına bağlanıp, X programlarının çıktısını kendi makinanıza alabilirsiniz.

A.14 Kaynak Kod Yamaları

Bazıları, açık kaynak kültürünün başarısını her projenin kodunun erişilebilir (okunabilir) olmasından hareketle, aynı zamanda kodun *herkes tarafından* CVS deposunda değiştirilebilir olması olduğunu zanneder. Halbuki, birçok aktif açık yazılım destekçilerinin çoğunun bir projeye hiçbir zaman CVS commit hakkı olmamıştır[1]. Peki o zaman "açık yazılımda test'çilerin fazlalığı ve hata onarımının çabukluğu" nasıl mümkün olmaktadır, ve bu ne demektir?

Burada kritik teknoloji, yama (patch) teknolojisidir. Bir yama dosyası, en basit anlatımıyla, bir text dosyasıdır. Bu dosya, programcı yamayı oluşturacak komutu verdikten sonra, kod geliştirme dizininde programcının yaptığı değişiklikler ile kodun eski, değişmemiş hâli arasındaki *farkların* toplamıdır. Bu değişiklikler, bir ya da daha fazla dosya üzerinde dağılmış olabilir; Yama dosyası bunun takibini doğru bir şekilde yapacaktır.

Yama dosyası, text bazlı bir dosya olduğu için, bir forum'a asılabilir, e-mail ile başka bir programcıya gönderilebilir, istenilen ortamda saklanabilir. Eğer bu yamayı başka bir programcıya alıp kullanmak isterse, bu programcı, kendi kod bazı üzerinde yamayı uygular (applying the patch) ve böylece değişiklikler, onun kod bazında da aktif hâle gelir. Bu uygulama işleminin güzel tarafı, uygulamayı yapan programcının önceden kendi yaptığı (farklı) değişiklikleri bile olsa, yamadaki değişikliklerin çoğunlukla başarıyla uygulanabilmesidir, çünkü değişikliklerin hangi dosyada olduğu yama çıkartma tarafından kaydedilmiştir, tek potansiyel problem aynı dosya üzerinde iki taraf ta değişiklik yapmışsa ortaya çıkar; Fakat bu şartlarda çakışmalar bir birleştirim (merge) ile çözülebilir.

Yama çıkarmak için `diff`, yama uygulamak için `patch` komutu kullanılır. Yama çıkartmak için kodun öncesinin bilinmesi gerekir; Bu "önce" kod bazı, ya bir CVS'e `anonymous` erişim üzerinden, ya da programcının sabit diskinde mevcut bir kod dizini üzerinden erişilebilir. Genellikle uygulanan yöntem CVS üzerindendir (fakat CVS yoksa, bir alternatif olduğunu bilmek iyidir).

Açık yazılım projelerinde yamalar, genelde önce kullanıcı ve programcı mailing list'lerine asılır, ya da, projenin herkese açık olan hata takip sistemine eklenir. Bu hataya eklenti (attachement) olarak tamiri gerçekleştirecek olan yama dosyası da eklenir.

Açık yazılım projesinin CVS'e commit hakkı olan çekirdek geliştiricileri, sürekli bu hata takip sistemini takip ediyor olurlar. Böylece en son girilen hatayı farkeden çekirdek gruptan bir programcı, yamayı hemen kendi kod bazına uygulayıp hakikaten bir hatayı iyi edip etmediğini kontrol edecektir. Eğer hata hakikaten düzeltilmişse, testi yapan programcı yamadan gelen değişiklikleri

[1] http://www.opensymphony.com/oscache/contribute.action

CVS'e ekler (bunu yapmaya hakkı vardır), ve arkasından hata takip sistemindeki hatayı kapatır.

A.14.1 Yama Üretmek

CVS üzerinden yama üretmek için, projenin en üst seviyesinden

```
cvs -q diff -u > /tmp/patchfile.patch
```

şeklinde bir komut kullanılabilir. Eğer CVS erişimi yoksa, şu komut kullanılır.

```
diff -rup /path/to/unmodified/source /path/to/modified/source >
     /path/to/patchfile.patch.
```

Yeni dosyaları yamaya eklemek için ise:

```
diff -u /dev/null MyCoolNewFile.java >> /path/to/patchfile.patch
```

A.14.2 Yama Uygulamak

Yamayı uygulamak için, CVS'ten en son kod alındıktan sonra, projemizin en üst seviyesinden `patch` komutunu şu şekilde kullanmalıyız.

```
patch -Np1 < /path/to/patchfile.patch
```

Eğer "missing header for unfiled diff at line 8 of patch" gibi hatalar geliyorsa, şu da denenebilir.

```
patch -Np0 < /path/to/patch.diff
```

Ek B

Düzenli İfadeler

Düzenli ifadeler (regular expressions), bir metin içinde belli bir düzene uyan bir kelime dizisini bulmak için kullanılır. Teknik olarak bu işlemin sonucundan bazen şu şekilde bahsedilir: "xyz düzenli ifadesi abc ifadesine uydu (matched)". İfade, aslında bir kelime dizisini temsil eden genelci bir beyandır; Aslında komut satırından bir düzenli ifadeyi sürekli kullanıyoruz. '`*`' işareti. Bu işareti bir listeleme komutu ile beraber kullandığımızda, '`*`' işareti "ne olursa olsun" düzenli ifadesi olduğu için, bu tarife uyan tüm dosyaları ve dizinleri geri almış oluruz.

Fakat düzenli ifade '`*`' işaretinden ibaret değildir. Çok daha güçlü ve sayesinde değişik ve çetrefil metinler çekip çıkarabileceğimiz ifadeler vardır. Ama ondan önce, üzerinde düzenli ifade işleyeceğimiz girdi dosyalarını nasıl okuyacağımızı görmemiz gerekiyor, yâni Perl'ün dosya erişim özelliklerini.

B.1 Perl ile Metin İşleme

Perl'ün metin işleme yeteneklerini bir örnek üzerinde görelim: Perl ile dosya açmak için `open` komutu kullanılır.

```
open IN, ‘‘fileName.txt’’;
```

Bu komutla `fileName.txt` okunmak için açılmış olur. Bir dosyayı satır satır, ya da tamamen okuyabiliriz. Çoğunlukla (meselâ kodda) yapılan metin değişiklikleri için dosyanın tamamını aynı anda okumak gerekir. Bunu yapmak için, `undef \$/;` işareti kullanmamız gerekecektir. Bu tanımdan sonra tüm dosya içeriğini bir değişkene atamak için, şu kullanılır.

```
\$fileContent = <IN>;
```

Bu yapıldıktan sonra `\$fileContent` değişkeni, dosyadaki tüm içeriği içinde barındırmaya başlar. Kodlamanızı daha kısa tutmak istiyorsanız, `\$fileContent` yerine özel bir değişken olan `\$_` değişkenini de kullanabilirsiniz. Gizli değişkeni kullanmanın iyi taraflarından biri, takip eden tüm düzenli ifade işlemlerinin (eğer onlar da değişken belirtmemişse) otomatik olarak gizli değişken üzerinde yapılabilmesidir. Çok basit örnek bir düzenli ifade kullanmak gerekirse

```
open IN, ‘‘fileName.txt’’;
\$_ = <IN>;
s/ahmet/mehmet/sg ;
print;
close IN;
```

Bu örnekte, “`ahmet`” kelimesini “`mehmet`” ile değiştirdik, ve ekrana bastık. `print` komutuna hiç parametre vermezseniz, aynı şekilde gizli değişken olan `\$_` kullanıldığını farz edecektir.

İlk düzenli ifademiz, hem bulan, hem değiştiren bir ifade. Bu ifadenin `s/¬ string1/string2/sg;` tabirinde olan `string1`, aranan kısımdır, `string2` ise, aranan ifade bulununca, yerine geçecek kısımdır. Değiştirmek yerine sadece bulmak istiyorsanız, `string1` ifadesini sadece `if` ile kullanabilirsiniz; `if (/¬ string1/) {.. .}` gibi.

B.1.1 Çıktı Dosyası

Metin değişimi yapmak için genellikle değiştirilen dosya ikinci bir dizin altında aynı isimde yazılır.

```
open IN, ‘‘fileName.txt’’;
open OUT, ‘‘/tmp/fileName.txt’’;
\$_ = <IN>;
s/ahmet/mehmet/sg ;
print OUT;
close IN;
close OUT;
```

Bu kod parçasındaki `OUT`, değişmiş içeriğimizin gittiği dosya olacaktır. Dosyayının aynı isimle açıldığı dizinin değişik bir yerde olduğuna dikkat edelim.

B.1.2 Birçok Giriş Dosyası

Eğer bir dizin altındaki birçok dosyayı işlemek istiyorsak, dosya listesi almak için `<>` işaretlerini kullanabiliriz. Bu işaretlerin arasında, çoğu işletim sisteminden alışık olduğumuz ‘`*`’ kullanımı mümkündür. `<>` kullanımında geriye bir Perl listesi gelecektir, bu listeyi `foreach` Perl komutu ile gezebiliriz. Meselâ

```
foreach \$file(<*.java>) {
  # ..
  # \$file ile işlemler yap
```

```
  # ..
}
```

Eğer biraz önceki değiştirme mantığını üstte gösterilen tüm dosyalara uygulamak istersek,

```
foreach \$file(<*.java>) {
  open IN, ‘‘\$file’’;
  open OUT, ‘‘/tmp/\$file’’;
  \$_ = <IN>;
  s/ahmet/mehmet/sg ;
  print OUT;
  close IN;
  close OUT;
}
```

Dosyaların okunacağı dizini değiştirmek için ise, Perl `chdir` komutu kullanılabilir; `chdir(‘‘/vs/vs’’)` gibi.

B.2 İfadeler

Artık daha zor düzenli ifadeleri gösterebiliriz. Bu yeni daha çetrefil ifadeleri `//` arasına yerleştirirsek, bu yeni ifadeleri kullanmiş oluruz. Böylece o ifadenin temsil ettiği metne göre, o anda okunmakta olan dosya üzerinde bu verilen ifade *uydurulmaya* çalışılacaktır. Uydurulan (bulunan) ifade ise, otomatik olarak değiştirme bölümünü devreye sokacaktır.

Düzenli ifadeler içinde her çeşit metin dizisi için, bir komut vardır. Meselâ eğer `22.33.22`, ya dâ, `33.22.44` yâni, genel olarak, “iki sayı, sonra bir nokta, iki sayı daha, sonra nokta ve son olarak iki sayı” içeren bir *düzeni* bulmak istersek, o zaman tek vermemiz gereken düzenli ifade şu olacaktır.

```
/\d{2}\.\d{2}\.\d{2}\./
```

Burada `\d` komutu, tek haneli bir sayıyı temsil etmektedir. Tek karakteri temsil eden tüm özel komutların listesi Tablo B.1 üzerinde bulunabilir.

Tablo B.1 komutları, elle gömeceğimiz sabit metin ile aynı ifadenin parçası olarak kullanılabilir. Ayrıca çoğu zaman, karakter komutlarını “bazı özel tekrar etme kuralları” ile beraber kullanmak gerekiyor. Bu özel tekrar kuralları, genellikle, bir düzenli ifade komutunun hemen yanına eklenir, ve yanına eklendiği tabirin temsil gücünü daha da arttırır. Tablo B.2 üzerinde tekrar kurallarını görüyoruz.

Örnekler

`\$_ = ‘‘abbbccdaabccdde’’` üzerinde bazı düzenli ifade örnekleri görelim[1]:

[1] Can Uğur Ayfer, `http://www.mycompany.com/localhost/mycompany/yazi.jsp@dosya=a_regular_expression.xml.html`

Tablo B.1: Düzenli İfade Komutları

Komut	**Târif**
`\d`	Tek haneli bir sayı
`\D`	Tek haneli haricinde her şey
`\w`	Bir alfabe harfi (a ile z, A ile Z arası, ve _ işareti)
`\w`	Alfabe haricinde herhangi bir karakter
`\s`	Beyaz boşluk (SPACE, TAB) karakteri
`\S`	Beyaz boşluk haricinde herhangi bir karakter
`\`	META (ESC) karakteri
`^`	Satır başlangıç karakteri
`.`	Herhangi bir karakter
`\$`	Satır sonu karakteri

Tablo B.2: Tekrar Komutları

Komut	**Târif**
`*`	0 ya da daha fazla
`+`	1 ya da daha fazla
`?`	1 ya da hiç
`{n}`	n kere
`{n,}`	en az ne kere
`{n,m}`	en az ne kere, ama m'den fazla değil
`str1 | str2`	Ya ifade1 ya da ifade2 bulunacak

- `/Abc/` Uymaz! `\$d` içinde hiç "A" yoktur yâni "Abc" bulunamaz.
- `/Abc/i` Uyar! Sondaki "i" ignore case, yani büyük-küçük harf ayırımı yapılmasın anlamında olduğu için "Abc" ile "abc" eşleşir.
- `/abc/` Uyar!
- `/^abc/` Uymaz! Çünkü "^" meta-karakteri dizinin başında anlamındadır ve "abc" alt dizisi `\$d`'nin basinda değildir.
- `/abc>/` Uymaz! Çünkü ">" meta-karakteri dizinin sonunda anlamındadır ve "abc" alt dizisi `\$d`'nin sonunda değildir.
- `/De>/i` Uyar! Dizimizin sonunda "de" var ve büyük-küçük harf ayırımı yapılmayacak!
- `/^ab*c/` Uyar, çünkü dizinin basinda "a" ve ardından "sıfır veya daha fazla b" ve ardından "c" var! Bu düzenli ifadede olan "*" karakteri hemen solundaki karakter için "sıfır veya daha fazla kez tekrar eden" anlamında bir meta karakterdir.
- `/aH*bc/` Uyar! Çünkü dizide "a" ve ardından sıfır veya daha fazla "H" ve ardinda "bc" var!
- `/aH+bc/` Uymaz! Çünkü dizide "a" ve ardından en az bir tane "H" ve onun ardında "bc" seklinde bir düzen yok! Bu Duzenli ifadede olan "+" karakteri hemen solundaki karakter için "bir veya daha fazla kez tekrar eden" anlamında bir meta-karakterdir.
- `/a*bc/` Uymaz! Çünkü dizinin içinde hiç "a" ve ardından gelen "*" yok! Bu örnekte "*" bir meta-karakter olarak değil; basit anlamıyla bir asterisk karakteri olarak kullanıldığı için önündeki "" ile işaretlenmiştir.
- `/a+bc/` Uymaz! Çünkü dizinin içinde hiç "a" ve ardindan gelen "+" yok! Bu örnekte "+" bir meta-karakter olarak değil; basit anlamıyla bir artı işareti olarak kullanıldığı için önündeki "" ile işaretlenmiştir.
- `/a c/` Uymaz! Çünkü dizinin içinde hiç "a" ve ardindan gelen "" yok!
- `/a.*b/` Uyar! Çünkü dizide "a ve aralarında birşeyler ve sonra b" var! "." (nokta) meta-karakteri herhangi bir karaktere uyar; ardından gelen "*" ile birlikte (yâni "`.*`") birşeyler olarak okunabilir.
- `/d.*a/` Uyar! Çünkü "birşey" anlamındaki noktanın ardından gelen "*" meta-karakteri sıfır veya daha fazla herhangi birşey anlamındadır.
- `/d.+a/` Uyar! Çünkü "birşey" anlamındaki noktanın ardından gelen meta-karakter bir "`+`"; Yâni bir veya daha fazla "herhangi birşey"
- `/da?/` Uyar! Çünkü dizide "d" ve ardından bir veya sıfır tane "a" gelen alt dizi var.

B.3 Gruplama ve Bulunanı Kullanmak

Çoğu değiştirme işlemi için, bulmak için kullandığımız düzenli ifadenin bulduğu metnin "bir bölümünü" yeni değiştirmenin bir parçası olarak kullanmamız gerekebilir. Meselâ, alttaki gibi bir kod parçası üzerinde

```
import com.sirket.paket.vs;

if (bilmemne)
   soyle boyle;

soyle boyle;

if (filan)
   soyle boyle oyle;
```

yapacağımız değişiklik şöyle olabilir; "sadece `if` komutunu içeren kod satırları (`;` işareti ile biten kod satırları, dosya satırları değil) içinde bütün "şöyle böyle" ibarelerini, "böyle şöyle" yapalım". Bu gibi durumlarda, bulunan kelimenin kendisi yeni kelimenin bir parçasıdır, bu yüzden bir şekilde onlara erişebilmemiz gerekir. Bunun için Perl'ün `()` gruplama işaretlerini kullanabiliriz. Perl'de her düzenli ifade komutu, parantez içine alınabilir. Alındığı zaman, ve bu ifade metin parçası ile uyuştuğu zaman (matching), parantez içindeki uyan kısım, değiştirme yapan (ikinci `//` arasındaki komutlar) kısımda, gruplamanın kullanılış sırasına göre, `\$1`, `\$2`, `\$3`, .. ile erişilebilir. Meselâ örneğimizde üç tane `()` kullanımı var, birinci uyan kısma erişmek için `\$1` kullanacağız.

```
s/\;(.*?)if (.*?) soyle boyle(.*?)\;/\;\$1if \$2 boyle soyle\$3\;/sg;
```

Kaynakça

[1] King, G., Bauer, C. *Hibernate In Action.* Manning Publications, Greenwinch, 2005.

[2] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading Mass., 1995.

[3] Meyer, B. *Object Oriented Software Construction, Second Edition.* Prentice Hall, New Jersey, USA, 1997.

[4] Chappel, D., Monson-Haeffel, R. *Java Message Service.* O'Reilly, Sebastapol CA, 2001.

[5] JBoss, Inc. *JBoss Admin Development Guide - JBoss 3.2.6.* JBoss, Inc., 2004.

[6] Burke, B. *JBoss Clustering.* JBoss Group, Atlanta, GA, 2003.

[7] Looseley, C., Douglas, F. *High-Performance Client/Server.* Wiley Computer Publishing, 1997.

[8] Hunt, C. *TCP/IP Network Administration.* O'Reilly, 1998

[9] Codd, E. F. *A Relational Model for Large Shared Data Banks.* CACM, 1:6, 1970

[10] Elmasri, R. Navathe, B. S. *Fundemantals of Database Systems, Second Edition.* Addision-Wesley, 1994

[11] Brooks, F. P. Jr. *The Mythical Man-Month, 1995 Anniversary Edition.* Addision-Wesley, 1995

[12] Lewin, R. *Complexity, Life at the Edge of Chaos, Second Edition.* Phoenix, 2001

[13] Nixon, R. *Six Crises.* Touchstone Press, New York, 1990

[14] Torvalds, L., Diamond, D. *Yanlızca Eğlenmek İçin (Orijinal: Just For Fun)*. Bilgi Yayınevi, 2005, Orijinal: Harper Business, New York, 2001